RICE UNIVERSITY

Efficient Distributed Shared Memory Based On Multi-Protocol Release Consistency

by

John B. Carter

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Willy Zwaenepoel, Chairman
Professor of Computer Science

Alan L. Cox
Assistant Professor of Computer Science

John K. Bennett
Assistant Professor of Electrical and
Computer Engineering

Houston, Texas

January, 1994

ABSTRACT

Efficient Distributed Shared Memory Based On Multi-Protocol Release Consistency

by

John B. Carter

A distributed shared memory (DSM) system allows shared memory parallel programs to be executed on distributed memory multiprocessors. The challenge in building a DSM system is to achieve good performance over a wide range of shared memory programs without requiring extensive modifications to the source code. The performance challenge translates into reducing the amount of communication performed by the DSM system to that performed by an equivalent message passing program. This thesis describes four novel techniques for reducing the communication overhead of DSM, including: (i) the use of *software release consistency*, (ii) support for *multiple consistency protocols*, (iii) a *multiple writer protocol*, and (iv) an *update timeout mechanism*. Release consistency allows modifications of shared data to be handled via a *delayed update queue*, which masks network latencies. Providing multiple consistency protocols allows each shared variable to be kept consistent using a protocol well-suited to the way it is accessed. A multiple writer protocol addresses the problem of *false sharing* by reducing the amount of unnecessary communication performed to keep falsely shared data consistent. The update timeout mechanism reduces the impact of updates to stale data. These techniques have been implemented in the Munin DSM system. The impact of these features is evaluated by comparing the performance of a collection of shared memory programs running under Munin with equivalent message passing and conventional DSM programs. Over half of the shared memory programs achieved at least 95% of the speedup of their message passing equivalents. For the other programs, the performance bottlenecks were removed via minor program modifications. Furthermore, Munin programs achieved from 25% to over 100% higher speedups than equivalent conventional DSM programs when there was a high degree of sharing. The results indicate that DSM can be a viable alternative to message passing if the amount of unnecessary communication is minimized.

Acknowledgments

This thesis is the culmination of many years of hard work, but it would not have been possible without the help of many other people. It is not possible to list all the people who have helped me make it through the past six years, but I would like to take this opportunity to thank as many of you as I can.

First, I want to thank my advisor (Willy Zwaenepoel) and the other members of my committee past and present (John Bennett, Hans Boehm, and Alan Cox) for their assistance with the research discussed in this thesis. More importantly, I want to thank them for teaching me what constitutes quality research, how to communicate my research results, and what it takes to be a professor. There have been times when we have not seen eye to eye, but even on those occasions I learned a lot. I will especially remember the trips to Goode Company Barbeque with John where my best brainstorms occurred. Thank you!

Second, I would like to thank all of my friends who helped me to stay sane and enjoy life. I am in debt to many of you, including: Becky, Linda, Mootaz, Vernon, Alan, Mark, Mary, Kathryn, Clara, Sunil, Jennifer, Matt, G.T., Pete, Dave, Rick, Doug, Jacob, Steve, Chau-Wen, Preston, Kelly, Robin, and many others. When I was down, you lifted me up, made me smile, and gave me the strength to persevere.

Third, I would like to thank the National Science Foundation and NASA for their generous fellowship support, which enabled me to pursue this research.

Last, but definitely not least, I would like to thank my family for being there and giving me the support necessary to succeed in a venture of this magnitude. I would especially like to thank my mother, who was always there when I needed somebody to talk to, and my father, who motivated me to pursue a PhD and to whom this dissertation is dedicated. Mom, Dad, Vicky, and Bill – I love you!

Finally, I would like to close with a quote from President Theodore Roosevelt, who urges us all to strive towards lofty goals: "Far better is it to dare mighty things, to win great triumphs even though checkered by failure, than to take rank with those poor souls who neither enjoy much nor suffer much for they live in the grey twilight that knows neither victory nor defeat."

Contents

Illustrations

Preface

In the fifty plus years since Alan Turing postulated his theoretical computer, the entire field of computing has made tremendous progress. One could characterize this progress in terms of improvements in availability, cost, performance, size, and user-friendliness. Costs have fallen drastically for both hardware and software, performance has improved by several orders of magnitude, and ease of use has made computer literacy a virtual requirement for everyone.

Future performance improvements from hardware innovations are limited as we approach various physical limits, such as the time it takes an electronic signal to travel across a chip at the speed of light. Thus, future performance improvements must come from sources other than improved process technology. Much of the improvement will need to come from parallel processing, but unfortunately there are few dedicated multiprocessors and they are rather costly.

One attractive approach for circumventing the cost and availability problems of dedicated multiprocessors is to construct a *network multicomputer* by connecting a number of processors using conventional network technologies. The limiting factor to this approach is the complexity of programming the necessary message passing routines to share data, maintain its consistency, and synchronize processes. What is needed is an *efficient* distributed shared memory (DSM) runtime system that is easy to use and permits speedups approaching that of tailored message passing programs. To build such a system, the communication overhead of DSM must be reduced to be close to that of hand-coded message passing. The goal of this research was to explore the extent to which the performance of DSM could approach that of message-passing, without requiring programmers to substantially modify their shared memory programs.

Based on the observation that no single consistency protocol is best for all data access patterns, different consistency protocols are used for data with different access patterns. The key to making the system efficient turned out to be developing a means to overcome the problem of *false sharing*, which is common in shared memory parallel programs. This observation motivated the use of the more relaxed *release consistency* model as the basis for a software implementation of consistency and the development of a concurrent writer protocol. The implementation of these ideas is the Munin DSM system. To design a suite of consistency protocols that are well suited to support the various ways that shared data is characteristically accessed, I analyzed the data access patterns of a variety of programs on a 16-node shared memory multiprocessor. The programs displayed varying degrees of granularity, both in terms of the size and frequency of interprocessor sharing. Upon review, the number of access patterns was reduced to five, each with a unique consistency protocol designed to support it efficiently. With this background, I designed and implemented the prototype Munin runtime system for a distributed system consisting of sixteen SUN-3/60 workstations connected by a 10 megabit per second Ethernet.

The performance of the resulting system varied as problems of different granularity and with differing amounts of sharing were analyzed. In general, the strategy of basing the system on the release consistency model, supporting multiple consistency protocols to allow each shared variable to be kept consistent with a protocol well-suited to the way it is used, and attacking the problem of false sharing through a consistency protocol that allows multiple concurrent writers, was a good one. Munin was able to execute efficiently programs that exhibited a large amount of fine-grained sharing – programs that fared poorly under a conventional DSM system. Thus, this work extends the range of programs that can be solved effectively using DSM, which is important for DSM to become more widely used. As network speeds increase, as programmers become better at designing parallel programs, and as DSM systems become more efficient, the future of DSM will become increasingly bright and its use for "real" programs should increase. This will accelerate the rate at which distributed memory multiprocessors are used on problems other than trivially parallelizable scientific codes, and will help deliver on the promise of parallelism.

Chapter 1

Introduction

1.1 Background and Motivation

Future improvements in computer performance from hardware innovations are limited as we approach various physical limits. Thus, future improvements in power and speed must come from sources other than improved process technology (e.g., smaller components and faster clocks). The most promising source for performance improvement is parallel processing. There are two fundamental models for parallel programming: (i) shared memory and (ii) distributed memory.

The *shared memory model* is a direct extension of the conventional uniprocessor model wherein each processor is provided with the abstraction that there is but a single memory in the machine, so any change made to a piece of shared data is immediately visible to all the processors in the system. As there is but a single memory, each processor has an identical view of the state of each shared variable. For performance reasons, shared memory multiprocessors use a per-processor cache. The problem of keeping these caches coherent has been widely explored and hardware solutions have required complex and expensive hardware cache controllers [PP84, KEW$^+$85, AB86, TSS88, LT88].

In contrast, the *distributed memory model* does not support the abstraction that there is a single shared memory in the system, but instead maintains the abstraction that each processor has a private memory to which no other processor has direct access. The only way that processors can communicate is through explicit *message passing*. All communication on a distributed memory multiprocessor is explicit, so an expensive and complicated cache controller is not required. Thus, distributed memory multiprocessors are easier and cheaper to build, they scale better to large configurations, and they can be built from existing hardware (e.g., networks of workstations). The primary disadvantage of distributed memory multiprocessors is that they require programmers to use explicit message passing to communicate, while programmers generally prefer the shared memory model. For example, because distributed memory multiprocessors do not support a single shared address space spanning all the processors in the system, it is seldom possible to pass pointers meaningfully between processes executing on different processors.

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 basic idea behind DSM is to treat the local memory of a processor as if it were a coherent cache in a shared memory multiprocessor. Figure 1.1 illustrates this high-level model of a DSM system. The system consists of the same physical hardware as that found in a distributed memory multiprocessor, with the addition of a software layer, represented by the dashed line, that provides the *abstraction* of a single shared memory. In practice, each memory remains physically independent, and all communication takes place through explicit message passing performed by the DSM software layer.

Figure 1.2 illustrates how a typical DSM software layer implements this abstraction [LH89]. The DSM software manipulates each processor's virtual memory subsystem to perform the same type of operations, such as invalidations and detection of misses, on virtual memory pages as the cache controller of a shared memory multiprocessor performs on cache lines. Each page in the global address space is represented by a uniquely shaded box. The boxes at the bottom of the figure represent the pages that are present in each processor's local memory. In Figure 1.2, Proc1 has attempted

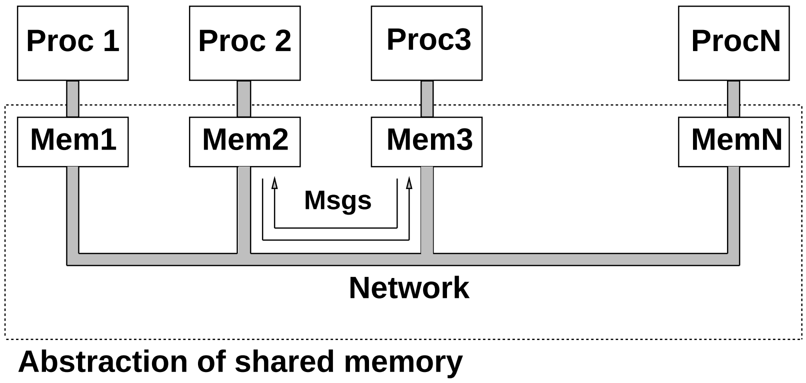

Figure 1.1 Example Distributed Shared Memory System

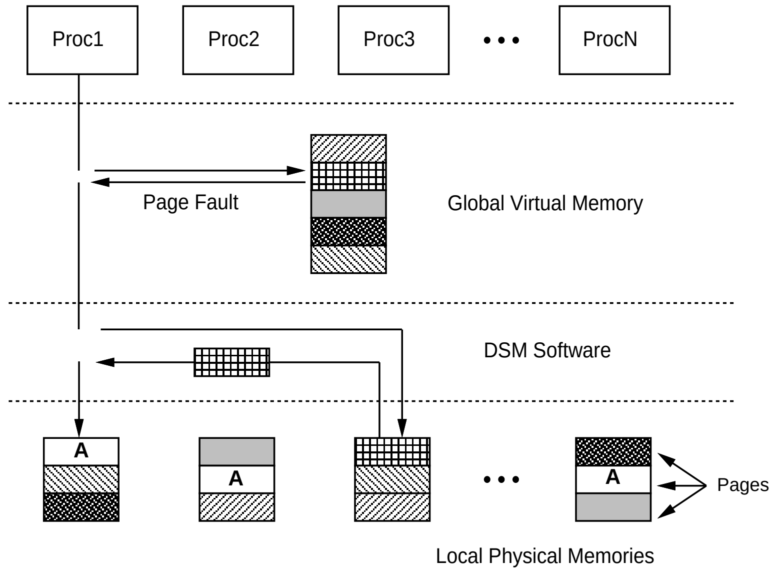

Figure 1.2 Example Distributed Shared Memory Implementation

to access a page of global virtual memory for which it does not have a copy. This results in a page fault that is caught and handled by the DSM layer, which retrieves a copy of the missing page from another node, in this case Proc3. The specific protocols used to emulate shared memory and maintain consistency differ from DSM system to DSM system. Since the DSM layer is implemented in software, whereas coherence in a shared memory multiprocessor is performed in hardware, DSM can implement protocols that are more complex or require more state than is reasonable for a hardware implementation.

Throughout the rest of this dissertation, the term *conventional DSM* [LH89] refers to a DSM system that employs a page-based write-invalidate consistency protocol (such as Berkeley Ownership [KEW+85]). In such a system, there can be multiple readable copies of any page of memory, but before a processor can write to a page of shared data, it must first invalidate all replicas of the page. For example, in Figure 1.2, the page represented by the boxes marked "A" is originally present on processors 1, 2, and N. These processors can read data contained in that page directly from the local copy with no DSM overhead. However, before a processor (e.g., Proc1) can modify the page, it must first obtain ownership and invalidate the two remote copies. Similarly, if Proc3 attempted to write to the same page, it would need to obtain both ownership and a copy of the page, and then invalidate the three remote copies in processors 1, 2, and N.

1.2 Problem Statement

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 [LH89, BHJ+87, RAK88, BT88, FP89, CAL+89, MF89, MF90, BR90, SZ90, BZS93, AHJ91, NL91], DSM is not widely used. The reason 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.

The challenge in building a DSM system is to achieve good performance over a wide range of parallel 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. For DSM to become widely accepted, programs that execute on a DSM system must achieve performance comparable to equivalent programs written using explicit message passing. Programmers may accept some overhead in return for the much improved programming interface, but they will not accept a significant performance degradation. This argument is somewhat analogous to the debate over using assembly language versus a high level language. When compilers generated poor code, hand optimization using assembly code was common. However, as the quality of compilers dramatically improved, the need for hand-coded assembly code diminished. Similarly, if the performance of applications using DSM became sufficiently close to that of the same applications written using message passing, there would be little reason to use message passing.

The primary source of DSM overhead is the large amount of communication that is required to maintain consistency. Since DSMs use general-purpose networks and operating systems to communicate, the latency of each message between nodes is high. This latency varies on different distributed memory architectures, ranging roughly from 100 usecs on dedicated distributed memory hardware like the Intel iPSC-2 [Int90] to over 1000 usecs on a distributed system composed of workstations on an Ethernet [CZ83]. However, sending a message to another processor is always several orders of magnitude more expensive than performing a local memory access. 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 goal of this research is to make DSM a more viable solution for distributed processing by reducing the frequency and amount of communication performed by a DSM program to roughly that performed by an equivalent message passing program.

1.3 Contributions

The primary result of the research reported here is the demonstration that DSM is a viable option for distributed memory multiprocessing. We have developed several techniques that substantially reduce the amount of communication required to maintain consistency compared to previous DSM systems. These innovative communication-reducing features include:

- *software release consistency*, which allows a DSM system to mask network latency and reduce the amount of communication required to keep memory consistent through the use of a *delayed update queue*,
- the use of *multiple consistency protocols*, keeping each shared variable consistent with a protocol well suited to its expected or observed access pattern,
- 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, and
- the use of an *update timeout mechanism* to address a problem with write-update protocols whereby updates are sent to all replicas of a data item, even those replicas that are no longer being actively used.

These innovative features have been incorporated into the design of the Munin[1] DSM system [CBZ91]. 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. An examination of the latter three cases indicates that the addition of a function shipping capability would bring their performance to within 10% of the message passing performance. 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.

Based on our experience with Munin, we propose several strategies for improving the performance of future DSM systems. These include integrating the DSM system into a parallel programming environment with a sophisticated optimizing compiler tuned for DSM, and providing powerful performance debugging and visualization tools to guide programmers who wish to improve the performance of their programs.

1.4 Overview

The rest of this dissertation is organized as follows.

Chapter 2 details the four novel DSM implementation techniques presented in this thesis: the use of software release consistency, support for multiple consistency protocols, support for multiple concurrent writers, and the update timeout mechanism.

In Chapter 3, the design and implementation of the Munin prototype system is presented so that the basis for the experimental results is clear. This chapter begins with a discussion of the design philosophy, followed by a detailed description of the major components of the Munin system, particularly the runtime system. The consistency protocols and mechanism that supports software release consistency are presented in detail.

Chapter 4 describes the test suite of seven parallel programs used in the evaluation of Munin, and presents the results of a series of experiments that compare the performance of these programs using Munin DSM, conventional DSM, and explicit message passing. The results of these experiments both demonstrate the degree to

[1]According to Norse mythology, the god Odin had two ravens that sat on his shoulders, Hugin and Munin. Every evening, Hugin and Munin flew over all the lands of people who worshiped the Norse gods and surveyed the state of Mankind. Hugin collected all of Man's thoughts while Munin collected all of Man's memories, and then they reported all that they learned to Odin. Thus, if you consider all of mankind to be the ultimate distributed memory multiprocessor, then Munin was the original (and ultimate) distributed shared memory system.

which Munin has successfully solved many of the outstanding performance problems of DSM and indicate areas where further work is necessary.

Chapter 5 compares Munin to existing software and hardware DSM systems, with special emphasis on the difference in design strategies and the tradeoffs that result.

The major conclusions of this dissertation are presented in Chapter 6.

Appendix A contains a complete sample Munin program, `finite_diff`, to illustrate the Munin programming model.

Chapter 2

Mechanisms for Improving the Efficiency of DSM

If DSM systems are to approach the performance of their message passing counterparts, then their communication requirements cannot be substantially higher than those of message passing systems. We have developed several techniques that reduce the amount of communication needed for keeping the distributed memories consistent. These include:

1. *software release consistency*, an implementation of release consistency [GLL+90] specifically aimed at reducing the number of messages required to maintain consistency in a software DSM,

2. the use of *multiple consistency protocols*, which allows individual shared variables to be maintained using a consistency protocol well-suited to the way in which they are characteristically accessed,

3. a *multiple writer protocol*, which addresses the problem of false sharing in DSM by allowing multiple processes to write concurrently to a shared page, with the updates being merged at the appropriate synchronization point, and

4. an *update timeout* mechanism, which reduces the impact of updates to stale data, a common problem with update-based consistency protocols.

2.1 Software Release Consistency

Prior DSM systems employed the *sequential consistency* model [Lam79] as the basis for their consistency protocols. Sequential consistency requires that the distributed memories in a DSM have the same consistency properties as a time-shared uniprocessor, which requires that the global state of memory be consistent after every read or write to shared memory. This requirement imposes severe restrictions on possible performance optimizations. These restrictions have led to both theoretical [LS88] and empirical [BCZ90a, ZB92] arguments that DSM systems based on sequential consistency require a substantial amount of communication and are thus inefficient. Therefore, we chose to explore a more relaxed notion of consistency in DSM.

A number of consistency models have been developed for hardware shared memory multiprocessors that relax the restrictions of sequential consistency. We evaluated the following relaxed consistency models to determine which one would lend itself to the most efficient DSM implementation: weak consistency [DS90], PRAM [LS88], processor consistency [Goo91], and release consistency [GLL+90]. Release consistency (RC) imposes strictly fewer constraints on the DSM implementation than the other relaxed consistency models [BCZ90b, GGH91], and imposes few additional constraints on the programming model. These features of release consistency make it the best choice for DSM despite being somewhat more complicated to implement than the other models. Because DSM is implemented in software, the added performance optimizations allowed by release consistency to reduce the communication required to maintain consistency are far more significant than the small added overhead induced by the more complicated implementation. In the remainder of this section, we first compare sequential and release consistency and illustrate the reduction in communication overhead that can be achieved by basing a DSM system on release consistency rather than sequential consistency. We then contrast the DASH multiprocessor's implementation of release consistency [GLL+90], which uses pipelining to mask the latency of writes, with a scheme that buffers writes to reduce communication. Finally, we contrast implementations of release consistency based on update and invalidate protocols.

The following definitions are used by both the sequential and release consistency models[DSB86].

Definition 2.1 A `read` by processor p_i is **performed with respect to processor** p_j at a point in time when the issuing of a `write` to the same address by p_j cannot affect the value returned to p_i.

Definition 2.2 A `write` by processor p_i is **performed with respect to processor** p_j at a point in time when a `read` from the same address by p_j returns the value defined by the `write` (or some subsequent `write` to the same address).

Definition 2.3 An access is **performed** when it is performed with respect to all processors.

Definition 2.4 A `read` access is **globally performed** when it is performed and the `write` that is the source of the returned value has also been performed.

The *sequential consistency* model is an extension of the uniprocessor access model, in which each memory access must complete before the uniprocessor is able to generate the next access [Lam79]. Lamport defined sequential consistency as follows:

Definition 2.5 A system is **sequentially consistent** if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified in the program.

The most common implementation of sequential consistency requires the memory subsystem to obey two constraints [DS90]:

1. Before a `read` is allowed to perform with respect to any remote processor, all previous `reads` must be globally performed and all previous `write` accesses must be performed.

2. Before a `write` is allowed to perform with respect to any remote processor, all previous `reads` must be globally performed and all previous `write` accesses must be performed.

In addition to being straightforward to implement, sequential consistency provides the easiest framework for reasoning about the semantics of a parallel program. The programmer is ensured that no inconsistencies can arise because of unexpected buffering or access reordering. All accesses are atomic. Before the instruction following a write to a shared variable can begin executing, all copies of the shared variable must have been updated or invalidated. Similarly, a read from a shared variable is guaranteed to return the most recent write.

However, this convenient programming model does not come without a cost. Memory latency is a serious problem. Lipton and Sandberg [LS88] proved a bound on the best case performance of any implementation of sequential consistency, which they stated informally as: "*No matter how clever or complex a protocol is, if it implements [sequential consistency], it must be 'slow'. A [sequentially] consistent shared memory cannot be both fast and scalable.*"

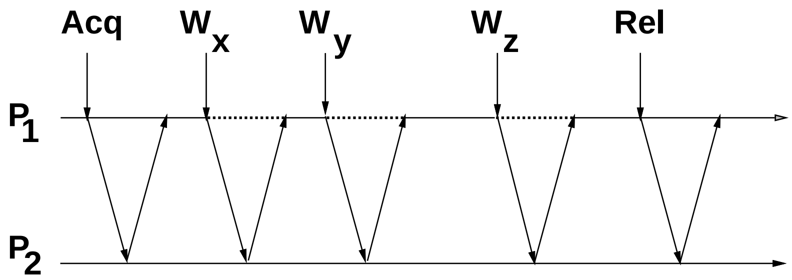

Figure 2.1 Implementation of Sequential Consistency

Figure 2.1 illustrates one reason that sequential consistency is inherently inefficient. In this example, two processors (P_1 and P_2) each have a cached copy of the variables X, Y, and Z, and P_1 modifies each of them within a critical section. On a sequentially consistent memory, each write must be delayed until the previous write completes, even within a critical section. Thus, despite the fact that P_2 cannot rely on the consistency of the shared data while P_1 is within the critical section, sequential consistency requires P_1 to communicate with P_2 whenever it modifies one of the shared variables, as represented by the arrows. The periods during which P_1 must stall while communicating are represented by the dashed portions of the execution of P_1. These unnecessary messages are the reason for sequential consistency's poor performance when applied to DSM, where the cost of communication is high.

In addition to leading to inefficient implementations, sequential consistency does not eliminate the need for programmers to think carefully about synchronization. Even on a time-shared uniprocessor, the use of preemptive scheduling can cause shared data to become inconsistent when insufficient synchronization is used and a thread is swapped out before it has completed a series of writes. Any program that requires threads to see a consistent view of shared data must perform sufficient synchronization to guarantee that a thread never reads data while another thread could be writing it.

The *release consistency* model was developed as part of the DASH multiprocessor project to address the problems of sequential consistency [GLL$^+$90, LLG$^+$90]. It exploits the fact that programmers use synchronization to avoid conflicting accesses to shared variables. Since accesses to shared variables that occur between synchronization points cannot conflict with other accesses to shared variables, the system need only guarantee that memory is consistent at select synchronization points. This ability to allow temporary, but harmless, inconsistencies is what gives release consistency its power. Intuitively, when a thread successfully acquires a lock, it gains access to a version of shared data that includes all modifications made before the lock was last released. After a thread performs a release, other threads that subsequently access the protected variables should access updated copies. If all writes to shared data are synchronized, as is most often the case, this guarantee can be satisfied by stalling releases until all previous writes complete. Doing so guarantees that after a thread exits a critical section, the next time any thread reads a shared variable that was modified in the critical section, it will see the new value (or a subsequent one)[2]. If a programmer uses sufficient synchronization, the program generates the same results on a release consistent memory system as it would on a sequentially consistent memory system. Experience with release consistent memories indicates that because of the need to handle arbitrary thread preemption, most shared memory parallel programs contain sufficient synchronization even when written assuming a sequentially consistent memory [CBZ91, GGH91].

The following constraints on the memory subsystem ensure release consistency:

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

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

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

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

[2]Release consistency also has a notion of *special* accesses, accesses to shared data that are *not* protected by locks and can thus involve *race conditions*. This type of access appears in parallel programs that can tolerate temporary inconsistencies, such as chaotic relaxation algorithms. This type of access is uncommon, so we ignore it throughout the rest of this discussion.

shared data consistent. In the case of pipelining, used extensively by the DASH multiprocessor [GLL+90, LLG+90], multiple writes can be outstanding without stalling the processor. In the case of buffering, which we found to be very useful in a software DSM system, multiple writes can be combined into a single message, reducing the number of messages and thus the communication overhead. Further, because a release consistent memory system recognizes the difference between synchronization accesses and normal read/write accesses, it can implement synchronization using separate mechanisms, making it more efficient.

The benefit of using relaxed consistency models increases dramatically as the memory latency increases [ZB92]. In addition, the performance benefit of relaxed consistency is most significant when there is a high degree of sharing and thus a large amount of consistency traffic [ZB92]. Thus, in a DSM system, with high memory latencies and potentially large amounts of sharing, the use of some form of relaxed consistency is of paramount importance. Of the existing options, release consistency provides the most opportunities for reducing the number of messages transmitted to maintain consistency, thus improving the performance of DSM, without imposing many new burdens on programmers.

Buffered versus Pipelined Release Consistency

The release consistency model was developed as part of the DASH project, and proved to be effective at hiding the effects of memory latency by *pipelining* invalidation messages caused by writes to shared data. However, the implementation of release consistency used by the DASH multiprocessor was not designed to minimize the number of messages sent, but rather to mask the latency of writes using pipelining. Because of the high overhead of sending messages in software, it is more important to reduce the frequency of communication in a DSM environment than it is to mask latency by pipelining messages. For this reason, we developed an implementation of release consistency that *buffers* writes instead of pipelining them, as illustrated in Figures 2.2 and 2.3. These figures illustrate how writes to three shared variables (x, y, and z) within a critical section are handled by the DASH implementation of release consistency and an implementation that uses buffering, respectively. When a processor writes to several different replicated cache lines within a critical section, the DASH scheme sends one message per write, while the new scheme buffers writes to shared data until the subsequent release, at which point it purges the buffered

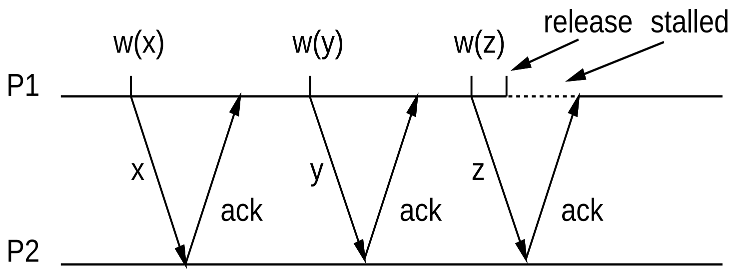

Figure 2.2 Pipelining Invalidations in DASH

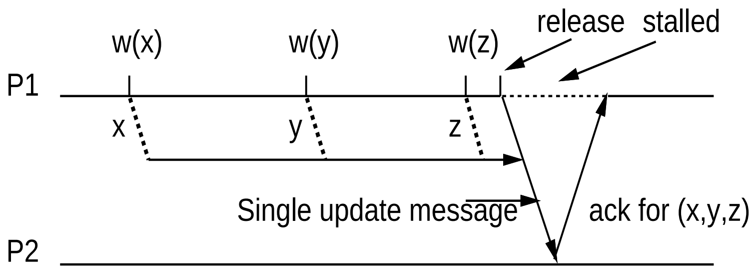

Figure 2.3 Buffering and Merging Updates

writes. Ideally, this strategy reduces the number of messages transmitted from one per write to one per critical section when there is a single replica of the shared data. The dashed line portion of the execution graph represents the delay that a processor experiences when releasing a lock. Because our scheme delays all writes until the release point, it must transmit all buffered writes then, increasing the latency of releases. Nevertheless, the reduction in the number of messages far outweighs the effect of the higher release latencies.

Update- versus Invalidate-based Release Consistency

Buffering and pipelining reduce the cost of writes, but have no effect on the cost of read misses. The impact of read misses can be partially mitigated by using an update protocol. Update protocols can reduce the frequency of read misses compared to invalidate protocols for data that is frequently read but infrequently written. This access pattern is common in shared memory parallel programs [WG89, BCZ90a]. For data accessed in this way, it is better to transmit updates than to transmit an equal number of invalidates, because the invalidates will be followed by a flurry of data requests as the remote caches reload the invalidated data. In addition to reducing the amount of communication, update protocols can also reduce the frequency with which a thread stalls waiting for invalidated data to be reloaded. Nevertheless, update protocols based on sequential consistency have fallen out of favor because of the large amount of communication required to send update messages for every write. However, an update protocol based on release consistency can buffer writes, which reduces substantially the amount of communication required. Previously, no relaxed memory model had been implemented using an update protocol, but our results show that it can improve the performance of DSM. Our *write-shared* protocol, an update-based implementation of release consistency that uses buffering to reduce the amount of required communication, is discussed in detail in Section 2.3.

2.2 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 [LH89] supported a page-based emulation of a conventional hardware protocol while Emerald [JLHB88] 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 [BCZ90a] and others [EK88a, EK88b, SA88, AG88, WG89, VF92, DL92] 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 [EK88a, EK88b, SA88, AG88, WG89, BCZ90a, VF92].

2. The number of characteristic sharing patterns is small and most shared data can be characterized as being accessed in one of these ways [WG89, BCZ90a].

3. Synchronization variables are accessed in an inherently different way than data variables, and are more sensitive to increased access latency [WG89, BCZ90b].

4. The conventional notion of an object does not correspond to the appropriate granularity of data decomposition for parallelism, because it is often the case that multiple threads concurrently access and modify independent portions of large objects such as arrays [BCZ90a, DL92].

5. The characteristic access pattern of individual variables does not change frequently during execution [BCZ90a, VF92], 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.

The addition of two special cases that further refine the characterization of how shared data is accessed results in a total of five consistency protocols: *conventional*, *read-only*, *migratory*, *write-shared*, and *synchronization*[3].

Conventional shared variables are replicated on demand and are kept consistent using an invalidation-based protocol that requires a writer to be the sole owner before it can modify the data. When a thread attempts to write to replicated data, a message is transmitted to invalidate all other copies of the data. The thread that generated the miss blocks until all invalidation messages are acknowledged. This single owner consistency protocol is typical of what existing DSM systems provide [LH89, FP89, DCM+90].

---

[3] The results of our original study indicated that there were eight basic access patterns (private, write-once, migratory, write-many, producer-consumer, result, read-mostly, and synchronization), but experience has made it clear that several of the protocols were redundant [BCZ90b]. Specifically, the *result* and *producer-consumer* access patterns were sub-cases of the *write-shared* access pattern.

Once *read-only* data has been initialized, no further updates occur. Thus, the consistency protocol simply consists of replication on demand. A runtime error is generated if a thread attempts to write to read-only data. Read-only data is provided as a special case of conventional data for debugging purposes.

For *migratory* data, a single thread performs multiple accesses to the data, including one or more writes, before another thread accesses the data [WG89, BCZ90b]. This access pattern is typical of shared data that is accessed only inside a critical section or via a work queue. The consistency protocol for migratory data propagates the data to the next thread that accesses the data, provides the thread with read *and* write access (even if the first access is a read), and invalidates the original copy. This protocol avoids a write miss, and a message to invalidate the old copy when the new thread first modifies the data. Furthermore, if it is known which lock protects the data, then the data can be migrated at the same time as ownership of the lock is transferred [RAK88]. This avoids a read miss when the data is first accessed, and allows data motion and synchronization to be merged into a single message exchange. Several researchers have found that migratory data is fairly common and that direct support for it can improve the performance of shared memory [CF93, SBS93].

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, access to write-shared data suffers from the effects of *false sharing* if the DSM system attempts to keep these independent portions of the data consistent at all times. A common example of false sharing occurs when two or more threads modify different portions of a shared array, some of which may reside in a single page, and then synchronize at a barrier. The write-shared protocol uses an update-based protocol that buffers modifications to handle this form of sharing. 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.

In our study of parallel programs, we identified three types of *synchronization* variables: locks, barriers, and condition variables. Because synchronization variables are accessed in a fundamentally different way than normal data objects, it is important that synchronization *not* be provided through shared memory, but rather via a suite of synchronization library routines or similarly specialized implementation. This reduces the number of messages required to implement synchronization, especially compared to conventional spinlock algorithms [And90], and thereby reduces the amount of time that threads spend blocked at synchronization points.

2.3 Write-Shared Protocols

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*. False sharing arises because the DSM system cannot distinguish individual words when protecting regions of memory, as the virtual memory hardware provides control only at the granularity of a page. Figures 2.4 and 2.5 illustrate two common examples of false sharing.

In Figure 2.4, two independent shared variables reside on the same page of memory, each being modified by a different processor. In Figure 2.5, 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.

False sharing causes performance problems in DSMs that maintain consistency at the granularity of entire pages or entire objects. Every time a thread modifies a page

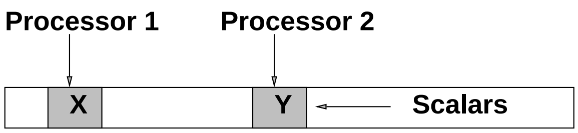

Figure 2.4 False Sharing: Scalars

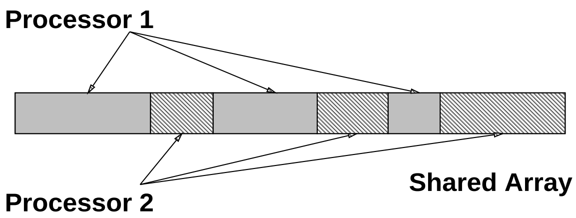

Figure 2.5 False Sharing: Arrays

of shared data or portion of a shared object, these systems must invalidate or update all replicas. Because each processor accesses an independent portion of the page or object, these operations and messages are logically unnecessary and should be avoided to the extent possible. 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. This observation leads to the conclusion that it is necessary to support a fine-grained access granularity, rather than operating on a strictly per-page or per-object granularity.

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 2.3. A single processor often performs a series of writes to a shared block within a critical section [BCZ90a]. 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.

The mechanism used to buffer and combine update messages, the *delayed update queue* (DUQ), is illustrated in Figures 2.6 and 2.7. 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. This causes the DSM runtime to make a clean copy of the page (X_{twin}), put it on the DUQ, and remap the page to be writable, as illustrated in Figure 2.6. If a copy of the page is already present on the processor, no communication is required. All subsequent writes incur no DSM overhead, until the processor next performs a release. At this time, the DSM runtime determines if any updates have been buffered on the DUQ. In the example illustrated in Figure 2.7, it determines that X has been modified. It then compares the modified data, X, with the original clean data, X_{twin},

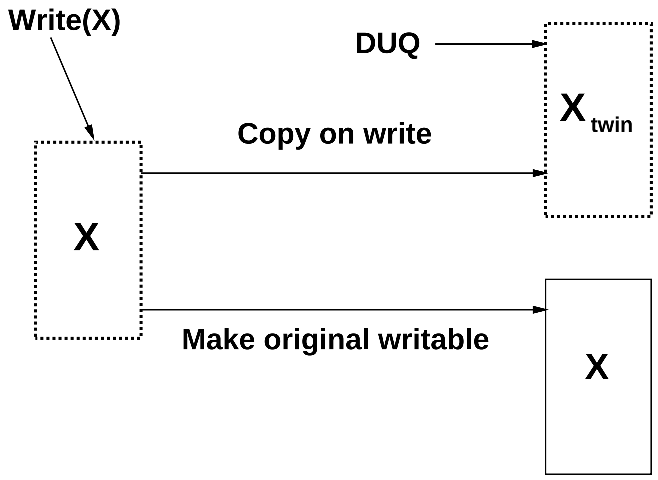

Figure 2.6 Creating a Twin

to determine the extent of the modifications, encodes the buffered modifications, and sends them to the replicas of X, if any. If, while updating the remote replicas of X, the DSM system determines that X is still replicated, X is write-protected anew to ensure that subsequent writes are detected and buffered.

Incorporating an update normally entails simply traversing the encoding and copying the 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. A *data race* occurs when multiple threads modify the same *word* of data without intervening synchronization, which usually represents a synchronization bug. 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.

The results of several trace studies indicate that the overhead of buffering updates and transmitting them at synchronization points is small compared to the amount of overhead eliminated by reducing the amount of communication required to maintain consistency. The average number of different objects accessed between synchronization points is small [BCZ90a], so the DSM system will need to enqueue changes

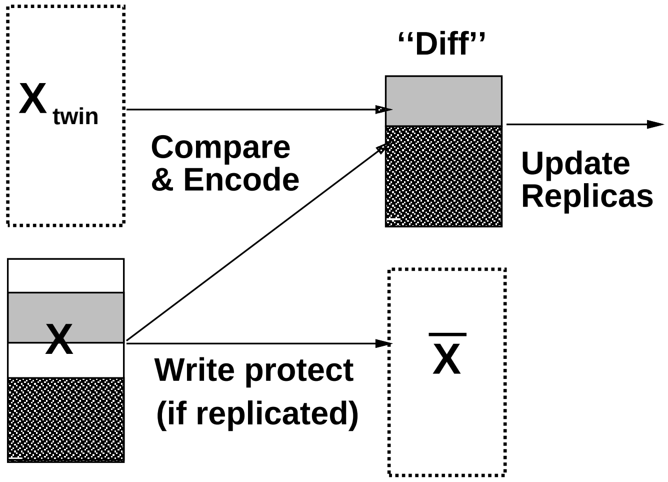

Figure 2.7 Flushing the Delayed Update Queue

infrequently. Furthermore, when a shared data object needs to be updated, it usually resides on a small number of other processors [SA88, AG88]. Thus, the number of update messages required to maintain consistency will be small.

2.4 Update Timeout Mechanism

A problem with conventional update-based cache protocols is that updates to a particular data item are propagated to all of its replicas, including those that are no longer being used. Without special provisions, updates to these *stale* replicas can lead to a large number of unnecessary consistency messages, resulting in poor performance. This effect is a major reason that existing commercial multiprocessors use invalidation-based protocols.

We address this problem with a timeout algorithm similar to the competitive snoopy caching algorithm devised by Karlin [KMRS86]. The goal of the update timeout mechanism is to invalidate replicas of a cached variable that have not been accessed recently upon receipt of an update. An example of the problem and our solution are illustrated in Figure 2.8.

In the example shown in Figure 2.8, the circles represent four processors in the system, and the arrows represent the source and destination of updates for a particular shared variable. In this example, exactly two processors access the variable at any given time. A dark shaded circle represents a processor that is actively using the

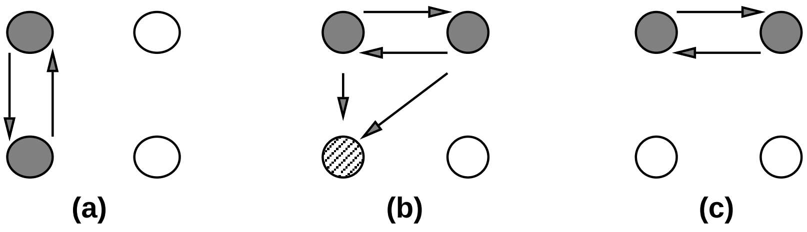

Figure 2.8 Update Timeout Mechanism

shared variable, while a cross-hatched circle represents a processor that is not using the shared variable but that is still receiving updates to it. In Figure 2.6(a), only the two processors using the variable are sending or receiving updates involving the variable, which is the desired situation. In Figure 2.6(b), only two processors are using the variable, but a third is receiving updates to it because it recently used the variable and still has a copy of it in its cache. Without the use of an update timeout mechanism, this processor will continue to receive updates to the variable until the program terminates or the variable is invalidated because of memory capacity limits, even if it never again uses the variable. If, over the entire execution of the program, every processor accessed the variable at some point, then eventually every processor would receive updates for the variable, even though at most two processors were using it at a time. The timeout mechanism eliminates these logically unnecessary updates by invalidating stale replicas. Figure 2.6(c) illustrates the situation after the timeout mechanism has invalidated the stale replica that had been cached in the lower left processor. Only the two processors using the variable are sending or receiving updates involving the variable, which is the desired situation. The implementation of the update timeout mechanism is described in detail in Section 3.5.3.

2.5 Summary

This chapter introduces and motivates four novel techniques for reducing the amount of communication required to maintain consistency in a software DSM. We illustrate the importance of exploiting the flexibility of *release consistency* to support a multiple writer consistency protocol using a *delayed update queue*. In addition, we demonstrate the potential benefit of buffering and merging consistency operations, which release consistency allows. *Support for multiple consistency protocols* is motivated by the observation that shared data is accessed in many ways. DSM's software implementation gives it the flexibility to support a suite of protocols so that each shared data can be maintained by a protocol that is well-suited to the way that it is characteristically accessed. We attack the problem of *false sharing* by introducing the *write-shared* protocol, which supports multiple concurrent writers to individual shared variables. Our write-shared protocol differs from DASH's hardware implementation of release consistency in several ways. It uses updates to avoid high latency read misses and it buffers writes to shared data rather than using invalidation to reduce the number of messages exchanged to keep falsely shared data consistent. Finally, we motivate the use of an *update timeout* mechanism to address a common problem of update protocols, that of the large number of messages sent to update stale data.

Chapter 3

Munin: Prototype Design and Implementation

The techniques described in Chapter 2 were implemented in a prototype DSM system called Munin. The Munin prototype was implemented on a network of sixteen SUN-3/60 workstations running a modified version of the V operating system [CZ83]. This chapter contains a detailed description of the design and implementation of the Munin prototype. It begins with a discussion of the philosophy that was used throughout the design process, its rationale, the major policies that resulted, and their implications. After a description of how Munin programs are written, the majority of the chapter describes in detail the major components of the Munin prototype implementation: the preprocessor and linker support, the operating system support, and the runtime library that implements Munin's consistency protocols and synchronization operations. Special care is given to the discussion of the runtime system, the core of the Munin prototype, and its major data structures, novel consistency protocols, and limitations. This description includes an evaluation of the options that were considered for the various data structures and consistency protocols, the rationale behind the chosen alternatives, the ramifications of these choices, and a discussion of options that might be preferred in a future implementation. Finally, we conclude with a summary of what we did, what we learned from it, and what we would do differently in the future.

3.1 Design Philosophy

The goal of the Munin project was to explore techniques for providing efficient support for executing shared memory programs on distributed memory multiprocessors. The reason for supporting a shared memory programming interface, as opposed to a conventional message passing system, was to ease the chore of programming distributed memory machines. This required Munin to provide a programming interface that did not require programmers to make significant modifications to their shared memory parallel programs. A secondary goal of the Munin project was to determine how both software and hardware DSM systems should be built on future large-scale multiprocessors with high bandwidth interconnects.

These goals formed the basis of the Munin design philosophy, and led to the following policies:

1. Provide a user interface as similar as possible to that provided by true shared memory multiprocessors.

2. Do not restrict the programmer's choice of programming language.

3. Do not require compiler optimizations to attain acceptable performance, but avoid decisions that preclude their use in future implementations.

4. Avoid algorithms and data structures that will restrict the ability of the system to scale to future large multiprocessors.

5. When possible, avoid using language, hardware, or operating system features that are not likely to be available on a wide variety of systems.

These policies had a major impact on the many aspects of Munin, including the programming interface, the compile-time tools, and the algorithms and data structures used by the runtime system. Some examples of the ramifications of these policies are: (i) the decision to provide a specialized library of synchronization operations to improve the performance of synchronization in Munin (policy 4), (ii) the decision to support the C programming language in its full generality, including pointers (policies 1, 2, 3, and 5), and (iii) the use of distributed ownership protocols implemented without hardware multicast (policies 4 and 5). These policies enabled us to fairly evaluate the value of Munin's novel techniques for improving the performance of DSM, but they limited the number of optimizations that we could perform. For example, while the prototype system was built on top of a network of workstations connected via an Ethernet, we did *not* use the hardware multicast feature of Ethernet, because it is not clear that hardware multicast will be present in future large-scale multiprocessor systems. Similarly, although we had access to the sources of the compiler and operating system, and thus could have added special-purpose functionality to improve performance (e.g., Munin-specific in-kernel fault handlers), we chose not to do so. We limited the extent to which we modified existing software and programming interfaces or relied on language, OS, and hardware features that are not available on many systems. Doing so allows us to extrapolate our results to widely available distributed systems (e.g., networks of workstations running conventional operating systems) and guide the design of future systems. In addition, by evaluating how these policies impacted the performance of the Munin prototype, we can determine which policies should be reconsidered in the design of future DSM systems.

3.2 Writing A Munin Program

Munin programmers write parallel programs using threads, as they would on a uniprocessor or shared memory multiprocessor. Munin provides special library routines to create and delete threads for this purpose. Appendix A contains a sample Munin program, finite differencing, for illustrative purposes. As can be seen in this example program, the format of a Munin program is very similar to that used by common parallel programming environments like Presto [BLL88] and C-threads [CD88].

User initialization is performed by a sequential `user_init()` routine. Similarly, a sequential `user_done()` routine is run when the computation has finished. Both `user_init()` and `user_done()` are optional. Munin currently does not perform thread migration or global scheduling. User threads are run in a round robin fashion on the node on which they were created.

If the programmer does not provide a `user_init()` routine, a NULL routine is invoked by default. All single-threaded initialization should be performed in the `user_init()` routine. Some examples of what should be done in `user_init()` are: the parsing of command line arguments, the initialization of shared variables (read-only variables in particular), the creation of global synchronization variables (locks, barriers, and monitors), and the specification of the number of processors that Munin should use. All writes to read-only variables *must* occur in `user_init()`. Any changes to global variables that are performed within `user_init()` are propagated to the other Munin nodes as they are created.

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 what protocol to use to keep it consistent, e.g., "`shared {protocol} <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. Normally Munin maintains consistency on a per-variable granularity, but like Emerald [BHJL86, BHJ+87, JLHB88], the programmer can specify that a collection of variables be treated as a single entity. By default, variables larger than a virtual memory page are broken into multiple page-sized pieces and each variable is placed on a separate page.

Synchronization is supported by library routines for the manipulation of locks, barriers, and condition variables. All synchronization operations must be explicitly visible to the runtime system, which means that programmers must use the library routines provided by Munin. This restriction is necessary for release consistency to operate correctly. The routines provided include `AcquireLock()`, `ReleaseLock()`, `WaitAtBarrier()`, `WaitCondition()`, and `SignalCondition()`.

3.3 Compiling A Munin Program

The default C compiler has been modified to support the creation of Munin executables. After writing a Munin program in the style described in Section 3.2, the programmer compiles the Munin program using the normal compilation commands, with the addition of a special flag (`-m`) that indicates to the compiler and linker that it is a Munin program. The compiler front end has been modified to invoke a simple preprocessor on Munin source files that reads the shared variable declarations, uses them to create an auxiliary file, and then strips the Munin annotations from the source file before piping it to the standard compiler. The Munin flag tells the linker to use the auxiliary file when converting intermediate files into an executable and to link in the Munin runtime library.

While the current prototype only supports C programs, the compiler modifications were limited to the creation of a preprocessor that makes source-to-source transformations and creates a Munin-specific (but language-independent) auxiliary file. Thus, it would be easy to port the current version of Munin to other languages.

Preprocessor

The Munin preprocessor, `mpp`, filters the source code in search of annotated shared variable declarations. When it finds such a declaration, it removes the Munin-specific "`shared {protocol}`" annotation and adds an entry to the `.munin` auxiliary file. The `.munin` file consists of a count of the number of shared variables declared followed by a one line description of the shared variables, a simple example of which appears in

Figure 3.1. The order of the entries in the auxiliary file is significant, as the linker allocates space for variables in the shared data segment in the order that they appear in the `.munin` file. The "+" is a hint to the linker that it should place that particular variable on a separate page of memory from the variable declared before it.

Placing each variable on a separate page could waste memory if there are a lot of small variables, but in practice the number of statically-declared shared variables is likely to be small, because it is tedious to declare a large number of independent scalar variables. Arrays are laid out contiguously in memory. Users can edit the `.munin` file if they wish to hand tune the layout of the shared data segment. The example `.munin` file in Figure 3.1 specifies that `Array` is `write-shared`, that the task stack variables are `migratory`, and that the task stack variables should be on the same page as one another but different from `Array`. If there are multiple variables on a page, Munin treats them as a single entity, and uses the consistency protocol given for the first variable on the page.

Linker

When compiling a Munin program, the modified linker reads the auxiliary `.munin` file, relocates the variables specified in the auxiliary file to a special shared memory segment, and appends two new segments to the end of the Munin executable. The two new segments are a shared symbol table that describes the layout of the shared address space and the annotations of the shared variables, and a shared data segment that contains the data needed to initialize non-bss shared data. Shared data is separated from non-shared data so that access to non-shared data does not cause any consistency activity via false sharing. Finally, the linker marks the executable as a Munin program by placing a special magic number in the header. Using a special magic number allows the operating system to recognize the executable as a Munin program when it is invoked and handle it appropriately. It also guarantees that a Munin executable

```
3
Array           write-shared
+TaskStack      migratory
TaskStackTop    migratory
```

Figure 3.1 Example `.munin` File

cannot be started on a machine that is running a version of the operating system that does not support Munin.

The modifications to the linker were fairly complicated, as they involved changing a complex and poorly documented program. Alternatively, no modifications to the linker are necessary if all shared data is allocated via a special library routine like `shmalloc()`, although doing so complicates the programming interface by disallowing shared static variables.

3.4 Operating System Support

Two parts of the V operating system were modified to support Munin: the kernel and the process server[4] (the server that supports the creation and deletion of processes).

The standard V kernel was modified only slightly to support Munin. Less than 200 lines of code were added, of which approximately 10% were in assembler. This added kernel support came in two forms: (i) virtual memory manipulation routines and (ii) support for user-level fault handling. In addition, Munin makes extensive use of V's IPC mechanisms (`Send-Receive-Reply` and `MoveTo-MoveFrom`) and uniprocessor synchronization mechanisms (`P` and `V`).

To support Munin's need to manipulate the virtual memory hardware and modify page table entries, we added three memory-management routines:

- `MapSegment(start, len, protection)` allows Munin to map pages of virtual memory at arbitrary locations in the address space. It allocates the desired number of physical pages, if possible, and maps the specified page table entries to those physical pages with the appropriate protection.

- `UnmapSegment(start, len)` invalidates the specified page table entries, and frees the physical pages mapped in the given range, if any.

- `SetProtection(start, len, protection)` changes the page table entries of already allocated memory for the given range of addresses to the specified level of protection.

[4]In V, the set of resources associated with a single address space is known as a *team*, and the server that manages these teams is known as the *team server*. However, we will use the terms "process" and "process server" throughout this discussion, to avoid confusion.

These routines were new to the V kernel, but provide functionality similar to the extended virtual memory management support in Unix [LMKQ89] and Mach [YTR+87].

The Munin runtime on each node can register itself with the local kernel by calling `RegisterMuninHandler()`, which informs the kernel that all virtual memory exceptions by a thread in the Munin program should be forwarded via an exception message to a Munin thread for handling at user-level. This process is analogous to registering a signal handler to catch SIGSEGV and SIGBUS under Unix. When a user thread generates a page fault, the kernel saves its state, builds a message describing the exact nature of the fault (e.g., the PC of the instruction generating the fault, the fault address, and whether it was a read or a write operation), and sends this message to the registered Munin thread as if the faulting thread had sent it. When the Munin thread is done handling the exception, it replies to the message as it would to any other message, which has the effect of resuming the faulted thread and causing it to re-execute the instruction that caused the fault.

Only minor modifications were made to the process server. During startup, the process server queries the local kernel to determine if it supports the operations described above. If so, it allows programs with the Munin magic number to be executed, otherwise it returns an error condition. Immediately signalling an error when a Munin program attempts to run on a workstation that does support the required functionality simplifies error recovery. In addition, a new system call, `CreateRemoteProcess()`, was added to support a form of distributed `fork()` operation. `CreateRemoteProcess()` creates an empty address space on a remote processor and lets the invoking process initialize the address space, rather than always loading it from a file as is done by the default process server. `CreateRemoteProcess()` allows the Munin root node to spawn replicas of itself, complete with initialized data.

To summarize, a user-level implementation of Munin requires the following functionality from the operating system:

- the ability to manipulate arbitrary virtual memory mappings by a user process,
- the ability to handle page faults (both segmentation faults and protection violations) at user level, and resume threads after they fault,
- the ability to create and destroy processes on remote nodes,
- the ability to exchange messages with remote nodes (perform IPC), and

- uniprocessor synchronization support (`P()` and `V()`).

An operating system that provides this functionality should be able to support Munin. All of these requirements are present in current operating systems. Mach [ABB+86] supports an external pager [YTR+87] that provides exactly the functionality required. Chorus [ARS89] provides similar capabilities. Even modern Unix systems support these requirements through the user of signal handlers and the BSD-derived memory mapping routines [LMKQ89] (e.g., `mprotect()`).

3.5 Munin Runtime

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.

Figure 3.2 illustrates the organization of a Munin program during runtime. 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. 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. The details of this entire process are presented in the following sections.

Linking the Munin runtime code and data into the same address space as the user program has several advantages and disadvantages. User threads can perform procedure calls to Munin routines, such as `AcquireLock()`, and user routines, such as `user_init()`, can be directly invoked by Munin threads. Munin threads can directly access user data, and the user program can tune the runtime system's behavior by modifying global variables, although extensive double-mapping of pages between address spaces could accomplish the same effect. Also, placing the runtime system in

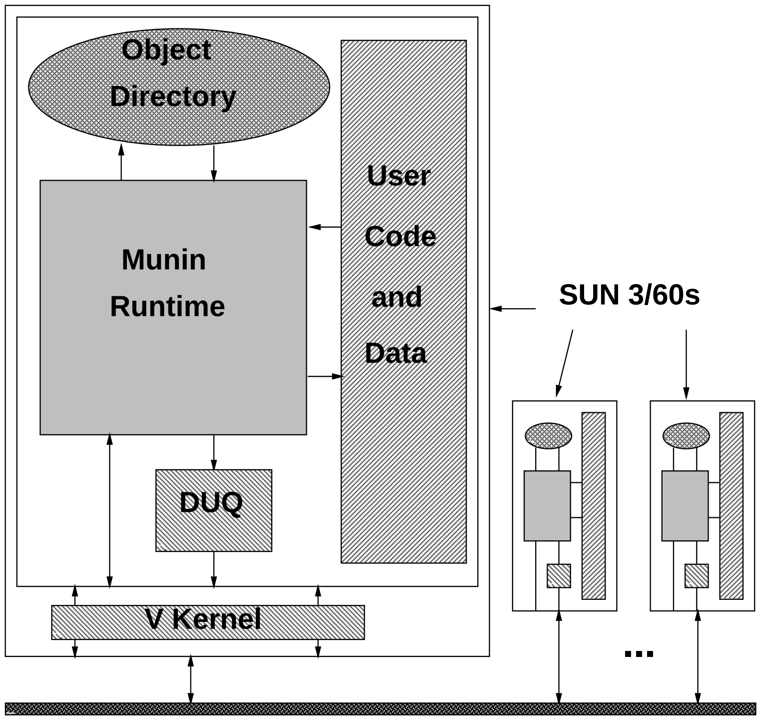

Figure 3.2 Munin Runtime Organization

the same address space as the user program reduces the overhead of context switching between the user program and the runtime system. The primary disadvantage of placing the runtime code into the same address space as the user program is a bug in the user program can overwrite Munin data structures. However, since each instance of an application executes an independent instance of the runtime system, the most serious problem that allowing a user bug to overwrite system data can cause is the bug may become more difficult to locate.

The Munin runtime threads consist of one *root* thread and a number of *worker* threads. Both the root and worker threads loop continuously, accepting and handling requests. The root thread handles all operations that cannot block, such as a request by a remote node for a lock, and forwards all others to an arbitrary worker thread. Similarly, the kernel sends exception messages to an arbitrary worker thread. The specifics of the various requests are detailed in the sections on fault handling and synchronization (Sections 3.5.3 and 3.5.4).

3.5.1 Executing a Munin Program

When a Munin program is started, the operating system creates an empty address space large enough to contain the program, loads the text and data segments from the executable, clears the bss segment, and creates and starts the Munin root thread. The root thread reads the shared variable symbol table located in the executable, initializes the object directory accordingly, initializes the shared data segment, and clears the shared bss segment. After initializing itself, the Munin root thread executes the routine `user_init()`. After exiting `user_init()`, it creates the number of nodes specified by `num_hosts` using the machines specified by `MachineName[]`, if provided. Several auxiliary Munin worker threads are created on each node to assist in performing the consistency and synchronization protocols. Each node is initialized to contain an exact duplicate of the non-shared data and bss segments, while the shared data and bss segments are not mapped whatsoever. A side-effect of this is that all variables that are not specified as shared have their initial values duplicated everywhere. Thus, read-only variables can simply be marked as non-shared if the programmer does not care about the detection of illegal writes.

After all remote nodes have been created and initialized, each Munin node registers itself with its local kernel so that virtual memory faults will be handled appropriately. The Munin root thread on the root node then creates a single user thread executing the `main()` routine. Within `main()`, the user can create worker threads to perform the parallel computation using Munin's threads package.

3.5.2 Object Directory

On each node, the Munin runtime system maintains an *object directory* containing information on the state of each shared variable in the global shared memory. Each node's directory maintains that node's view of the state of each variable.

Munin uses two optimizations to reduce the number of messages required to maintain the distributed object directory. First, in keeping with the goal of avoiding centralized algorithms that can lead to performance bottlenecks, Munin distributes the state information associated with write-shared data across the nodes that contain cached copies of the data. Specifically, there is no "owner" for each write-shared variable that must be involved with every consistency operation that affects a shared variable. This approach also allows Munin to exploit locality of reference. This design differs from directory-based caching schemes proposed for scalable shared memory multiprocessors [ASHH88, CF78, CKA91], all of which maintain a single directory entry for each piece of data. Scalable shared memory multiprocessors use a fixed owner directory protocol because it is difficult to support complex protocols and variable-sized directories in hardware. Second, for the protocols that do have the notion of an "owner" (conventional and migratory), Munin implements a dynamic ownership protocol to distribute the task of ownership across the nodes that use a particular piece of data. Proposed scalable shared memory multiprocessors statically assign owners to data.

These techniques improve the performance of a DSM system. Consider a write-shared data item α that is being cached by nodes X and Y. In a conventional directory scheme, before either node X or Y can satisfy a third node Z's request for a copy of α, $owner_\alpha$ must be informed that Z is now caching α. Under Munin's distributed directory scheme, X or Y can immediately respond to Z's request, without communicating with any other node (e.g., $owner_\alpha$). In this example, if X receives a read request from Z for write-shared data α, it sends a copy of α directly to Z, and updates its own directory entry, without informing Y or any other node. The update protocol must guarantee that updates are received by all nodes with a cached copy of a shared data item, and not just those that the updating node originally knew about. When a node receives an update, it includes in the acknowledgement the set of nodes that it believes have a copy of the shared data item. In the example above, when Y sends an update to X, the only node that Y knows has a copy of α, X informs Y that Z also needs to be updated. Including the copyset in every acknowledgement guarantees that updates are propagated to all nodes caching a copy of the data, without requiring unnecessary messages either during normal operation or while updating. Depending on how updates are propagated, this mechanism can, however, increase the time required to perform an update.

In addition, the fact that ownership of conventional and migratory data migrates dynamically whenever the previous owner stops caching the shared data can also improve performance. Only nodes that are caching the data are affected by operations on that data. Once again, consider a shared variable α that is being cached by nodes X and Y, neither of which is $owner_\alpha$. In a conventional hardware directory scheme, whenever node X needs to invalidate the copy of α on node Y, it must first communicate with node $owner_\alpha$ to determine what other nodes are caching α. Since $owner_\alpha$ is not caching α, communication involving it are strictly overhead. The owner of a shared variable in Munin always has a copy of the data, so no extra messages are required to maintain copysets or to satisfy read requests. Another advantage of Munin's distributed ownership protocol is that it eliminates the bottleneck that can arise when the statically assigned owners of the most frequently modified data are not evenly distributed across the nodes in the system. Experience with the Ivy system demonstrated that these dynamic, decentralized protocols improve DSM performance[Li86].

One potential disadvantage of Munin's object directory implementation is that it increases the amount of memory used to maintain directory information, an important issue for hardware implementations. The number of directory entries was relatively small for the programs that we studied, both because of the relatively small number of shared data items and their large granularity (8-kilobyte pages). If the number of directory entries became a problem, a per-node cache of directory entries could be used, and directory entry information could be flushed to "home" nodes or disk when an entry needed to be reused. In addition to increased memory usage, Munin's optimizations make the directory implementation somewhat more complicated, but added complexity is less of an issue for software systems than for hardware systems.

Munin's object directory is structured as a hash table that maps a virtual address to an entry that describes the data located at that address. The object directory on the Munin root node is initialized from the shared data symbol table located in the executable. The object directories on the other nodes are loaded as they are created, and then re-initialized to indicate that no shared data is present. When a Munin runtime thread cannot find an object directory entry in the local hash table, it requests a copy from the root node.

The fields of an entry in the directory include:

- **Lock**: provides exclusive access to the entry.
- **Start address** and **Size**: act as keys for looking up a shared data item's directory entry, given an address within the data.
- **Protocol**: specifies the protocol that the fault handler should use when servicing access faults on the data item.
- **State bits**: characterize the dynamic state of the data, such as whether it is present locally and whether it is writable.
- **Copyset**: a "best guess" of the set of processors with copies of the data.

- **Probable owner**: a "best guess" of the owner of the data, which has different interpretations depending on the protocol used to keep the data consistent.

- **Performance data**: runtime statistics used for performance tuning.

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. It is this property that ensures the correctness of the distributed ownership version of the update mechanism.

The *probable owner* is used to determine the identity of the Munin node that currently owns the data [Li86]. 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 [WH88].

Uniprocessor locks provided by the V system are used to control access to the individual directory entries, because the DSM runtime is multithreaded. Without locking the directory entries, two or more runtime threads could modify an entry simultaneously, resulting in an inconsistent entry. Deadlocks are avoided through careful implementation of the runtime protocols.

3.5.3 Fault Handling

Munin supports four consistency protocols: *write-shared*, *conventional*, *migratory*, and *read-only*. In addition to the four 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.

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 some or all accesses to it generates exceptions. These exceptions invoke the kernel exception handler, which creates an exception message describing the fault and sends it to one of the local Munin runtime threads, hereafter referred to as the *fault handler*. This fault handler examines the exception message to determine the location and nature of the exception. The fault handler uses the fault address to look up the directory entry corresponding to the shared data at that location. If it cannot find one, such as occurs when a bug in the program causes a NULL pointer reference, it prints an error message and invokes the standard debugger. Otherwise, the directory entry identifies the protocol being used to keep the data consistent. The protocol may require communication with other nodes to perform protocol operations or request data. The Munin runtime thread on a remote node that performs this operation or satisfies this request for data is referred to as the *server* throughout the rest of this discussion. 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.

Several situations other than faults can result in a fault handler being invoked. The most common occurs when a thread performs a release operation that requires the delayed update queue to be flushed.

In the discussion of the consistency protocols and figures that follow, we use *italics* to refer to items in the variable's directory entry and `typewriter type` to refer to local variables and function names.

Write Shared

The write-shared consistency protocol, presented in Figures 3.3 through Figure 3.7 and illustrated in Figures 2.6 and 2.7, operates as follows. The initial copyset of a write-shared data item is empty on all nodes, including on the root node. The root node has the *present* bit set, but the data itself is mapped as *supervisor-only*[5] so that it is not immediately accessible by the root user thread. On the other nodes, the *present* bit is reset and the data is not mapped at all. When a node first attempts to access write-shared data, a fault occurs either because it is not present (on nodes other than the root node) or it has been made inaccessible (on the root node). In the latter case, the root node simply remaps the data to be read-write, adds itself to the empty copyset, and resumes the thread. This seemingly unnecessary fault-remap-resume sequence is used to determine the working set of the root node, so that it will not retain a copy of data that it is not using. If another node requests a copy of data that the root node has not yet accessed, the root node responds with a copy of the data and invalidates its own copy. By doing so, the other node is able to immediately map the data as read-write, because it has the only copy. Furthermore, the root node does not receive updates for data that it never uses.

Figure 3.3 presents the read fault handler for write-shared data. As described above, if the data is present but has been read protected, such as the first time it is accessed on the root node or 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. It also marks the data as accessed as part of the update timeout mechanism. When a particular 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.

Figure 3.4 shows the write fault handler, which is similar to the read fault handler except when the data is shared. In this case, the write-shared protocol invokes the DUQ mechanism. 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, as illustrated in Figure 2.6. 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

[5]Directly accessible only by the kernel.

P(*lock*)
if ($present = 0$) Request data and *copyset* from *prob_owner*
else Change protection to READ_ONLY
$copyset \leftarrow copyset + \{$`this_node`$\}$
if ($copyset = \{$`this_node`$\}$) **begin**
    /* Special: we are the only one's using it. */
    Change protection to READ_WRITE
    $prob\_owner \leftarrow$ `this_node`
end
$not\_used \leftarrow 0$
V(*lock*)

Figure 3.3 Read Fault Handler for Write Shared Data

P(*lock*)
if ($present = 0$) Request data and *copyset* from *prob_owner*
if ($copyset = \{$ `this_node` $\}$)
    /* Special: we are the only one's using it. */
    $prob\_owner =$ `this_node`
else begin
    Make a twin
    $twin\_present \leftarrow 1$
end
Change protection to READ_WRITE
$not\_used \leftarrow 0$
V(*lock*)

Figure 3.4 Write Fault Handler for Write Shared Data

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. This feature is unique to Munin's write-shared protocol.

Figure 3.5 shows the server routine that handles remote requests for write-shared data. 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 data is remapped to be read-only so subsequent changes to the data are detected and eventually purged.

```
P( lock )
if ( present = 0 ) Forward request to prob_owner
else begin
      copyset ← copyset + {requesting_node}
      if ( writable = 1 ∧ twin_present = 0 )
            Change protection to read-only
      Send data and copyset to requesting_node
      if ( copyset = {requesting_node} ) begin
            /* Special case: we (root) have not used it, so throw it away. */
            prob_owner = requesting_node
            Invalidate data
      end
end
V( lock )
```

Figure 3.5 Data Request Server for Write Shared Data

When a local thread performs a release operation (releases a lock, arrives at a barrier, signals a condition variable, or terminates), 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. The algorithm used to flush the DUQ is illustrated in Figure 2.7, and formalized in Figure 3.6. To start each phase, the server walks down the DUQ and creates a differential encoding of every enqueued data item. It encodes the modifications by performing a word-by-word[6] comparison of the data item (X) and its original contents (X_{twin}). As it finds differences, it copies them to 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. The encoding algorithm has the property that it is never larger than the twin, and it never overwrites parts of the twin that have not yet been examined as part of the differential encoding process. Thus, reuse of the twin buffer is safe. As part of updating each enqueued data item, Munin maintains a global copyset of all

[6]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 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.

```
P( PurgingQueues )
NumEncoded ← 0
PurgingTo ← LocalNode
PurgedTo ← LocalNode

do  /* Encode all the objects on the DUQ */
    object ← Top of DUQ
    Encoded[NumEncoded].addr ← object.addr
    Encoded[NumEncoded].copyset ← object.copyset
    Encoded[NumEncoded++].nacked ← 0
    Add object.copyset to PurgingTo

    P( object's lock )
    Encode object
    Map object read-only
    Reset object's twin_present bit
    Remove object from DUQ
    V( object's lock )
while (DUQ not empty)

while ( PurgingTo ≠ PurgedTo ) do
    Send Encoded[*] to {node :  node ∈ PurgingTo - PurgedTo}
        (Returns return_copyset[1...NumEncoded]
            and NACKed[1...NumEncoded])
    for i = 1...NumEncoded begin
        Add returned_copyset[i] to Encoded[i].copyset
        Add returned_copyset[i] to PurgingTo
        if ( NACKed[i] ) Add i to Encoded[i].nacked
    end
    Add node to PurgedTo
enddo

for i = 1...NumEncoded begin
    P( Encoded[i]'s lock )
    object.copyset ← ( Encoded[i].copyset ⋃ object.copyset ) -
                         Nacked[i]
    if ( object.copyset = LocalNode )
        Map object read-write
    V( Encoded[i]'s lock )
end
V( PurgingQueues )
```

Figure 3.6 Algorithm for Flushing the DUQ nodes that need to receive updates of any of the data being encoded (`PurgingTo`). It also builds a data structure that describes the variables that have been encoded (`Encoded[]`).

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 represented by `PurgingTo`. Each of these recipients determines if it needs to receive updates to any or all 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 (ACK/NACK) 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, the data is remapped to be read-only if there are still other copies in the system so that the process may begin anew.

This algorithm was developed based on our experience with the Munin prototype. Originally, the update phase was performed serially. The updating node encoded a single data item and sent the encoded changes, and not just an indication of the data that has changed, to each of the nodes in the item's copyset. This algorithm worked well when every node receiving the encoding needed it. However, for those cases where the receiving node no longer needed to receive updates for the data, the potentially large encoding message was a waste of time and bandwidth. In addition, the purely serial algorithm caused the updating node to spend a lot of time waiting for each response. To alleviate this problem, we modified the original algorithm to search the DUQ for data items with identical copysets, and encode these items into a single large buffer. While this modified algorithm allowed multiple updates to proceed in parallel, it was often the case that the copysets of a collection of data items were each slightly different, in which case they again were sent serially. The obvious alternative, encoding all enqueued data items into a single large buffer and sending this buffer to each node that needed any of the encoded data, did not work well either. This alternative increased the potential parallelism, but nodes were often forced to search through large encodings to find the small portions that related to data that they were caching. In addition, the aggregate size of all the enqueued encodings often exceeded the size of even fairly large (64-kilobyte) encoding buffers, which resulted in the procedure again taking multiple passes to complete.

The algorithm currently being used is a good compromise between sending too much information and sending too little information speculatively. Upon receipt of a description of all the updates available at the updating node, the receiving node is immediately able to determine the complete set of data items for which it requires an update. Furthermore, by not sending the encodings themselves, the size of the first update message containing the list of all modified data is small, and generally fits into a single Ethernet packet. In addition, the amount of data shipped is minimized because a node only requests the encodings that it requires. However, requiring the recipient to request the data increases the lower bound of the number of message exchanges required to perform an update[7]. Nevertheless, this method worked well in practice.

Although it is not shown in Figure 3.6, the updates are performed by using multiple Munin runtime threads to send point-to-point messages in parallel. The statement "**while** (`PurgingTo` $\neq$ `PurgedTo`)" is run in parallel with sufficient synchronization to ensure that the global data structures are kept consistent. Before the updates were parallelized, our decision to not exploit Ethernet's multicast capability sometimes caused noticeable performance problems because the updating node was idle while each recipient node incorporated the updates. The addition of a software multicast mechanism using lightweight threads helped alleviate this problem, although it pointed out that hardware multicast or efficient OS-supplied software multicast should be exploited when available. The problem with sending the updates in parallel is that it sometimes saturates the network and messages are forced to "backoff" repeatedly. However, this effect was insignificant compared to the problem of the updating thread spending most of its time idle when the updates were performed serially.

Figure 3.7 presents the algorithm used 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. If it still has a copy of any of

[7]The runtime is sufficiently sophisticated to determine when the aggregate *encoding* is smaller than a single Ethernet packet, in which case it is appended to the descriptor. Thus, when only a small amount of data needs to be shipped, the update mechanism requires only a single message, the optimal result.

the data described in the update message, it reads 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 requested, the three way comparison discussed in Section 2.3 is performed to detect data races. 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.

This process is complicated somewhat by the timeout mechanism. The basic strategy of the update timeout mechanism is to only accept updates to a data item for a limited period of time after it was last accessed, and to time out and invalidate data that has grown stale. 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

```
P( lock )
nack ← 0
if ( present = 0 ) nack ← 1
else if ( not_used ∧ twin_present = 0 )
     cur_time ← ReadTime()
     if ( timestamp − cur_time > FREEZE_TIME )
          if ( prob_owner = this_node ) FindNewOwner( object )
          nack ← 1
     end
end

if ( nack )
     Invalidate object
     Send NACK and copyset − {this_node}
else
     Incorporate update into object
     if ( twin_present = 0 ) Change protection to SUPER_ONLY
     not_used ← 1
     timestamp ← ReadTime()
     Send ACK and copyset
end
V( lock )
```

Figure 3.7 Update Server for Write Shared Data directory entry at this time. If the data is accessed before another update arrives, the subsequent fault simply remaps the data and resets the `not_used` flag. However, if an update to the data is received and the node has not used the data since the last update ($not\_used = 1$), sufficient time has elapsed since the last update ($timestamp -$ `cur_time` $>$ FREEZE_TIME), and the data is not dirty ($twin\_present = 1$, meaning unpropagated modifications are present), the update server invalidates the local copy. An alternative mechanism that we did not investigate is to use the *reference* bits in the page table to determine whether a page has been accessed since an update was last incorporated. This mechanism requires the operating system to provide a means to read and clear reference bits when updates arrive and maintain the state of reference bits when data is paged to disk and later retrieved by the virtual memory system. Since few current systems provide this capability, we chose not to pursue this alternative.

The timeout mechanism currently does not invalidate dirty data under the assumption that it is likely to be flushed soon. In practice, the assumption works well. If this assumption turns out to be invalid for some applications, even dirty data could be tested for staleness, although flushing dirty data requires its delayed updates to be propagated before it is invalidated.

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. Before a node can timeout the copy of last resort, it must first find another node to hold the data.

The write-shared protocol works well when there are multiple writes to a data item between flushes, which allows the expense of the copy and subsequent comparison to be amortized over a large number of write accesses. Table 3.1 breaks down the time to handle updates to an 8-kilobyte data item through the DUQ. The totals include the time to handle a fault (including resuming the thread), make a copy of the data, encode changes to the data, 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 data item has changed. In the second, every word in the data 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 runs.

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

Table 3.1 Time to Handle an 8-kilobyte Data Item through DUQ (msecs)

A technique similar to the delayed update queue was used by the Myrias SPS multiprocessor [Myr90]. 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 and rejected two other approaches for implementing release consistency in software: (i) force the thread to page fault on every write to a replicated data item so that the modified words can be queued as they are accessed, and (ii) have the compiler add code to log writes to replicated data as part of the write. The first approach works well if the data 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 common for a single data item to be updated multiple times between DUQ flushes [BCZ90a], the added overhead of handling multiple page faults makes this approach unacceptable. The second approach is used by Emerald [BHJL86] and Midway [BZS93]. 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. It is an attractive alternative for systems that do not support fast page fault handling, such as the iPSC-i860 hypercube [Int90]. However, if the number of writes to a particular data item between DUQ flushes is high, as is often the case [BCZ90a], this approach will perform relatively poorly because each write to a shared variable is slowed down.

Conventional

Munin's *conventional* protocol is based on Ivy's distributed dynamic manager protocol [LH89], with the addition of Mirage's notion of temporarily freezing pages after they are accessed, so that in the presence of write-sharing the page does not ping pong between nodes continuously [FP89].

Munin's conventional consistency protocol is presented in Figures 3.8 through 3.12. Like Ivy, in the absence of a bus that can be snooped and without using broadcasts, we use a *prob_owner* field to coordinate the changes in ownership induced by an invalidate-based protocol. At each node, the *prob_owner* field indicates the last known owner of the data. Initially, the *prob_owner* fields for all data on all processors is set to the root node. It is updated when the following circumstances occur:

- a processor receives an invalidation request,
- a processor relinquishes ownership, which can happen only for a *write* fault, or
- a processor forwards a *write* request.

This scheme is slightly different from Ivy's, which also changes ownership on read faults. As discussed in Li's thesis [Li86], this scheme increases the worst-case bound on the number of messages required to locate K owners. Changing ownership on both reads and writes results in a worst-case number of messages for locating K owners on an N-processor system of $O(N + K \log N)$ [Li86]. However, as pointed out in Li's discussion of this phenomenon, this worst-case occurs when the data is written sequentially by each processor. We found that in practice the last writer of a data item was likely to be the next writer, even when other nodes read the data between the two writes. Thus, it is important that writers retain ownership and be able to invalidate the replicas without first following the *prob_owner* chain to re-acquire ownership and get the new copyset.

P(*lock*)
Get copy of *object* from *prob_owner*
prob_owner ← Node that responded to our request
V(*lock*)

Figure 3.8 Read Fault Handler for Conventional Data

```
P( lock )
if ( prob_owner = this_node ) begin
    /* Check for Mirage-style FREEZE */
    if ( copyset = {this_node} ) begin
        cur_time ← ReadTime()
        diff_time = cur_time - timestamp
        if ( diff_time < FREEZE_TIME )
            Delay( FREEZE_TIME - diff_time)
        Change protection to READ_ONLY
    end
    copyset ← copyset + {requesting_node}
    Send data to requesting_node
end
else Forward request to prob_owner
V( lock )
```

Figure 3.9 Read Server for Conventional Data

```
P( lock )
if ( prob_owner ≠ this_node ) begin
    if ( present = 0 ) Request data and copyset from prob_owner
    else Request copyset from prob_owner
end
InvalidateRemoteCopies( copyset )
copyset ← {this_node}
Change protection to READ_WRITE
timestamp ← ReadTime()
V( lock )
```

Figure 3.10 Write Fault Handler for Conventional Data

We incorporated a simplified version of the freezing mechanism from Mirage [FP89] 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). The use of 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. We tried various freeze times between 10 msecs and 250 msecs, and 100 msecs resulted in the best average performance.

```
P( lock )
if ( prob_owner = this_node ) begin
    /* Check for Mirage-style FREEZE */
    if ( copyset = {this_node} ) begin
        cur_time ← ReadTime()
        diff_time = cur_time - timestamp
        if ( diff_time < FREEZE_TIME )
            Delay( FREEZE_TIME - diff_time)
    end
    copyset ← copyset − {this_node}
    Send data (if requested) and copyset to requesting_node
    Invalidate object
end
else Forward request to prob_owner
prob_owner ← requesting_node
V( lock )
```

Figure 3.11 Write Server for Conventional Data

```
Invalidate object
prob_owner ← requesting_node
```

Figure 3.12 Invalidate Server for Conventional Data

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. We considered the option of making user threads run at a higher scheduling priority than Munin threads to eliminate this problem, but the resulting remote request starvation was such a serious problem that we had to abandon this option.

Migratory

The migratory sharing pattern is a sub-class of the conventional sharing pattern, where between synchronization points it is generally the case that only a single processor accesses it via either reads or writes. 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. Figures 3.13 and 3.14 contain Munin's protocol for migratory data. It is a simplification of the protocol that we use for conventional data, wherein every miss is treated as a write miss. Explicit invalidations are unnecessary because only the prior owner can have a cached copy, and it invalidates its copy when it responds to a request for the data. The *prob_owner* field is still used as a chain to find the current owner, but since every access is treated as a write, nodes that forward requests always update their *prob_owner* field. Thus, the worst-case number of messages for locating K owners of a migratory data item on an N-processor system is $O(N + K \log N)$. 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, as it reduces the number of faults and messages needed to migrate the data. It was used in the test programs that used a task queue model of parallelism, where it was easy to apply.

Read Only

Read-only data is writable only inside `user_init()`, which allows it to be initialized along with the rest of the program. Once the program returns from `user_init()`,

```
P( lock )
Get copy of object from prob_owner
prob_owner ← this_node
V( lock )
```

Figure 3.13 Fault Handler for Migratory Data

```
P( lock )
if ( prob_owner = this_node ) begin
      Send data to requesting_node
      Invalidate object
end
else Forward request to prob_owner
prob_owner ← requesting_node
V( lock )
```

Figure 3.14 Data Request Server for Migratory Data

read-only data is handled as if it were conventional, with the exception that the write fault handler prints an error message and forwards the exception request to the system debugger. 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.

3.5.4 Synchronization

Synchronization variables are accessed in a fundamentally different way than normal data objects [BCZ90a], so Munin does *not* provide synchronization through shared memory. Rather, Munin provides as primitives distributed locks, barriers, and condition variables. More elaborate synchronization objects, such as monitors and atomic integers, can be built using these basic mechanisms. Each Munin node maintains a synchronization object directory, analogous to the data object directory, containing state information for the synchronization data. All of Munin's synchronization primitives cause their invoking thread to block on an "acquire", and cause the local delayed update queue to be purged on a "release".

Munin's provision of a specialized synchronization package distinguishes it from most other DSM systems, which do not provide synchronization primitives outside of shared memory, except perhaps the ability to lock a single data item [BHJL86, DCM+90]. Conventional shared memory implementations for synchronization operations such as locking can lead to high overheads in multiprocessor cache systems [ALL89], leading researchers to develop more efficient algorithms for synchronization operations [GVW89, HFM88, MCS91]. Given the high overhead associated with sending even one extra message in a software DSM system, it is of paramount importance to provide a synchronization package separate from the shared memory implementation. Doing so allows us to optimize the mechanisms used to handle synchronization variables, which are accessed quite differently than the way in which shared data is accessed. Failure to do so can lead to poor performance for simple spinlocks such as witnessed in Ivy [Li86], or force system designers to augment their shared memory system with complex heuristics to support spinlocks as was done in Mether [MF90].

Locks

Munin employs a queue-based implementation of locks similar to existing implementations on shared memory multiprocessors [GVW89, MCS91]. Using queue-based locks allows a thread to request ownership of a lock and then block awaiting a reply without repeated queries. A distributed queue identifies the user threads waiting for each lock, wherein each enqueued thread knows only the identity of the thread that follows it on the queue. This design allows a release-acquire pair to be performed with a single message exchange if the acquire is pending when the release occurs. Experience on shared memory multiprocessors has shown that the use of queue-based locks can reduce the number of bus transactions performed [ALL89, GVW89, HFM88, MCS91]. Given the much higher communication costs of software DSM, using a queue-based implementation in preference to a spinlocking implementation is important.

When a thread wants to acquire a lock, it calls `AcquireLock()` (see Figure 3.15). If the lock is local and free ($free = 0$), the thread immediately acquires the lock and continues executing. If the lock is not free, the thread places itself at the tail of the local queue of waiting threads ($last \leftarrow$ `this_thread`) and then sends a request

```
P( semaphore )
/* Put self at end of local thread queue */
if ( first = NULL ) first ← last ← this_thread
else last.next_local_thread ← this_thread
last ← this_thread
last.next_local_thread ← NULL

previous_end ← end_of_line
end_of_line ← this_node
last.next_pid_in_queue ← LAST_IN_LINE

if ( free ) begin
      free ← 0
      lock_holder ← this_thread
      V( semaphore )
      return
end

V( semaphore )

Send request for lock to previous_end
/* The send will block until lock ownership is acquired. */
lock_holder ← this_thread
```

Figure 3.15 Lock Acquisition Protocol

towards the end of the distributed queue, for which *end_of_line* is a hint analogous to the *prob_owner* field for conventional data. Using a distributed queue may require the request to be forwarded multiple times, as can be seen at the top of Figure 3.16. When the request arrives at node with the tail of the distributed lock queue, the servicing thread immediately forwards the lock to the requester if it is free. Otherwise, it adds the requesting thread to the end of the distributed lock queue (*last.next_in_line* ← `requesting_pid`). When the thread holding the lock calls `ReleaseLock()` and the associated queue is non-empty, lock ownership is forwarded to the next thread in the queue (*first.next_in_line*).

Before a lock can be passed between nodes, the delayed update queue must be flushed. Currently this is done by flushing the delayed update whenever a release is performed. An alternative that we did not investigate would be to only flush the queue when a lock that was held locally is sent to a remote node. This scheme avoids unnecessary updates when the same node reacquires a lock. Flushing the DUQ before passing a lock between nodes satisfies the requirements of release consistency that all writes be globally performed before a release can be performed. If any migratory data has been associated with the lock using the `AssociateDataAndSynch()` call, the data is sent along with the lock.

Barriers

A barrier is created using the `CreateBarrier()` routine, which requires a count of the number of threads that must reach the barrier before it is lowered. When a thread wishes to wait at a barrier, it calls `WaitAtBarrier()`, which flushes the local delayed updates, sends a message to the central barrier server, and awaits a response[8]. After the barrier server has received the requisite number of barrier request messages, it sends a single reply message to each waiting thread to wake it up. We debated the

```
P( semaphore )
if ( end_of_line ≠ this_node ) begin
    Forward request to end_of_line
    end_of_line ← requesting_node
    V( semaphore )
    return
end

if ( free ) begin
    free ← 0
    Purge delayed updates
    Reply with lock and any migratory data associated with lock
end
else last.next_pid_in_queue ← requesting_thread

end_of_line ← requesting_node
V( semaphore )
```

Figure 3.16 Lock Server Acquisition Protocol

[8]Munin provides a means for the programmer to make a per-processor hint of how many threads will be waiting at the barrier for any particular processor. If this value is greater than one, then the local runtime only flushes the delayed modifications when the *last* of the local threads arrives at the barrier. Only flushing the DUQ when the last thread on a processor arrives at the barrier reduces the number of unnecessary flushes of the delayed update queue.

P(*semaphore*)
if ($lock\_holder \neq$ `this_thread`) **begin**
    Print error message
    V(*semaphore*)
    return
end

Purge delayed updates

if ($first.next\_in\_line = LAST\_IN\_LINE$) **begin**
    $free \leftarrow 1$
    V(*semaphore*)
    return
end

Send lock and any migratory data associated with lock to $first.next\_in\_line$

$first \leftarrow first.next\_local\_thread$
if ($first = NULL$) $last \leftarrow NULL$
$free \leftarrow 0$
V(*semaphore*)

Figure 3.17 Release Lock Protocol

utility of using a distributed barrier mechanism similar to those designed for scalable multiprocessor systems [HFM88], but decided that for the size of the prototype implementation, a simple centralized scheme would suffice. In particular, all the barriers used in our test suite were used to mutually synchronize *all* of the user threads in the program. In this case, the centralized algorithm requires fewer total messages than the more sophisticated distributed algorithm.

Condition Variables

The semantics of Munin's condition variables are as follows:

- Initially a condition's value is set to FALSE.
- If a thread performs a `WaitCondition()` call and the value of the condition variable is TRUE, it receives an immediate reply and the value of the condition is set to FALSE.

- If instead the value is FALSE, the thread is placed at the end of the queue of threads waiting on the condition.
- If a thread performs a `SignalCondition()`, the thread at the front of the queue is resumed. If none are waiting, the condition variable is set to true. `SignalCondition()` itself is non-blocking.
- Finally, if a thread performs a `BroadcastCondition()`, all threads waiting on the condition are resumed.

The semantics of Munin's condition variables are slightly non-standard, as we do not support automatic signaling of conditions based on changes to state variables [LR80], but instead require the user to perform signals explicitly. Except for the ability to broadcast to all threads waiting on a condition, Munin's condition variables can be used as distributed binary semaphores (`P()` and `V()`). Condition variables give threads the ability to synchronize indirectly and thus stall until another thread signals them to continue. Any thread can perform a signal, which differentiates condition variables from locks because only the lock owner can release a lock. While it is possible to build condition variables using locks, we found it to be more efficient to include them as primitives. Like barriers, Munin's protocol for condition variables uses a centralized server to control the queue. We considered building the condition variable protocol on top of the locking protocol, since in the absence of broadcasts they have roughly the same semantics, but because condition variables are used infrequently in our test programs, we opted for the simplest solution. In accordance with the requirements of the release consistency model, the DUQ is flushed before the signal or broadcast message is sent to the condition server.

3.6 Summary

This chapter contains a detailed description of the design and implementation of the Munin prototype, with special emphasis given to its novel features. 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 are relevant to the implementation of future DSMs. These include:

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 that gets linked with user programs 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. Providing synchronization support through an independent library rather that through shared memory improves performance without impacting programmability. Simple synchronization algorithms works quite well in the prototype, although it is not clear how well they will scale.

4. 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, asynchronous communication, or multicast.

5. 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. Adding protocols that do not differ significantly from existing ones has little impact on performance and can make the task of correctly annotating shared variable declarations more difficult.

Chapter 4

Results

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 a message passing version. The message passing programs represent best case implementations of the parallel programs[9]. 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 cumulative impact of Munin's novel features, two basic comparisons were performed. First, 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. For MULT, DIFF, TSP-C, and FFT, Munin achieved performance within 5% of the speedup of message passing. For TSP-F, QSORT, and GAUSS, Munin's performance is within 29% to 34%. Second, 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. For the programs with large grained sharing (MULT and TSP-C), the conventional versions achieved

[9]The message passing programs are constrained to use the same basic algorithm and data structures as the shared memory programs for comparison purposes, so they might not be the true best case programs for the hardware. However, they were tuned with distribution in mind.

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 50-80% of Munin. For QSORT, conventional performance was under 50% of Munin performance, while for FFT conventional performance was orders of magnitude worse.

After evaluating the cumulative impact of Munin's novel features, we examined the effects on performance of the individual techniques for reducing communication described in Chapter 2: software release consistency using a multi-writer protocol, support for multiple consistency protocols, and the use of an update timeout mechanism.

Finally, to determine the sources of DSM overhead, we examined more closely those programs for which the Munin version did not perform within 10% of the message passing version. We found that the addition of a function shipping capability brought the performance of the Munin versions of both programs to within 5% of their message passing equivalents. When a function shipping capability was added to the same conventional DSM programs, one exhibited similar improvement, while the other did not improve at all because of the presence of false sharing. This latter point reinforces the point that for DSM to be useful across a broad spectrum of programs, the problem of false sharing must be solved, and that Munin's novel features can significantly reduce the impact of false sharing.

4.1 Description of Test Programs

This section describes the test programs that were used to evaluate Munin, with special emphasis on their major data structures, the way in which they were parallelized, and the amount of sharing involved in the parallel implementations. Throughout this description and the rest of this chapter, N refers to the problem size, while P refers to the number of processors.

Matrix Multiplication

Matrix multiplication (MULT) is the canonical trivially parallelizable program. The problem is to multiply two N by N input arrays, $A[N][N]$ and $B[N][N]$, and put the result in $C[N][N]$. The work can easily be easily decomposed into disjoint sections by giving each worker thread its own portion of the result array to compute. Other than any indirect effects caused by false sharing, each thread can independently calculate its portion of the result array, as the input arrays are read-only and each element of the result array is accessed by only one thread.

The version of MULT used in the evaluation distributes a number of contiguous rows of the result array to each worker thread. The worker threads simply loop computing their assigned portions of the result. The DSM versions use a barrier to signal completion. After each worker thread has terminated, the root thread reads in the result array and terminates. The Munin version declares the output array as `shared write_shared output[N][N]`, because the uncoordinated writes to the output array are to disjoint portions of the array.

Finite Differencing

The basic finite differencing (DIFF) algorithm is a simple iterative process. The area being modeled 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, every matrix element is updated to some function of the elements near it. In our test program, this function is the average of its nearest neighbors (above, below, left, and right). To avoid overwriting the old value of a matrix element before it is used for the computation of its neighbors' new values, the program can either use a scratch array or only compute every other element per iteration (the so-called "red-black" technique). Both techniques are equally valid. The scratch array approach was employed because the "red-black" approach requires communication after each half-iteration, while the scratch array approach only requires communication after the scratch array is copied back into the main matrix. This applies both for the message-passing and DSM versions.

DIFF can be parallelized by dividing the area into subarrays and assigning each worker thread to a subarray. An example where the matrix is decomposed into consecutive rows is illustrated in Figure 4.1. Newly computed values along the boundaries between two threads' subarrays must be exchanged with adjacent nodes at the end of each iteration. These elements are shaded in the illustration. This exchange engenders a *producer-consumer* relationship between grid points at the boundaries of adjacent sections. The algorithm requires only those elements that lie directly along the boundary between two threads' subarrays to be communicated at the end of each iteration. This organization presents the potential for false sharing in a page-based system. Figure 4.1 illustrates the problem of false sharing using the dashed box to

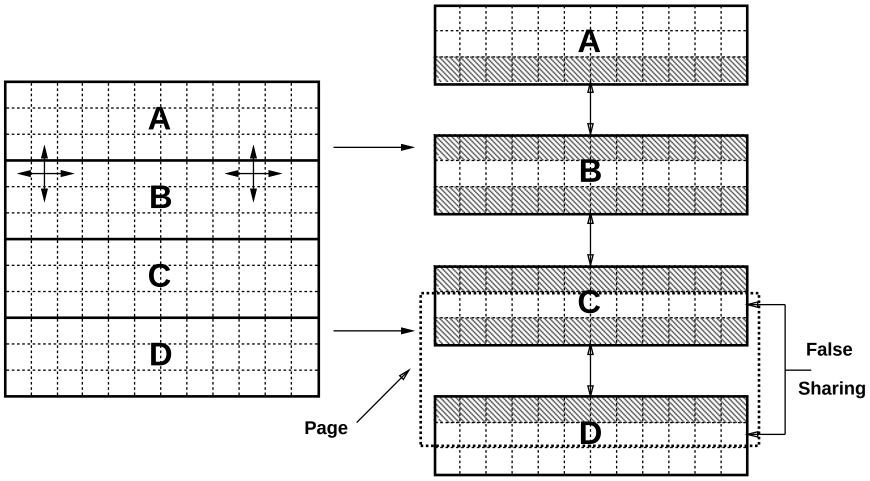

Figure 4.1 Parallel Finite Differencing, $N = 4$

represent a page. All non-edge elements within the dashed page are falsely shared. Any communication other than the edge elements is unnecessary. Thus, nodes need only communicate at the end of each iteration, at which point they should exchange only the edge elements.

For all versions of the program, the work is decomposed by rows as illustrated in Figure 4.1, since this is how the C compiler lays out arrays. Each thread is assigned $\lfloor N/P \rfloor$ rows, with extra rows being divided evenly amongst the threads. In the DSM versions, the programmer is *not* required to specify the data partitioning to the runtime system – it is inferred by the runtime based on the observed access pattern. This lack of explicit partitioning differentiates Munin from compiler-only systems [BFKK90, ZBG88]. After each iteration, the DSM worker threads synchronize by waiting at a barrier, while the message passing workers exchange results directly between neighboring nodes. After all workers have completed all iterations, the program terminates.

In the Munin version of DIFF, the matrix representing the grid is annotated as `shared write_shared matrix [...]` because it is modified in parallel by multiple threads, which can lead to false sharing. This program annotation instructs Munin to buffer the changes to the array, which results in Munin only sending data when a thread reaches the barrier at the end of each iteration. The conventional DSM version, on the other hand, ping pongs the shared pages back and forth, albeit infrequently.

Traveling Salesman Problem

The Traveling Salesman Problem (TSP) finds the minimum length path that starts at a designated node, passes through every other node exactly once, and returns to the original node. A complete path is known as a *tour*. The most brute force solution to this problem tries all possible path permutations and selects the shortest one, but the time to check all possible permutations is prohibitive. However, several simple optimizations can improve performance. The most basic optimization is that if the current length of a subtour plus a lower bound of the remaining portion of the path is longer than the current shortest tour, the subtour can be ignored because it cannot lead to a shorter tour than the current minimum length tour. This process of removing subtrees from the search space is known as pruning. The version of TSP used in the evaluation maintains a priority queue of the partially evaluated tours, represented by a sorted heap, sorted in inverse order of a lower bound of their total length. This lower bound is calculated using a fast greedy algorithm. Thus, the subtours that are "most promising" are evaluated first, making it likely that the shortest tour will be found near the beginning of the search, which tends to prune uninteresting subtours quickly.

In the DSM versions, the major shared data structures are the global minimum tour and tour length, an array of structures describing both partially evaluated and unused tours, a priority queue containing pointers to partially evaluated tours, and a stack of pointers to tour structures that can be reused. A TSP program written specifically for a distributed memory multiprocessor would not normally use indirection arrays as this one does, but we chose not to modify the code to demonstrate the effectiveness of DSM for unmodified shared memory programs. In the Munin version, the array of tour structures is declared to be `write-shared`, because different individual tours (array entries) are modified in parallel by multiple threads. The priority queue and stack of unused tours are declared to be `migratory`, because they are accessed exclusively within critical sections controlled by locks. Unlike the other shared variables, the optimal annotations for the global minimum tour variables were not obvious. `Read-only` is clearly inappropriate, and they are read frequently by multiple threads in parallel, which eliminates `migratory` as a good option. The fact that the minimum tour variables are modified infrequently would seem to indicate that the `write-shared` and `conventional` protocols should work equally well, but it turns out that the update-based `write-shared` protocol performs noticeably better, for reasons described in Section 4.5. This may seem to indicate that the annotation process is tricky, but in practice there are few variables for which the choice was not obvious.

In the DSM versions, a lock protects the priority queue, and a condition variable is used to signal when there is work to be performed. Worker threads acquire the lock and remove partial tours from the queue until a "promising" tour has been found that can be expanded sequentially, at which time the lock is released. For the message passing version, the *master* maintains the priority queue that contains the indices of subtours to be solved. The *slaves* send request messages to the master, which performs the subtour expansion and responds either with a subtour to be solved directly, or an indication that there is no more work. Workers tell the master when they find a new global minimum, and the master is responsible for propagating it.

Two basic TSP experiments were performed, a *fine-grained* TSP and a *coarse-grained* TSP. Both experiments searched for the minimum length tour of eighteen cities. The difference between TSP-C and TSP-F is their definition of "sufficiently short" when determining whether to recursively solve a partially expanded tour. TSP-C solves the tours recursively when there are thirteen nodes out of eighteen remaining, while TSP-F solves the tours recursively when there are twelve nodes out of eighteen remaining.

Quicksort

Quicksort (QSORT) is a recursive sorting algorithm that operates by repeatedly partitioning an unsorted input list into a pair of unsorted sublists, such that all of the elements in one of the sublists are strictly greater than the elements of the other, and then recursively invoking itself on the two unsorted sublists. In the implementation used in the evaluation, the base case of the recursion occurs when the lists are sufficiently small, at which time they are sorted directly. The definition of "sufficiently small" is an input parameter to the program. QSORT is parallelized using a work queue that contains descriptors of unsorted sublists, from which worker threads continuously remove unsorted lists.

The DSM version of QSORT differs from the DSM version of TSP in that when it releases control of the task queue, it may need to further subdivide the work by partitioning the subarray and placing the new subarrays back into the task queue, whereas TSP workers do not relinquish control of the task queue until they have removed a subtour that can be solved directly without further expansion. Thus, the task queue in QSORT is accessed more frequently per unit of computation. Offsetting this is the fact that the threads in TSP hold the exclusive lock protecting the priority queue for a longer time as they perform the expansion, and the thread holding the lock may in fact fault when attempting to access the structure containing partially evaluated tours while holding exclusive access to the priority queue.

In the DSM versions of QSORT, the major data structures are: an array to be sorted, a task queue that contains range indices of unsorted subarrays, and a count of the number of worker threads blocked waiting for work. In the Munin version, the array being sorted and the count of the number of threads awaiting work are declared to be `write-shared`, while the task queue is declared to be `migratory`. A lock protects the task queue, and a condition variable signals the presence of work to be performed. The worker threads repeatedly get work from the task queue and quicksort it. While doing so, worker threads push new work on to the task queue whenever they partition a subarray too large to solve directly. Getting work entails acquiring the lock protecting the task queue, reading the first element (if any), and then releasing the lock. If there is no work to be done, the thread checks to see if all of the threads are waiting. If they are, it signals the other threads and terminates, and if not, it waits on the condition variable until work is inserted or another thread signals that the computation is complete.

For the message-passing version, the *master* maintains the work queue and performs the partitioning on demand. The *slaves* send request messages to the master, which responds either with the sublist to be sorted directly or an indication that there is no more work. Along with the requests, the slaves ship the sorted results from their previous request, if any. Thus, in the message-passing version, the only data shipped are the unsorted sublists from the master to the slaves and the sorted sublists from the slaves to the master.

Fast Fourier Transform

The Discrete Fourier Transform $X(k)$, of a periodic continuous time function $x(n)$, can be computed using the equation: $X(k) = \sum_0^{N-1} x(n)\omega^k$, where N is the number of sample points, k is the index of the frequency domain representation of the signal, and the ω^k values are constants that are pre-computed during initialization and stored in an array. The FFT program used in the evaluation is based on the Cooley-Tukey

Radix 2 Decimation in Time algorithm [BP85]. It recursively subdivides the problem into its even and odd components ($x_1(n) = x(2n), n = 0, 1, ..., N/2 - 1$ and $x_2(n) = x(2n+1), n = 0, 1, ..., N/2 - 1$), until the input is of length 2. For this base case, the output is an elementary function known as a Butterfly, a linear combination of its inputs and the ω values.

For an input array of size N, the FFT algorithm requires $\log_2 N$ passes. On pass K, the width of each butterfly is $2^{\log_2 N - K - 1}$ or $N2^{-(K+1))}$. Thus, for the first pass ($K = 0$), the new value of $x[0]$ is a function of $x[0]$ and $x[N/2]$, and similarly the new value of $x[N/2]$ is a function of $x[N/2]$ and $x[0]$. On the second pass, the new value of $x[0]$ is a function of $x[0]$ and $x[N/4]$, and the width of each butterfly is now $N/4$. On each subsequent iteration, the width of each butterfly halves, until during the final iteration, the new value of $x[0]$ is a linear combination of $x[0]$ and $x[1]$. By starting with the wide butterflies, the result array is a permutation of the desired value, but this is rectified with an $O(N)$ cleanup phase.

If P processors are used to solve an N point FFT, then a reasonable initial decomposition of the work allows processor p to work with $x[p]$, $x[p+P]$, $x[p+2P]$, ..., $x[p+N-P]$. This allows all processors to perform the first $\log_2 N - \log_2 P$ passes without any inter-processor communication. Before executing the last $\log_2 P$ iterations, the processors exchange data and reallocate themselves to different (contiguous) subarrays, as illustrated in Figure 4.2. This figure illustrates an 8-element FFT being performed by two processors ($N = 8$, $P = 2$), where the dashed lines represent computations by p_0 and the dark lines represent computations by p_1.

Both the DSM and message passing programs are parallelized by dynamically allocating threads to data in such a way that during each phase of the computation, a given thread uses an independent set of data elements. By carefully allocating processors to data, it is possible to only reallocate the processors and exchange data at the end of the first $\log_2 N - \log_2 P$ phases. The DSM programs use a barrier to synchronize at this point. The DSM system automatically reallocates the data on demand. The message passing version manually encodes and shuffles the data, using a master process to collect and redistribute all the changes. This manual redistribution made the message passing version more difficult to write than the DSM versions. The processor reallocation is built into the algorithm itself.

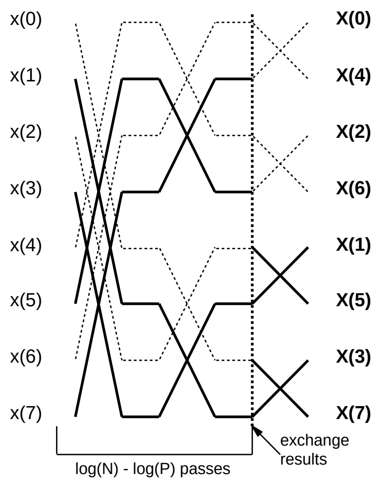

Figure 4.2 Parallel FFT: $N = 8$, $P = 2$

Gaussian Elimination with Partial Pivoting

Gaussian Elimination (GAUSS) decomposes a square matrix into upper and lower triangular submatrices by repeatedly eliminating the elements of the matrix under the diagonal, one column at a time. The basic algorithm for an N by N matrix is shown in Figure 4.3. For each iteration of the i-loop, the algorithm subtracts the appropriate multiple of the i^{th} row of the matrix from the rows below it, so that the elements below the diagonal in the i^{th} column are zeroed. Partial pivoting improves the numerical stability of the basic algorithm by interchanging the i^{th} row with the row in the range $[i+1...N-1]$ containing the largest (in absolute value) element of the i^{th} column. Algorithmically, this involves inserting a phase between the i and j loops that searches the i^{th} column for the pivot element and swaps that row and the i^{th} row.

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

Figure 4.3 The Basic Gaussian Elimination Algorithm

It is fairly easy to parallelize this algorithm. We decomposed the computation by column so that the pivoting phase, which can be a synchronization bottleneck, can be performed on a single processor. Each thread gets roughly $\lfloor N/P \rfloor$ columns. The computation itself involves N iterations, one per column, where each iteration consists of a pivoting phase and a computation phase.

The DSM versions are parallelized as follows. Each iteration starts with a barrier. After the barrier falls, the thread responsible for the current column performs the necessary pivoting, sets a shared pivot row variable to identify the row that needs to be pivoted with the current one, and copies the current column to a shared variable to be used by the other threads during the computation phase. A barrier is used to separate the pivoting and computation phases. After the barrier is passed, each thread performs the actual computation on its portion of the data, which involves performing the local pivoting, followed by the elimination step shown in Figure 4.3.

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

4.2 Experimental Methodology

For all three versions of each program, the sequential initialization routine is executed on the root node. Then the appropriate number of additional Munin nodes are created, which gives each node a copy of the non-shared data. The non-root nodes initialize themselves, and then synchronize with the root node by waiting at a barrier for the DSM versions and via an explicit message in the message passing versions. For the DSM versions, after the user thread on the root node creates the required worker threads, it reads the clock to get the initial value and then waits at the barrier, which causes the computation to begin. For the message passing versions, the root thread waits until all the worker threads have sent it the initialization message. It then reads the initial clock value and sends a message to each of the workers to start computation. At this point, the workers read their inputs, via page faults for the DSM versions, or via request messages for the message passing versions. Once all the workers have completed, the root thread again reads its clock and calculates the total elapsed computation time. For the sequential case, we only measure the compute time, without performing any synchronization or other unnecessary work.

All of the experiments were performed on a network of sixteen SUN-3/60 workstations using the V operating system[CZ83]. These workstations contain 20 MHz Motorola MC68020 microprocessors[Mot] and AMD LANCE Ethernet interfaces[Adv]. They were connected to a dedicated 10 megabit per second Ethernet[MB76] that was isolated from other campus networks during the experiments through the use of a software-controllable hardware switch. Isolating the network during the experiments guaranteed that the results were not affected by network traffic generated from external sources, either from spurious broadcasts or increased network contention.

The basic measurement performed for each experiment (run of a particular version of each program on a particular number of nodes) was its execution time. These times were used to calculate two standard measures of parallel system performance: *speedup* and *efficiency* [EZL89]. The *speedup* of a program on N processors is calculated by dividing the execution time of the program on one processor by the execution time on N processors. *Efficiency* is used to compare the performance of two versions of a program. The efficiency of version A of a program as compared to version B on N processors is calculated by dividing the speedup of version A of the program on N processors by the speedup of version B of the program on N processors. For example, if the message passing version of a program achieves a speedup of 8 on N processors, while the Munin version of the same program achieves a speedup of 4 on N processors, the efficiency of Munin relative to message-passing is said to be 50% for that program on N processors. Using efficiency, the quality of different implementations of different programs can be compared meaningfully. Both speedup and efficiency are used throughout the following analysis when comparing the quality of the various implementations to one another.

In addition to execution times, the Munin runtime system collects a large amount of state information for measuring such things as the number of faults, the amount of data transferred, and the amount of time stalled while performing various consistency operations. The message passing kernel collects similar data. Selected portions of these statistics are used throughout the analysis to highlight the reasons for observed performance differences between the different versions of the programs.

4.3 Quality and Diversity of Programs

An effort was made to select a suite of programs that represent a wide spectrum of shared memory parallel programs so that the results are not specific to a particular style of parallel programming. To that end, the programs selected use a number of different parallelization techniques, vary in the extent to which the data accessed by a particular thread changes over the course of the program execution, access data at differing basic granularities, and exhibit varying amounts of sharing. MULT and DIFF are numeric problems that statically distribute the data across the threads and access shared memory in predictable patterns. Both MULT and DIFF access data at a fairly large grain and exhibit only a limited amount of sharing. GAUSS, on the other hand, while accessing data at a fairly large grain, exhibits a much higher degree of sharing because of the pivoting process. Furthermore, the granularity of the problem shrinks quadratically as the problem progresses because of the nature of the algorithm. FFT also accesses shared memory in a predictable pattern, but the pattern is quite complicated and involves dynamically reallocating the data across threads on a fine-grained basis. Also, FFT exhibits an extremely high degree of sharing. TSP and QSORT use the task queue model to continuously subdivide the data into a series of subcomputations and to dynamically allocate work to threads. They both have moderate degrees of sharing, both true sharing between the time when a chunk of work is created and when it is performed, and false sharing between threads working on pieces of work that happen to be found in the same page of the work arrays. Finally, the granularity for TSP was varied (TSP-C and TSP-F access data at a coarse and fine grain, respectively). The problem sizes used in the experiments are given in Table 4.1. These problem sizes were chosen so that the uniprocessor running times would be in the hundreds of seconds, and the sixteen processor running times would be in the tens of seconds, or in other words, small to moderate sized problems.

Figures 4.4 and 4.5 illustrate the speedup and efficiency compared to ideal speedup for the message passing versions of seven programs. The X-axis for both graphs is the number of processors used in a particular run of the given program. The Y-axis of Figure 4.4 is the speedup achieved by running the program on a given number of processors, where the ideal curve represents the situation where the speedup on N processors is N. The Y-axis of Figure 4.5 is the percentage of the ideal speedup that the message passing version of the program achieves on the given number of processors. The closer the curve is to the 100% line at the top of the graph, the closer the program was able to come towards achieving perfect linear speedup.

The programs that access data at a fairly coarse grain (MULT, DIFF, QSORT, and TSP-C) are able to achieve speedups within 20% of the ideal for 16 processors

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

Table 4.1 Problem Sizes Used in Experiments

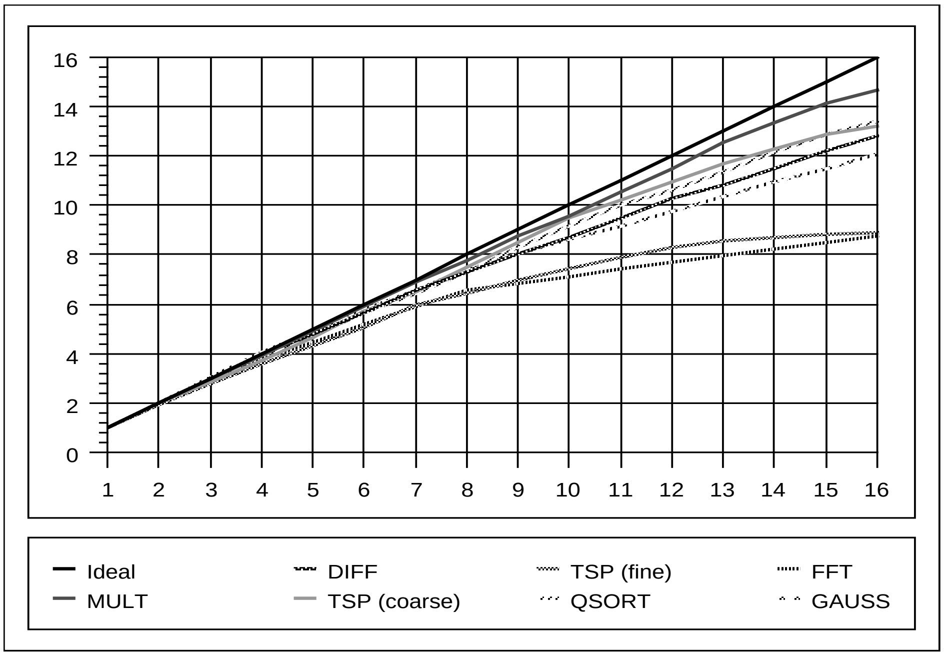

Figure 4.4 Speedup of Message Passing Programs

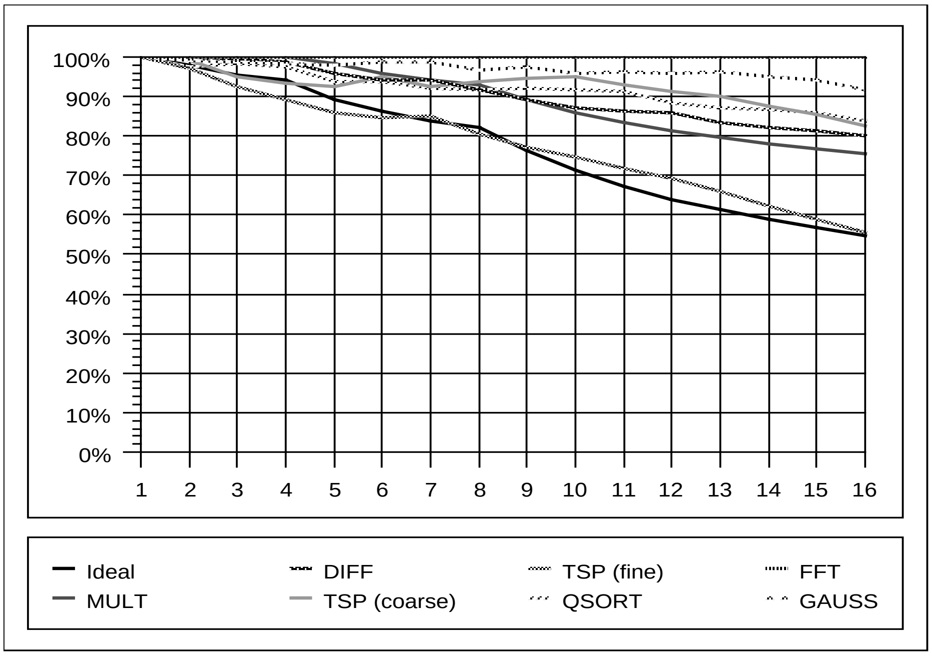

Figure 4.5 Efficiency of Message Passing Programs

and within 10% for 8 processors. This indicates that for these problems, there is considerable parallelism available. As the amount of sharing increases and the granularity of a given problem decreases, the performance of the programs falls off. GAUSS achieves 75% of ideal at 16 processors, while TSP-F and FFT achieve slightly over 50%. GAUSS's dropoff was caused by the small granularity of the work compared to the amount of communication required, while the dropoff of TSP-F and FFT were caused by the amount of communication required by the algorithm. This wide variance in performance is further evidence that the test suite used to evaluate Munin covers a wide spectrum of parallel programs.

Individual Speedup Curves

Figures 4.6 through 4.12 present the speedup achieved by the three different versions of each application. The X-axes represent the number of processors and the Y-axes represent the amount of speedup achieved.

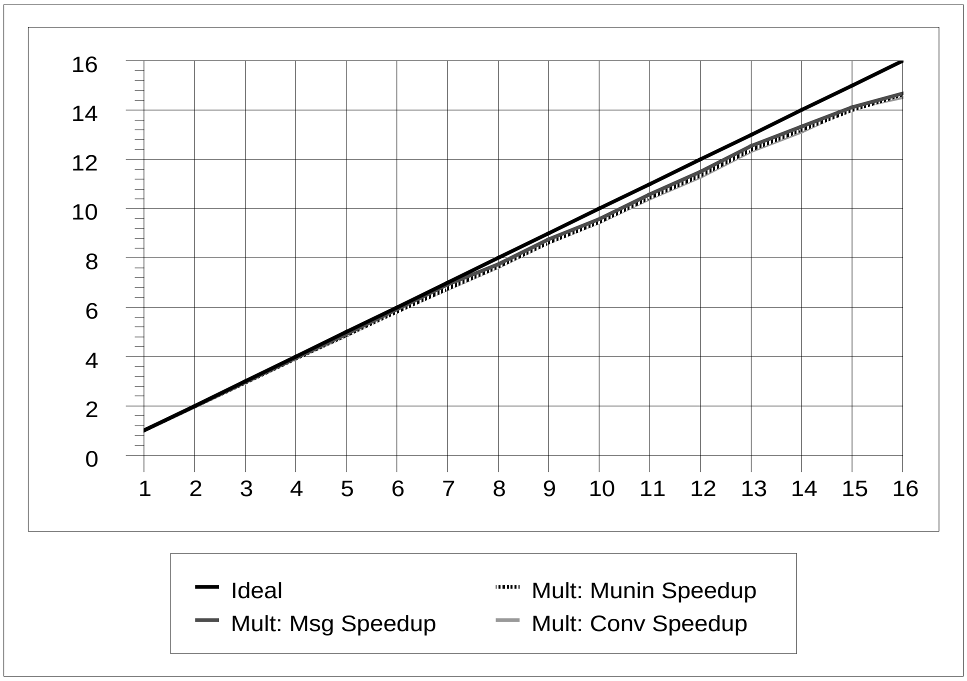

Figure 4.6 Matrix Multiplication (MULT)

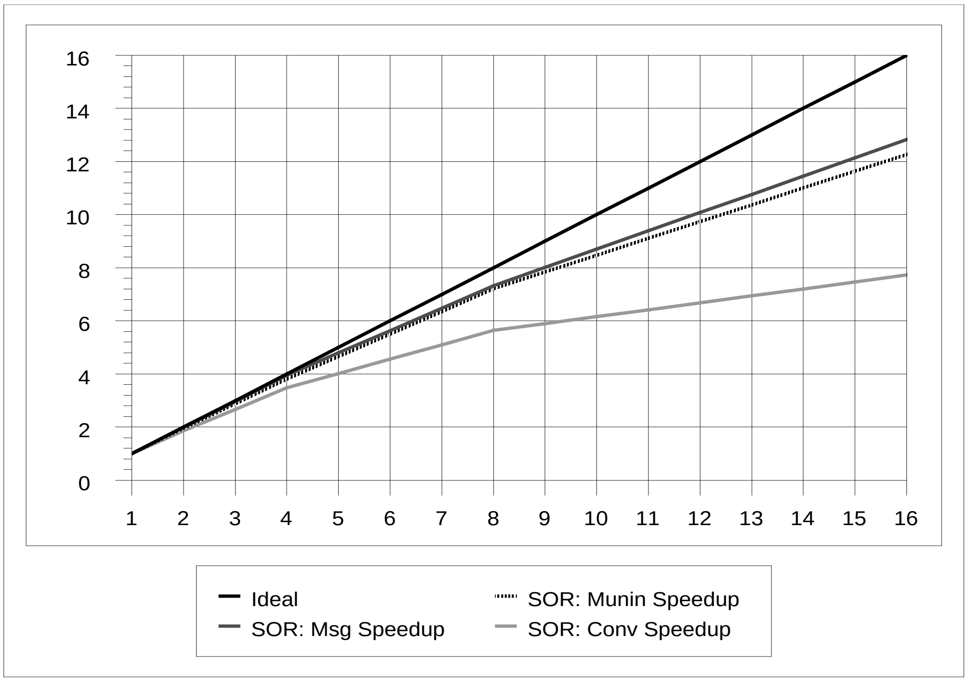

Figure 4.7 Finite Differencing (DIFF)

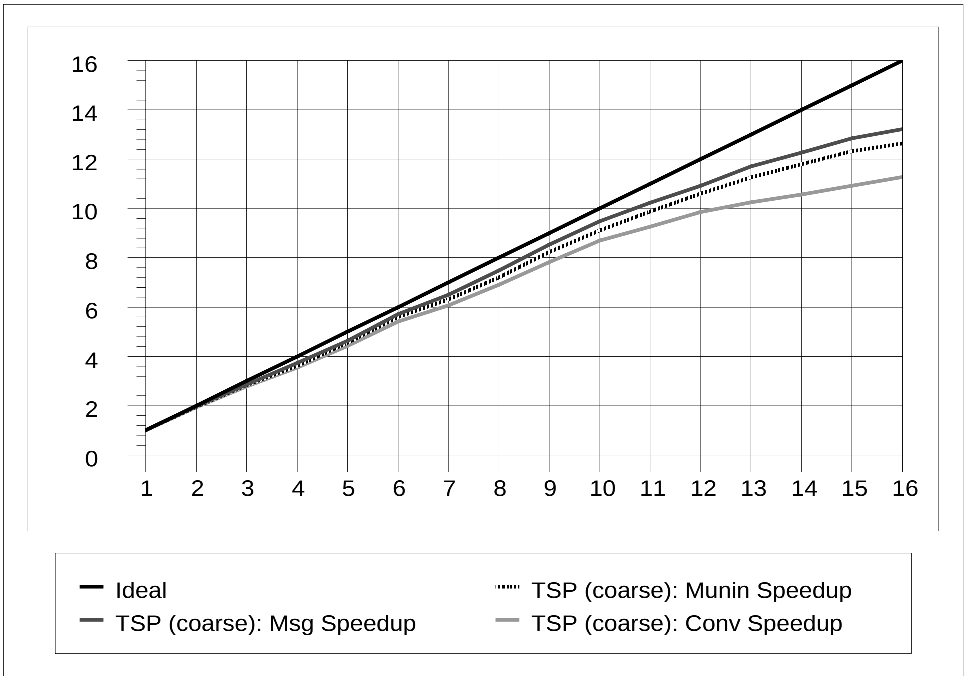

Figure 4.8 Coarse-Grained Traveling Salesman Problem (TSP-C)

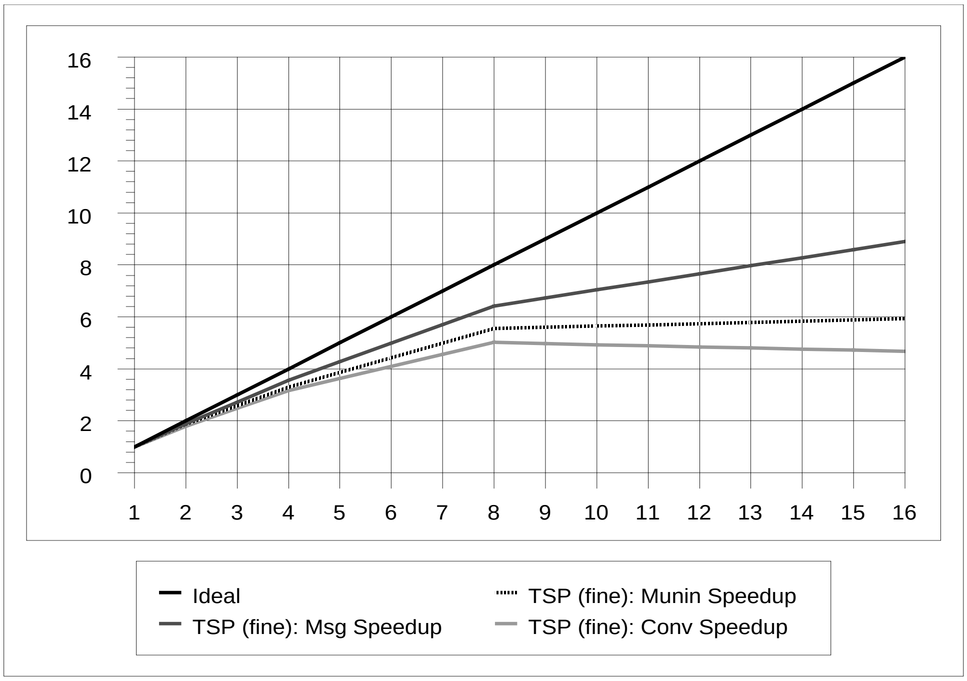

Figure 4.9 Fine-Grained Traveling Salesman Problem (TSP-F)

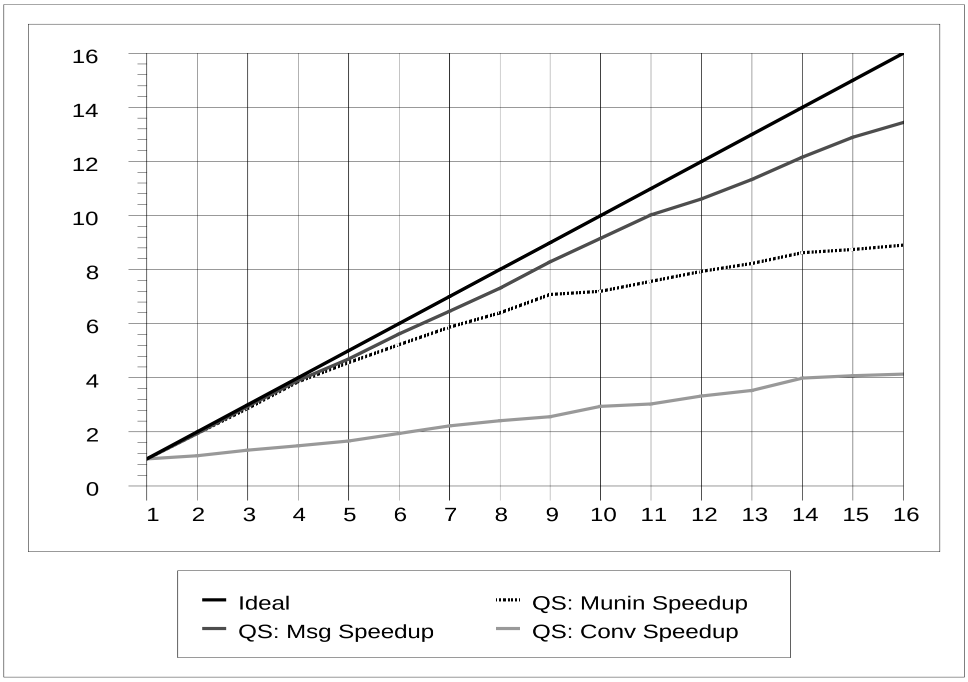

Figure 4.10 Quicksort (QSORT)

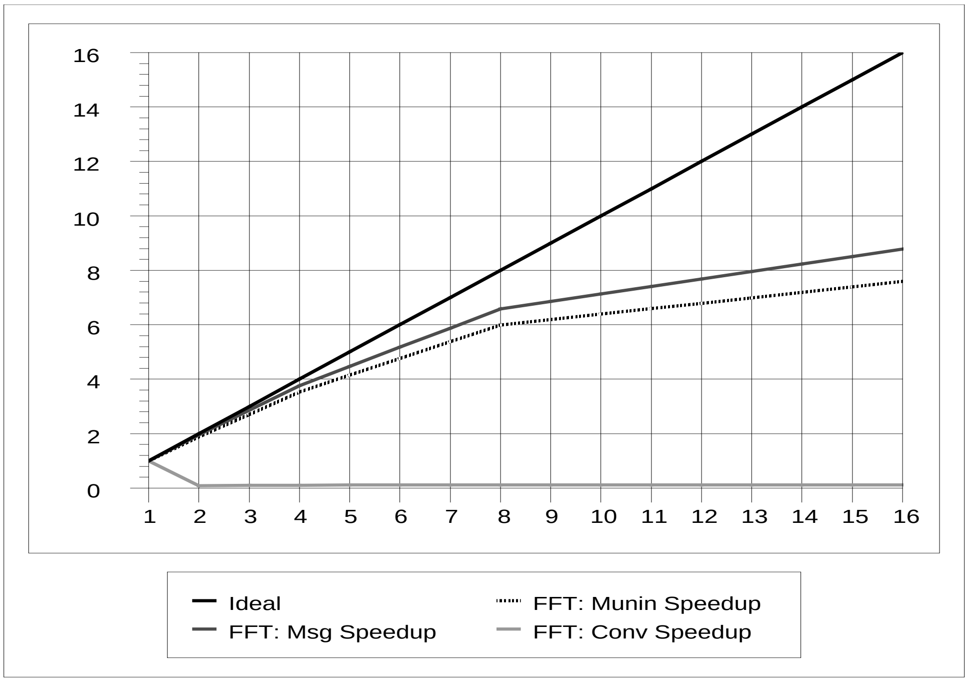

Figure 4.11 Fast Fourier Transform (FFT)

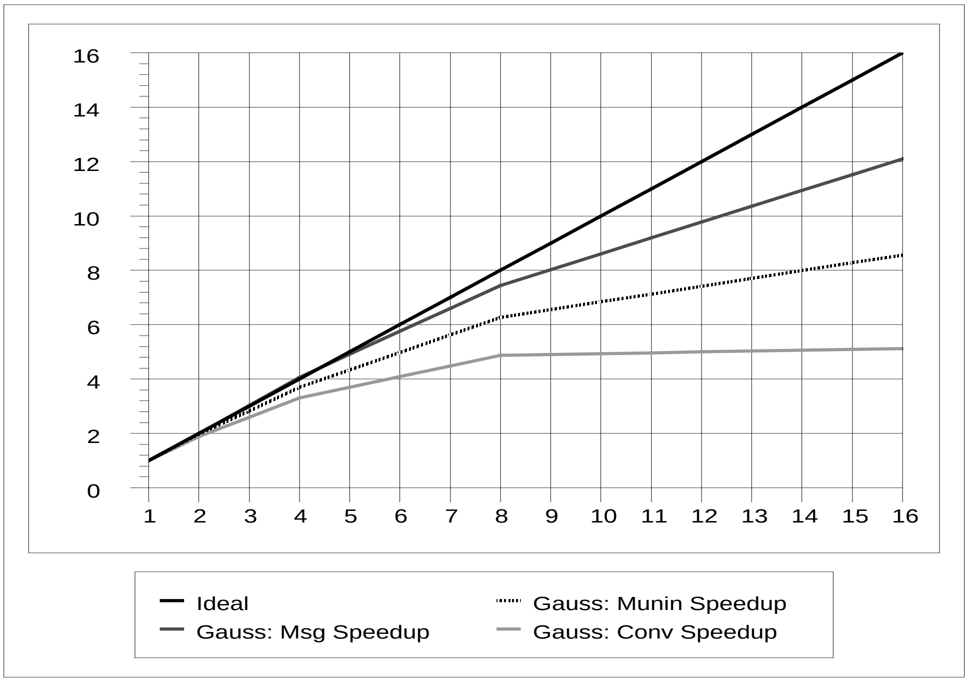

Figure 4.12 Gaussian Elimination with Partial Pivoting (GAUSS)

4.4 Munin DSM versus Message Passing

First, we compare the performance of a collection of programs running under Munin with equivalent message passing programs. For this comparison, each of the programs were run on from one to sixteen processors. The overall results of this experiment can be found in the relative efficiency graph found in Figure 4.13.

For four of the applications (MULT, DIFF, TSP-C, and FFT), the Munin versions achieve over 93% 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 unoptimized Munin variants is between 65% and 75% of their message-passing equivalents.

The detailed performance statistics gathered by the runtime system can provide insight into the reason for the observed efficiencies. Table 4.2 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. Table 4.3 presents the amount of communication performed during execution of the

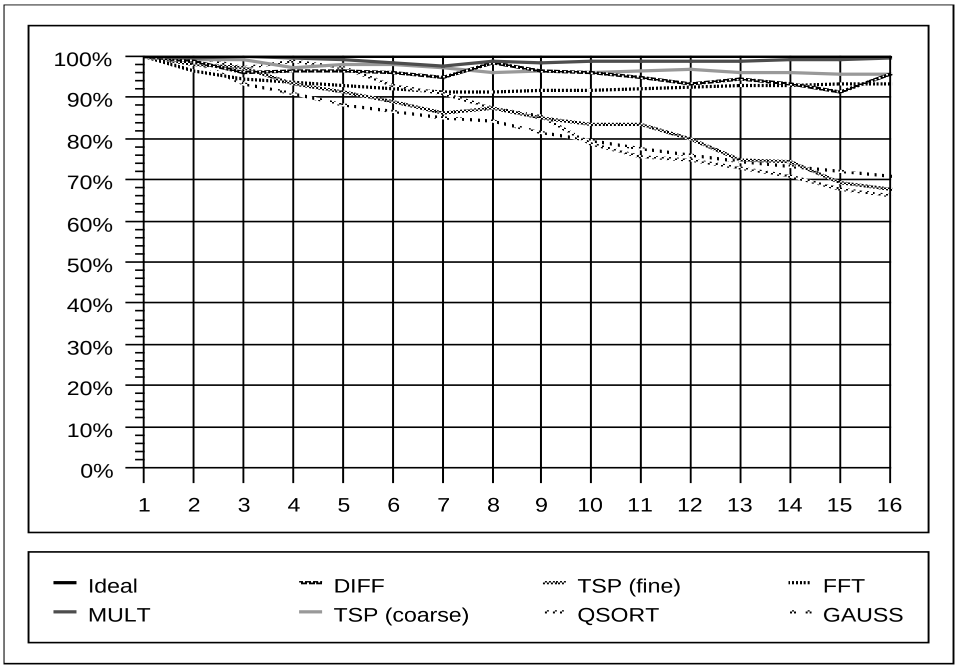

Figure 4.13 Efficiency of Munin DSM Compared to Message Passing

programs on sixteen processors, including both the number of messages and kilobytes of data transmitted.

The following program-by-program analysis of the specific reasons for the observed performance of Munin identifies the sources of overhead induced by Munin, which leads to a better understanding of the types of programs and program constructs for which Munin is best suited.

MULT

MULT is almost completely compute-bound, so the overhead added by Munin has only a small impact on performance. The Munin version performs approximately twice as much communication as the message passing version, because it faults in the zeroed out result array at the beginning of the computation, rather than simply initializing a local array as is done by the message passing version. However, compared to the overall execution time, the time spent communicating is insignificant, so the hand-coded message-passing and Munin versions perform almost identically. In particular, the cumulative computation time is 897 seconds, while the cumulative

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

Table 4.2 Munin vs Message Passing: Speedup (16 processors)

Program	Munin		Message Passing	
	Messages	Kilobytes	Messages	Kilobytes
MULT	1567	1384	672	640
DIFF	14,646	3645	14,164	8294
TSP-C	7870	4163	902	68
TSP-F	9776	4989	919	68
QSORT	31,866	14,565	667	524
FFT	15,322	11,621	9225	9339
GAUSS	26,034	5526	14,768	4995

Table 4.3 Munin DSM vs Message Passing: Communication (16 processors)

communication time is roughly 4 seconds, 3.6 seconds of which is "wire time" and 0.2 seconds of which is the time spent updating copies of shared pages when a thread arrives at the final barrier. This latter communication is logically unnecessary since the receiving worker will never use the data, but 0.2 seconds divided across 16 processors is insignificant.

DIFF

DIFF performs more communication per unit of computation than MULT, but the Munin version still performs within 5% of the message passing version (a speedup of 12.3 for Munin versus 12.8 for message passing). The reason for Munin's good performance is its use of software release consistency and the write-shared protocol. Together, these techniques result in the underlying communication pattern of the Munin version being nearly identical to that of the message passing version. When each thread first accesses a page of shared data, it reads a copy of the page. At the end of the first half-iteration, each node has a read-only copy of any pages that are shared by another thread, and a read-write copy of any pages for which it has the only copy. During the second half-iteration, during which each thread copies the new values from the scratch array to the shared array, each node makes a twin of its shared pages as it modifies them. There is no communication until the worker threads synchronize. When a thread arrives at the barrier following this half-iteration, it creates diffs of the shared pages and sends them directly to the appropriate neighbors. These diffs include all of the modified data on each shared page, not just the edge elements. Since the shared pages are still shared even after they are purged, they are write-protected again, so that subsequent writes will be detected. After all the diffs have been propagated, the thread sends a barrier message to the barrier master. For subsequent iterations, each thread generates a protection violation when it first writes to a shared page, at which time it creates a local diff. Between synchronization points, no communication occurs, and when a thread synchronizes after modifying shared data, the changes are sent directly to the other threads sharing the data. Thus, the data motion in the Munin version of DIFF is essentially identical to the message passing implementation – communication only occurs at the end of each iteration and only neighboring nodes exchange results. The only additional overhead comes from the fault handling, and from copying, encoding, and decoding the shared portions of the matrix and the modified non-edge elements.

As an aside, a curious phenomenon can be seen in Table 4.3. The Munin version of DIFF transmits less data than the message passing version. This phenomenon occurs because Munin only transmits the words that have been modified during each iteration, while the message passing version ships the entire edge row. During the early iterations, many of the edge values have not yet been modified, and thus Munin does not transmit any new values for them. In practice, the amount of data transmitted did not have much effect on the running times. Rather, Munin performs well because it transmits data only during synchronization and suffers no read misses (after the first iteration).

TSP

The two TSP experiments reinforce the importance of program granularity for achieving good performance on a DSM system. For the coarse-grained experiment, the Munin version performs within 5% of the message-passing version for sixteen processors. TSP-C is compute-bound: under 30 seconds of communication for the Munin version compared to a total execution time of 880 seconds. The performance difference between the message passing version and the Munin version comes from the cost of accessing the priority queue. In Munin, each time a thread tries to remove a tour from the queue, the queue data structure needs to be shipped to that thread. This behavior had two adverse effects on performance. First, worker threads cumulatively spent 62 seconds waiting on the task queue lock. Second, the Munin version shipped 4 megabytes of data, compared to only 900 kilobytes in the message passing version.

While the Munin version of TSP-C performs very well, the Munin version of TSP-F does not. The Munin version of TSP-F achieves a speedup of 5.9, 34% less than the message passing version's speedup of 8.9. The reasons for the reduction in performance are the same as for TSP-C, but their relative importance is increased. The amount of time required to recursively solve a partially evaluated tour in TSP-F is approximately an order of magnitude less than the time required to solve a partially evaluated tour in TSP-C. In TSP-F, worker threads spend a cumulative 360 seconds waiting for the priority queue, and a total of 210 seconds performing useful computation, resulting in a speedup of only 5.9 ($210/(210 + 360) * 16 = 5.9$). In addition, 9.2 megabytes of data are transmitted in the Munin version, compared to only 920 kilobytes for the message passing version. The granularity is sufficiently small to limit the message passing version to a speedup of 8.9.

By expanding each subtour one step deeper before recursing, TSP-F both increases the amount of sharing and reduces the problem granularity (the time it takes to independently solve a tour recursively, and thus the average time between accesses to the priority queue). Thus, deeper expansion would seem to uniformly hinder performance, since more sharing and less granularity are both undesirable, but while the speedup does drop noticeably (5.9 versus 12.6 for TSP-C), the overall running time decreases because threads tend to spend less time evaluating non-minimal tours. The reason that the speedup drops is that the running time for one processor drops more than the running time for multiple processors because of the aforementioned effects.

QSORT

The Munin version of QSORT achieves only 67% of the speedup of the message passing version. Part of the reason can be seen in Table 4.3 – the Munin version requires significantly more communication than the message passing version. Most of the performance dropoff comes from the different way that work is distributed in each version. As with TSP-C and TSP-F, the source of most of Munin's overhead is the need to ship the task queue each time a node tries to perform a queue insertion or deletion. Compounding this problem is the fact that, unlike TSP, threads do not retain sole ownership of the task queue while subdividing the array into pieces sufficiently small to solve directly. They continually re-acquire the task queue as they divide the array until it contains at most 1024 elements. Worker threads cumulatively spend 842 seconds waiting on the task queue lock (an average of over 50 seconds per worker, out of a total running time on sixteen processors of 135 seconds). Furthermore, the Munin version ships 9-megabytes of migratory data. Only the task queue is migratory.

The amount of time that each thread spends performing useful work is not sufficiently large to completely mask the cost of shipping the large (4-kilobyte) task queue to all of the other processors between each time that a thread attempts to acquire the task queue. The problem granularity used in the experiment is sufficient to support eight workers accessing the task queue before the task queue becomes the major system bottleneck. The number of workers that can perform useful work can be increased by increasing the problem granularity, if there is enough work available to do so.

Like TSP, the message passing version of the program, in which the task queue is maintained on the root node and remote worker threads acquire work by performing explicit RPCs to the root, only requires two messages plus the messages required to ship the data itself each time a thread attempts to acquire work.

FFT

The FFT algorithm exhibits a very high degree of sharing, which results in it being bus bandwidth limited to a speedup of approximately ten on a twenty processor, single-bus multiprocessor like the Sequent Symmetry [Dwa92]. Because of the way that the data is distributed, every page is referenced (and modified) by every thread during the first $\log_2 N - \log_2 P$ iterations, the worst possible behavior for a DSM system. Despite this very highly intertwined access behavior, the Munin version of FFT achieves a speedup within 5% of the message passing version (8.2 versus 8.6). The good performance of the Munin version is made possible by Munin's efficient support for multiple concurrent writers to a single shared page of data, which results in Munin performing very little unnecessary work compared to message passing. Munin only takes 2168 faults and reloads a total of 12 megabytes of data, because, while FFT shares data at a very fine grain, the computation itself is relatively coarse grained. During each of the long compute phases, Munin performs no remote communication other than to fault in data the first time it is accessed. Munin's use of delayed updates allows each worker thread to proceed completely independently during the compute phase. Only when a thread arrives at the barrier between phases of the computation must it communicate. At this point, the data is reallocated across the processors so that subsequent phases can again perform independently. As was the case for DIFF, Munin's delayed update mechanism allows it to perform most of its communication at the same points in the program at which the message passing program sends messages, the desired behavior.

The primary source of overhead for the Munin program comes from sending out the updates during the data exchange phase after the first $\log_2 N - \log_2 P$ phases. At the beginning of the update phase, every processor has a copy of every page of shared data. Thus, each processor attempts to send updates for every page to every other processor. After the update phase, each processor uses only a small contiguous portion of the shared array, so most of the updates are unnecessary. Munin's update timeout mechanism keeps the processors from actually shipping most of the data to every node. Rather, most of the replicated pages are declared stale and purged. The subset of pages that each node needs after the update phase are then reloaded from the single copy of last resort, which incorporated all the modifications. The result is that the Munin version of FFT ships only slightly more data than the message passing version (11.6 megabytes versus 9.3 megabytes).

GAUSS

The Munin version of GAUSS achieves a speedup of 8.6 on sixteen processors, 71% of the message passing version's speedup of 12.1. The reason for this reduced performance is the relatively small amount of work done per iteration, particularly during the latter stages of the algorithm when there are very few non-zero elements left upon which to operate. This fine granularity accentuates the overhead imposed by both the general purpose barrier mechanism and the overhead of shipping updates to shared data during synchronization. On the average, each thread spends over 40 seconds waiting for barriers, which includes the time spent exchanging data.

Summary

Table 4.4 summarizes the relative performance of the Munin DSM versions of the programs compared with their message passing equivalents. Four of the programs immediately achieve over 95% of the speedup achieved by their message passing equivalents for sixteen processors. These programs (MULT, DIFF, TSP-C, FFT) require the least communication per unit of computation, as long as the false sharing inherent in DIFF and FFT is handled efficiently as it is in Munin. For QSORT and TSP-F, Munin does a good job of limiting the negative impact of the false sharing between the elements in the work arrays, but performance suffers somewhat because the data shipping paradigm of DSM requires that the task queue migrate between processors as the threads acquire work to perform. The problem granularities are too fine to completely mask the cost of locating and shipping the task queue each time a thread needs to acquire new work. Like QSORT and TSP-F, the small granularity of the work in GAUSS, particularly in the latter stages of the computation, accentuates the overheads of DSM.

Programs	Performance
MULT, DIFF, TSP-C, FFT	within 5%
QSORT, TSP-F, GAUSS	within 29% – 34%

Table 4.4 Munin vs Message Passing: Performance Summary

4.5 Conventional DSM versus Munin DSM

After comparing the performance of Munin to hand-coded message passing, we compared the performance of a collection of programs running under Munin to that of equivalent programs using a conventional DSM. To represent a conventional DSM, we annotated every shared variable declaration with Munin's conventional protocol, which implemented the page-granularity write-invalidate protocol of Ivy with Mirage-like ownership freezing extensions, as described in Section 3.5.3. These experiments were designed to determine the extent to which Munin's novel features improve upon the performance of a conventional DSM, and further to determine the particular types of programs for which these features are most important. The test programs were run on from one to sixteen processors, using the same problem sizes as before (see Table 4.1).

Table 4.5 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. Figure 4.14 illustrates the relative performance

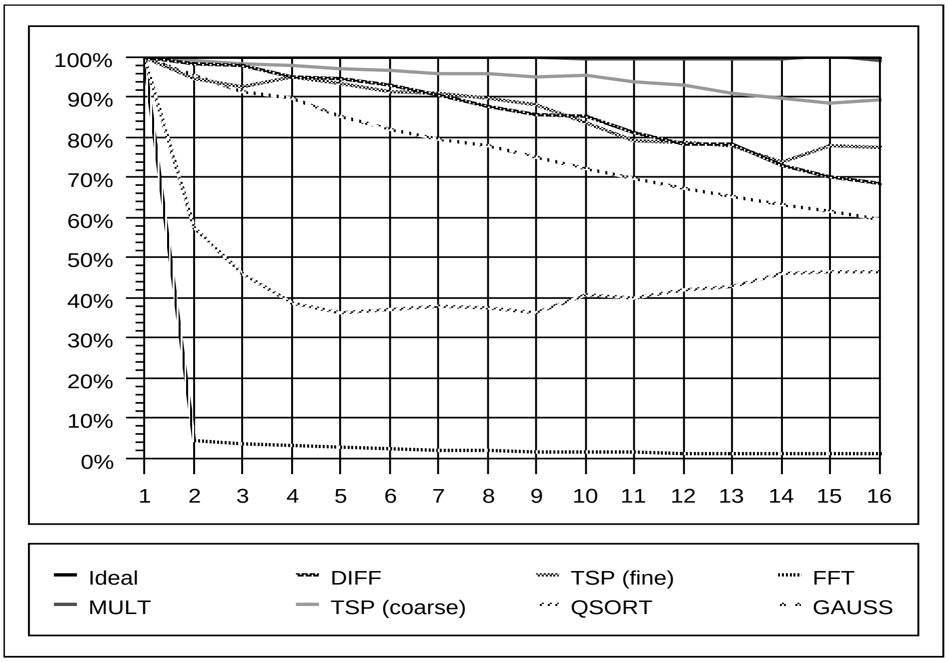

Figure 4.14 Efficiency of Conventional DSM Relative to Munin DSM

of Munin and conventional DSM. Table 4.6 presents the amount of communication required during execution of the programs on sixteen processors.

	Conventional	Munin	Percentage
MULT	14.5	14.6	99%
DIFF	8.4	12.3	68%
TSP-C	11.3	12.6	90%
TSP-F	4.7	5.9	80%
QSORT	4.1	8.9	46%
FFT	0.09	8.2	< 0.1%
GAUSS	5.1	8.6	59%

Table 4.5 Conventional DSM vs Munin DSM: Speedup (16 processors)

Program	Conventional		Munin	
	Messages	Kilobytes	Messages	Kilobytes
MULT	1490	1327	1567	1384
DIFF	35,486	26,534	14,646	3645
TSP-C	7940	4770	7870	4163
TSP-F	10,194	5963	9776	4989
QSORT	129,428	101,007	31,866	14,565
FFT	1,594,952	1,336,317	15,322	11,631
GAUSS	32349	7388	26034	5526

Table 4.6 Conventional DSM vs Munin DSM: Communication (16 processors)

Tables 4.7 through 4.13 present detailed performance statistics collected by the DSM runtime system for each of the programs when run on sixteen processors. The first set of statistics presented are the number of access faults and the number of faults that were writes. These statistics are followed by the time[10] spent sending fresh copies of data, waiting for a data request to be satisfied, invalidating remote data, purging the delayed update queue, waiting to acquire locks, and waiting at barriers. The final statistic is the number of times that the ownership freezing mechanism was invoked to slow the rate at which data ping pongs.

The following program-by-program analysis of the specific reasons for the observed difference in performance between the Munin and conventional DSM versions of each program exposes the situations in which Munin's novel features are most important.

MULT

As with the Munin version, MULT is so completely compute-bound that the overhead added by the conventional DSM runtime is insignificant. The cumulative execution time is approximately 900 seconds, while the cumulative communication time is roughly 6 seconds. The conventional DSM version ships slightly less data, but its running time is slightly higher because the root node is forced to fault in the result data sequentially at the end of the program. The Munin version ships the data to the root node immediately. This difference is reflected in the extra time spent waiting at barriers in the conventional version, as seen in Table 4.7. The slightly higher overhead of the conventional DSM is insignificant in this case, which allows the conventional DSM version of MULT to achieve 99% of the speedup of the Munin version. MULT is so completely compute-bound and has so little sharing that any DSM system should support it well.

DIFF

The conventional DSM version of DIFF achieves a speedup of only 8.4, compared to 12.3 for Munin (68% efficiency). Table 4.8 exposes some of the reasons for the relatively poor performance of the conventional DSM version. The conventional DSM version suffers from (1) frequent read faults and reloads of shared data caused by the use of a write-invalidate protocol, and (2) blocking on writes as a result of sequential consistency. The Munin version of DIFF creates and transmits *diffs* at the end of each iteration, which results in the shared data being present before it is accessed during

[10]The times presented in Tables 4.7 through 4.13 are all *cumulative times*. The cumulative time spent performing a particular operation is measured as the total time spent by all sixteen threads performing that operation.

	Munin	Conventional
Total number of faults (writes)	167 (73)	162 (74)
Cumulative time sending copies (waiting)	4.3 (6.7)	4.4 (7.3)
Cumulative time spent invalidating	0.0	0.9
Cumulative time purging DUQ	0.0	—
Cumulative time waiting for locks	0.0	0.0
Cumulative time waiting at barriers	42.5	54.4
Number of ownership freezes	—	0

Table 4.7 Execution Statistics for Matrix Multiplication

the next iteration. Updating rather than invalidating the shared data eliminates read misses and reloads on the next iteration, which reduces the number of read misses for Munin to 270 compared to 3239 for the conventional DSM. Furthermore, because threads experience faults at different times during the execution of each iteration and thus skew the barrier arrival times, the conventional DSM version spends twice as long waiting at barriers. In addition, write faults can be handled without communication in Munin if the data is already present, which is the case for all but the first iteration. The local node simply makes a twin of the data. The conventional DSM implementation sends an invalidation message and waits for a response. The tradeoff is that synchronization under Munin is slowed down because memory needs to be made consistent before the synchronization operation can complete. However, the total time that the Munin worker threads spends blocked while waiting for memory to be made consistent (71.5 seconds) is far less than the time spent invalidating and reloading the data (a total of 356.1 seconds) in the conventional version. The time spent invalidating and reloading seriously impacts execution time (356.1 seconds of a total execution time of 662.1 seconds).

TSP

The difference in performance between the Munin and conventional DSM versions of TSP-C (a speedup of 12.6 for Munin versus 11.3 for conventional DSM) stems from (1) the use of a migratory protocol for the task queue, and (2) the use of an update, instead of an invalidate, protocol for the minimum tour length. The slightly higher

	Munin	Conventional
Total number of faults (writes)	3270 (3000)	6330 (3091)
Cumulative time sending copies (waiting)	32.0 (36.2)	181.1 (337.8)
Cumulative time spent invalidating	0.0	18.3
Cumulative time spent purging DUQ	71.5	—
Cumulative time waiting for locks	0.0	0.0
Cumulative time waiting at barriers	85.6	169.5
Number of ownership freezes	—	1472

Table 4.8 Execution Statistics for Finite Differencing

overhead caused by loading and invalidating, rather than simply migrating, the task queue has the effect of causing more processors to idle themselves while waiting to acquire work from the task queue. The number of idle processors increases because access to the task queue is the primary bottleneck (a total of 94.2 seconds for the conventional version versus only 62.2 in the Munin version). In TSP-F, this increased overhead of accessing the task queue limits the speedup to 4.7, versus 5.9 for Munin and 8.9 for message passing.

The minimum tour length is an example of a shared data item for which an update protocol is better than an invalidate protocol, because it is read much more frequently than it is written. With the conventional protocol, a thread that needs to update the minimum tour length will typically send $N-1$ invalidations and then wait for $N-1$ acknowledgements. The cost of sending the $N-1$ invalidation messages, each of which causes a user thread to fault and idle waiting for a read request to be handled, is considerably higher than the cost of sending $N-1$ update messages and incorporating them on the $N-1$ other nodes. The large number of threads idled while the data is reloaded (on average $N-1$) causes the TSP-F speedups to drop off as the number of processors increased.

An interesting phenomenon occurred before Mirage's ownership freezing mechanism was introduced. On more than six processors, the running time of the conventional version of TSP-F suddenly increased by several orders of magnitude. The reason for this increase in running time was that the global minimum was being paged around almost constantly, so much so that very little progress was being made.

	Munin	Conventional
Total number of faults (writes)	946 (289)	1712 (627)
Cumulative time sending copies (waiting)	7.8 (24.3)	8.3 (77.7)
Cumulative time spent invalidating	0.0	2.8
Cumulative time spent purging DUQ	10.8	—
Cumulative time waiting for locks	62.2	94.2
Cumulative time waiting at barriers	82.5	77.8
Number of ownership freezes	—	52

Table 4.9 Execution Statistics for the Coarse-grained TSP

	Munin	Conventional
Total number of faults (writes)	1109 (314)	1787 (680)
Cumulative time sending copies (waiting)	9.6 (18.9)	10.0 (66.0)
Cumulative time spent invalidating	0.0	2.7
Cumulative time spent purging DUQ	10.8	—
Cumulative time waiting for locks	357.5	501.6
Cumulative time waiting at barriers	15.9	16.0
Number of ownership freezes	—	55

Table 4.10 Execution Statistics for the Fine-grained TSP

The problem was caused by the phenomenon discussed in Section 3.5.3 wherein non-exclusive readers interfere with a single writer's ability to acquire exclusive ownership. This problem led to the introduction of Mirage's ownership freezing mechanism.

QSORT

The conventional DSM version of QSORT achieves a speedup of only 4.1, 46% of the 8.9 speedup of the Munin version. In addition to the cost of invalidating and reloading the task queue, rather than simply migrating it, the difference in performance between the conventional DSM and the Munin version is primarily due to the presence of false sharing when two threads simultaneously attempt to sort subarrays that reside on the same page. The array in QSORT is partitioned dynamically, which introduces a large amount of false sharing because the data is partitioned with no regard for page boundaries. It is common for two threads to be concurrently sorting subarrays that resided on the same page, which causes them to page the data back and forth. As a result, communication increases from 23 megabytes in about 30,000 messages, for the Munin version, to 110 megabytes in 231,000 messages for the conventional version. Furthermore, the conventional DSM version suffers almost an order of magnitude more faults (24,807 versus 5122 for the Munin version), and spends over an order of magnitude more time transmitting data (2434.7 seconds, or roughly 150 seconds per node, versus 96.0 seconds for the Munin version).

Another indication of the heavy amount of false sharing is the large number of ownership freezes performed by the conventional implementation: 11,682. The ownership freezing mechanism helps alleviate the problem of page bouncing by allowing a thread to get some useful work accomplished after acquiring ownership, but it also results in the invalidated thread being idled during the freeze period. Only 199.4 seconds out of the 2434.7 seconds that threads spend waiting for data are spent actually transmitting the data. Once the freeze period expires, both threads are able to execute only until one of them attempts to write to the shared page, at which time the process begins anew. The ownership freeze mechanism reduces the frequency of page bouncing and lowers the execution time from essentially infinity to several hundred seconds, but it does not solve the problem, as illustrated by the conventional DSM's poor performance for QSORT.

	Munin	Conventional
Total number of faults (writes)	5122 (2724)	24,807 (14,184)
Cumulative time sending copies (waiting)	16.7 (96.0)	194.4 (2434.7)
Cumulative time spent invalidating	0.0	46.6
Cumulative time spent purging DUQ	46.3	—
Cumulative time waiting for locks	857.0	268.9
Cumulative time waiting at barriers	14.3	4.6
Number of ownership freezes	—	11,682

Table 4.11 Execution Statistics for Quicksort

FFT

The very high degree of false sharing in FFT causes the conventional DSM version to *slow down* by a factor of ten to twelve for two or more processors, while the Munin version manages a speedup of 8.2 on sixteen processors. The cause for this dramatic difference in performance is Munin's ability to efficiently support multiple concurrent writers to a shared block of data. The conventional DSM version of FFT is plagued by continual page bouncing, despite the introduction of Mirage's page freezing mechanism. Each thread spends most of its time waiting for data, only able to perform useful computation during the relatively small periods during which the page was frozen. The conventional DSM implementation takes over 300,000 faults, requires 1.35 gigabytes of data to be shipped and 1.65 million messages to be transmitted, and cumulatively spends over 25,000 seconds waiting for requests to be satisfied. The sequential execution time is only 224 seconds, so it is clear that a conventional implementation of DSM is inappropriate for this style of program. Munin's write-sharing mechanism, on the other hand, is able to overcome much of the problem by allowing multiple threads to modify independent portions of the shared array simultaneously and only exchange results at the end of each phase of the computation. While not devoid of overhead, the Munin version requires orders of magnitude less communication. It only takes 2168 faults and reloads a total of 12 megabytes of data.

	Munin	Conventional
Total number of faults (reads, writes)	2169 (1088)	291,229 (135,608)
Cumulative time sending copies (waiting)	45.7 (130.9)	4998.7 (31,094)
Cumulative time spent invalidating	0.00	740.6
Cumulative time spent purging DUQ	80.4	—
Cumulative time waiting for locks	0.0	0.0
Cumulative time waiting at barriers	35.8	5210.9
Number of ownership freezes	—	32,582

Table 4.12 Execution Statistics for FFT

GAUSS

The conventional DSM version of GAUSS achieves a speedup of 5.1 on sixteen processors, 42% of the message passing version and 59% of the Munin version. In addition to the synchronization issues noted in the Munin implementation, the conventional DSM implementation also suffers from frequent read misses caused by accesses to invalidated data, as seen in Table 4.13. While the Munin implementation experiences 90 read misses, the conventional DSM implementation experiences 6780. This high read miss rate is caused by the use of an invalidation-based consistency protocol in the conventional DSM system. Since all of the modifications are made to shared data that is being actively shared (and constantly used) on all sixteen processors, the update-pruning advantage of an invalidation protocol is not relevant, while the increased number of read misses is a significant problem. Each thread in the conventional version stalls for an average of 50 seconds waiting for read misses to be serviced. In addition, because the last thread to have its read miss satisfied must wait until fourteen other threads have successfully acquired their data, the computations tend to complete at noticeably different times. This skewing of the thread completion times causes the average time spent waiting at barriers to increase from 30 seconds to 50 seconds. These two phenomena explain the lower performance of the conventional DSM implementation.

	Munin	Conventional
Total number of faults (writes)	602 (512)	8076 (1296)
Cumulative time sending copies (waiting)	2.03 (6.14)	63.2 (821.1)
Cumulative time spent invalidating	0.00	24.91
Cumulative time spent purging DUQ	26.7	—
Cumulative time waiting for locks	0.00	0.00
Cumulative time waiting at barriers	484.0	882.5
Number of ownership freezes	0	597

Table 4.13 Execution Statistics for Gaussian Elimination

Summary

Table 4.14 summarizes the relative performance of the conventional DSM versions of the programs compared with their Munin equivalents. For the programs with large grained sharing that did not tend to have more than one thread write-sharing a single page (MULT and TSP-C), the conventional versions achieved approximately 90% of the speedup of their Munin counterparts. For DIFF, TSP-F, and GAUSS, taking unnecessary read misses along the critical path hurt the performance of the conventional DSM programs. Finally, for the programs with very fine-grained sharing and for which it was common to have multiple threads reading and writing the same page of memory, the conventional versions performed quite poorly. The conventional DSM implementation of QSORT achieved less than half of the speedup of the Munin DSM version, while FFT performed orders of magnitude worse for conventional DSM than Munin DSM. The poor performance of the conventional DSM version was caused by its inability to handle false sharing efficiently, which results in frequent page bouncing and read misses. A somewhat unexpected result was the extent to which Mirage's page freezing mechanism was able to improve the performance of a conventional DSM in situations when there was a moderate amount of false sharing and trading off parallelism to avoid page bouncing was worthwhile.

4.6 Effect of Communication Reduction Techniques

In this section we try to isolate the effects on performance of each of the techniques for reducing communication that were described in Chapter 2. This isolation is made somewhat difficult because of the synergistic effect on performance of using the techniques in conjunction with one another. In particular, write-shared protocols cannot

Programs	Performance
MULT, TSP-C	within 10%
DIFF, TSP-F, GAUSS	within 20% – 50%
QSORT	under 50%
FFT	unacceptable

Table 4.14 Conventional vs Munin: Performance Summary

be used in the absence of release consistency. Therefore, we first compare Munin (including buffered release consistency and write-shared protocols) to a DASH-like system (with pipelined release consistency and exclusive-writer invalidate protocols). Then we compare the use of multiple protocols versus using a single protocol, write-shared. Finally, we determine the value of the update timeout mechanism in connection with the update protocol.

4.6.1 Munin versus a DASH-like System

We created a DSM version that implemented "DASH-like" release consistency. On a write fault, ownership is transferred to the faulting processor. Then write-invalidates are sent out in separate messages, allowing multiple invalidations to be concurrently outstanding, but ensuring that no synchronization operation could succeed until all outstanding invalidations were completed. We compared the performance of this version with the Munin and conventional DSM versions. For MULT, TSP-C, TSP-F, and GAUSS there was little difference between the DASH-like version and the Munin version. For SOR and QSORT the difference in speedup was 30% for 16 processors. For FFT the Munin version performed orders of magnitude better than the DASH-like version. For the latter three applications, the DASH-like version performs slightly better than the conventional DSM version. Figures 4.15 and 4.16 depict these results for SOR and FFT. The performance of QSORT is similar to that of SOR.

These results demonstrate that a DASH-like implementation of release consistency offers some performance gain over a conventional DSM implementation, because useful computation can be overlapped with invalidations. This overlap reduces the cost of writes, but does nothing to reduce the penalty associated with read misses. Furthermore, the DASH-like version suffers from false sharing, much in the same way that a conventional DSM system does. When read misses dominate, or when there is substantial false sharing, Munin's implementation is superior.

4.6.2 Multiple Consistency Protocols

The observation that no single consistency protocol is best-suited for all programs was discussed in Section 2.2. To evaluate the importance of Munin's support for multiple consistency protocols, we compared the performance of two versions of Munin: (i) a version in which multiple consistency protocols were used, and (ii) a version that labeled all shared data to be write-shared, thus employing Munin's most versatile

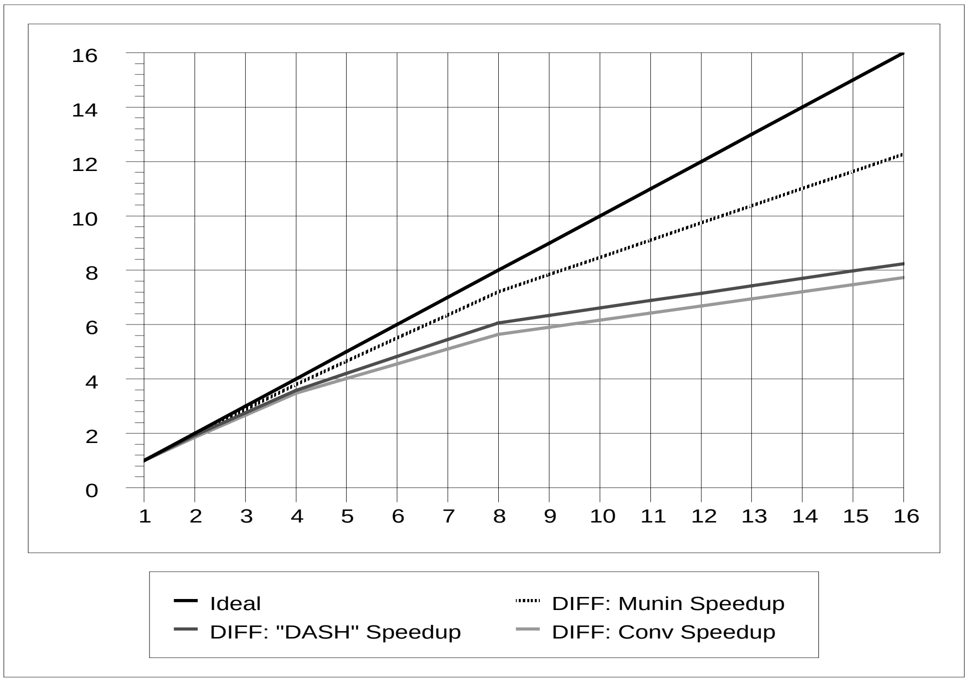

Figure 4.15 Munin RC versus “DASH” RC (DIFF)

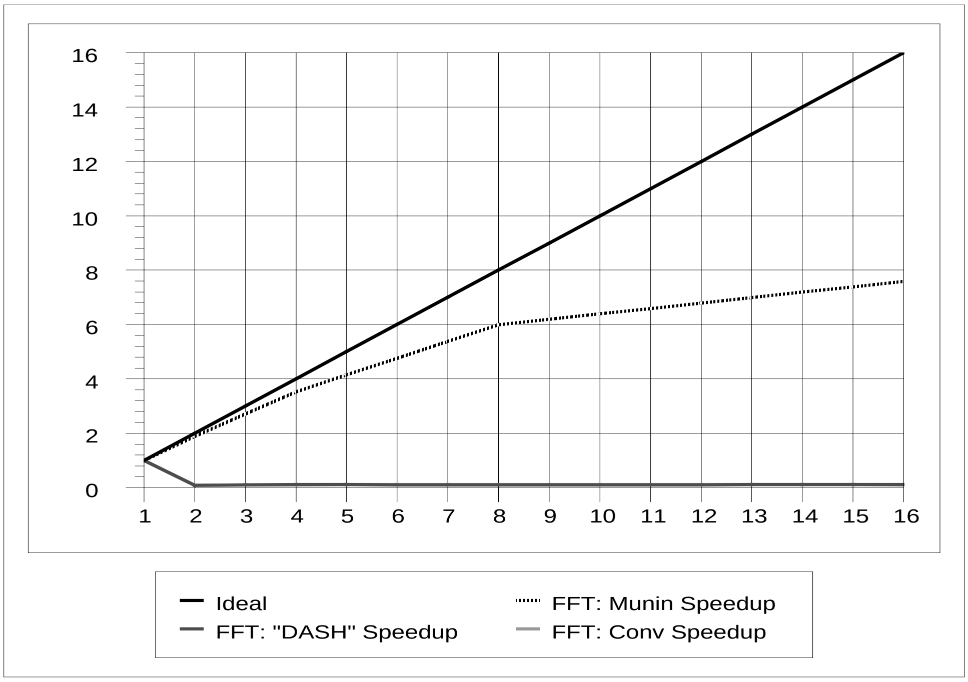

Figure 4.16 Munin RC versus “DASH” RC (FFT)

protocol. Figure 4.17 presents the results of this experiment for TSP-F; similar results were obtained for the other multiprotocol test programs (TSP-C and QSORT). For TSP-F, using multiple protocols and declaring the task queue to be migratory leads to a 30% improvement in speedup for the 16 processors. The task queue is accessed by only one thread at a time, so it is more efficient to transmit the eight-kilobyte queue in its entirety each time work needs to be allocated than it is to fault, create a *diff*, and transmit the *diff* to multiple nodes. Doing so reduces the amount of time spent in the protocol handler copying and "diffing", without increasing the amount of time spent communicating. While the *diff* is generally small (less than one packet), it is more efficient to send one large block of data in a single burst than it is to send a number of small packets, which explains why the performance difference is more noticeable as the number of processors, and thus the number of updates that must be sent, increases. The update timeout mechanism tends to mitigate this effect, but almost any work performed by the update and update timeout mechanisms is more than what is required to simply migrate the task queue between processors.

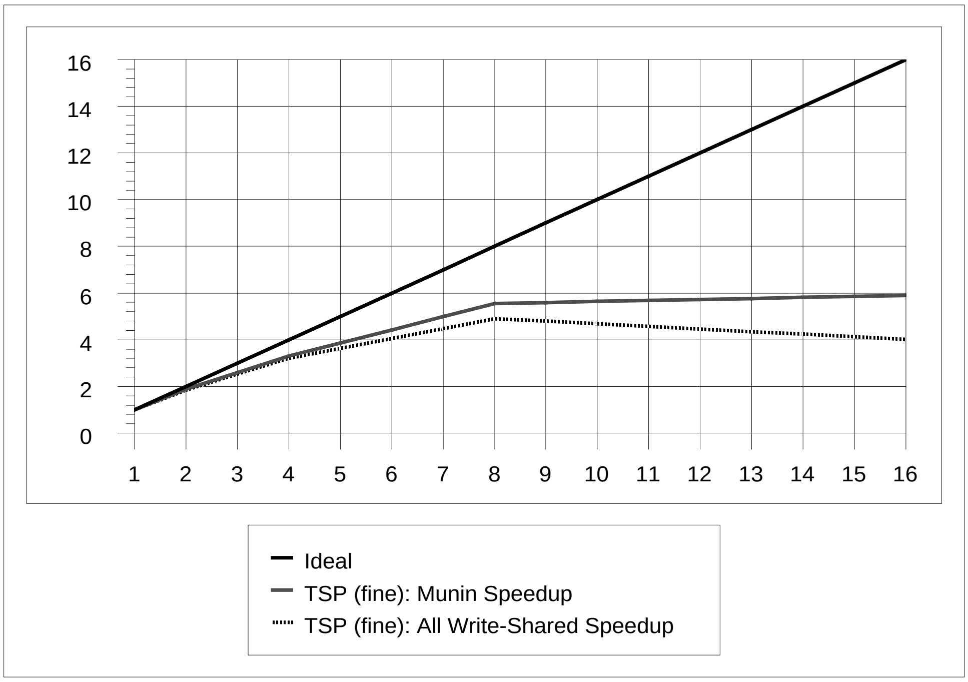

Figure 4.17 Multi-protocol versus All Write-Shared (TSP-F)

4.6.3 Update Timeout Mechanism

To test the value of the timeout mechanism in connection with the update protocol, we compared the performance of versions with and without the timeout enabled. For MULT, SOR, and TSP-C there was no difference. For TSP-F and QSORT the version with the timeout enabled was 10% and 15% faster for 16 processors, respectively. The difference was the largest for FFT. Speedup with 16 processors dropped from 7.6 to 3.6 when the timeout was disabled, as illustrated in Figure 4.18. Finally, for GAUSS, the timeout actually caused a 5% dropoff in performance for 16 processors.

In FFT, after each iteration of the parallel butterfly operation, each thread needs to update the replicas of the shared pages that it has just modified on some number of remote processors. When the butterflies are widest (at the beginning of the computation), every processor needs to read and write data on every page of shared data, so every processor contains a copy of every shared page. As the computation continues, each processor's working set changes dynamically, the result of which is that processors do not need to receive updates for all of the pages. Without the timeout mechanism, every processor will continue to receive updates for every page

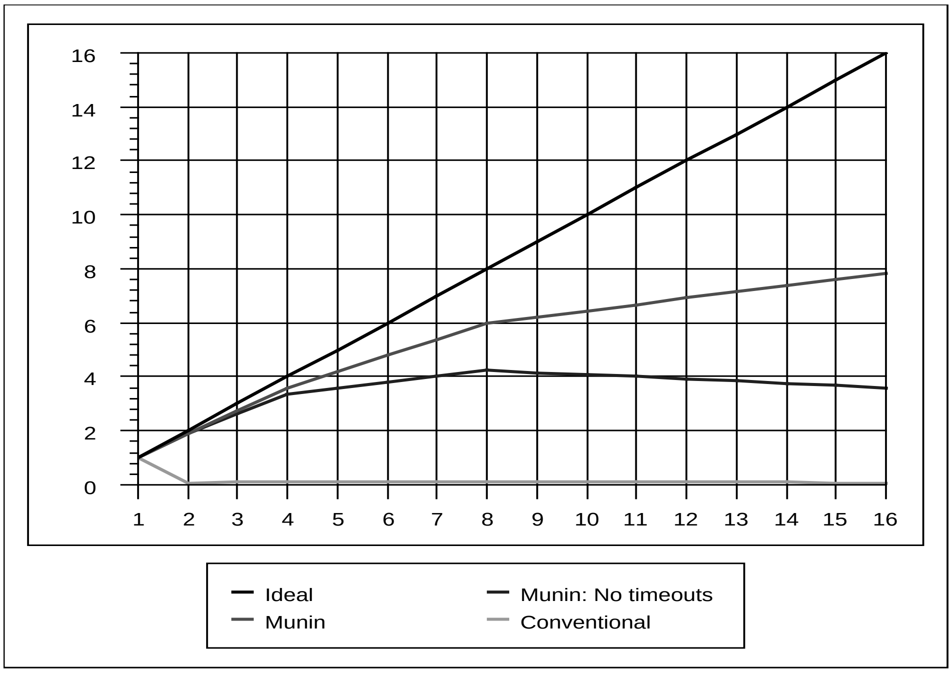

Figure 4.18 Effect of Update Timeout Mechanism on FFT

throughout the rest of the computation, despite the fact a given node might not use a particular page for several iterations.

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

FFT was not the only program for which the timeout mechanism improved performance. The other two programs in which each processor's working set changed dynamically over the course of the program execution, TSP and QSORT, were also aided by the use of the timeout mechanism. For TSP, each page of the shared tour array tended to be used by many different processors over time, but each processor only used it for a very short period of time, and only a few processors used a particular page at a time. Without the timeout mechanism, eventually almost every processor receives updates for almost every page. The shared sort array in QSORT exhibited a similar phenomenon.

With GAUSS all of the modified data are accessed every iteration. The slight dropoff in performance caused by the update timeout mechanism is caused by the fact that the default update timeout time of 50 milliseconds was too short to ensure that no updates were timed out. When the timeout mechanism was disabled, the sixteen-processor speedup of the Munin version improved slightly from 8.6 to 8.9, 74% of message passing.

4.7 Function Shipping

For TSP-F and QSORT, the two programs that used the task queue model of parallelism and that had a significant amount of sharing, the Munin sixteen processor versions achieved speedups of only 6.0 and 8.9, respectively, compared to 10.6 and 13.4 for the message passing versions. The conventional DSM versions performed even worse, achieving speedups of 4.7 and 4.1, respectively. As shown in Table 4.15, the major source of overhead for these DSM versions (with the exception of the conventional version of QSORT) was the amount of time spent waiting on the lock protecting the work queues.

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

Table 4.15 Lock waiting times for TSP-F and QSORT

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

In order to evaluate the feasibility and potential value of using a mixed data-shipping and function-shipping mechanism in a DSM system, we modified the DSM versions of TSP-F and QSORT such that the task queue remained attached to the root node, and all access to the task queue by other nodes was performed using RPC. This required the introduction of a `Flush()` operation to allow the delayed update queue to be flushed manually before each RPC. Not doing so can lead to inconsistencies, because an RPC inherently synchronizes the processes involved. The results of function-shipping access to the task queue for the TSP-F and QSORT are shown in Figures 4.19 and 4.20. These figures show the speedups achieved by Munin and conventional DSM both with and without function shipping for the task queue.

For TSP-F, function shipping causes both DSM versions to perform almost as well as the message passing version (on 16 processors, a speedup of 9.1 for conventional DSM, 9.8 for Munin, and 10.6 for message passing). In contrast, without function shipping, Munin achieved a speedup of only 6.0, and the conventional DSM a speedup of only 4.7. For the Munin version without function shipping, communication was substantially more (9229 messages and 4989 kilobytes of data) than the Munin version with function shipping (3630 messages and 888 kilobytes of data). Perhaps more

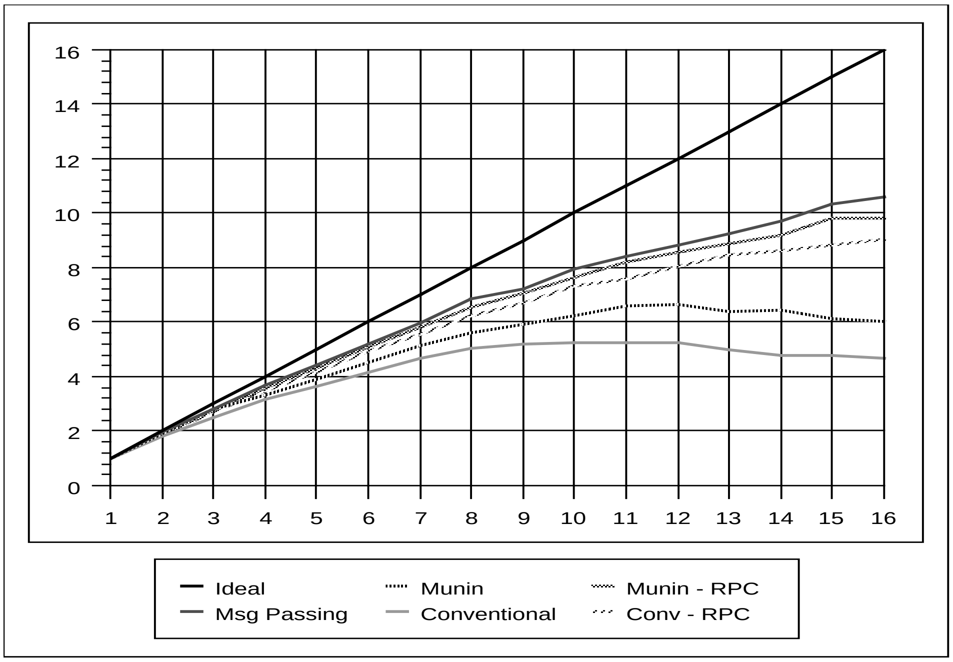

Figure 4.19 Effect of Function Shipping on Fine-grained TSP

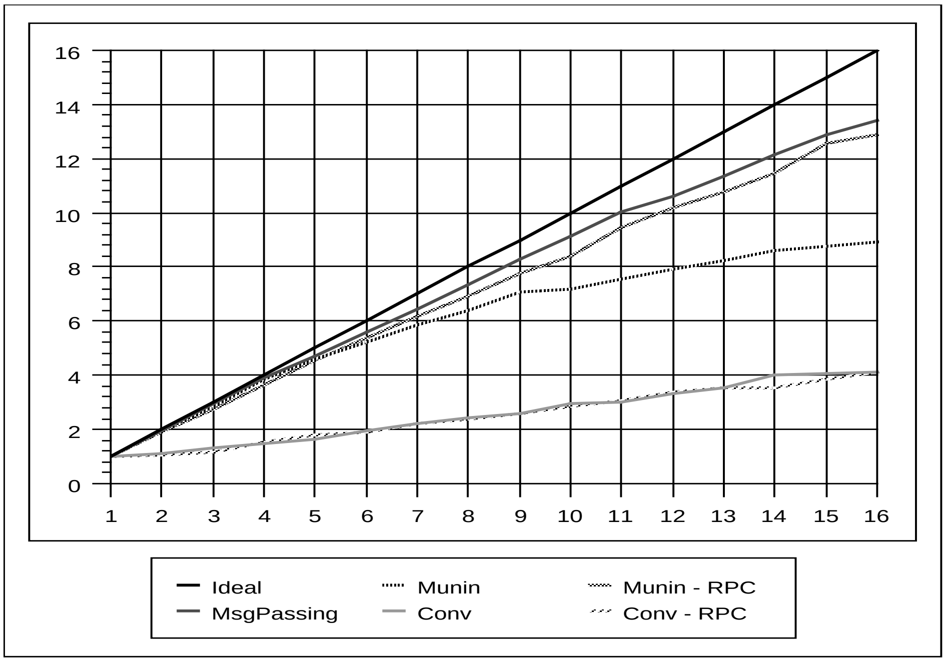

Figure 4.20 Effect of Function Shipping on Quicksort

importantly, the reduced communication of the function shipping version nearly eliminated the time that threads were idle waiting for access to the task queue.

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

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

4.8 Summary

The results presented in this chapter are very promising, and argue that the novel mechanisms introduced in Munin represent 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, when the sharing is at a large granularity, and when there are no concurrent writers to a single page of shared data. 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. In addition to highlighting the importance of supporting write-shared data in a DSM, the results also indicate that Munin's support for multiple consistency protocols, its use of delayed updates, and its update timeout mechanism can each improve DSM performance. Two other notable results were the importance of using a Mirage-like ownership freeze mechanism with write-invalidate protocols and the potential performance benefits of supporting function shipping in addition to data shipping.

Chapter 5

Related Work

This chapter compares Munin with a number of existing software and hardware DSM systems, giving special emphasis to the difference in design strategies and the tradeoffs that result.

Software DSMs

Ivy was the first DSM system [LH89]. It uses a single-writer, write-invalidate protocol for all data, with virtual memory pages as the units of consistency. The large size of the consistency unit makes the system prone to false sharing. The single-writer nature of the protocol can cause 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. However, despite these problems, Ivy demonstrated that software DSM was a viable alternative programming model to message passing for programs that did not exhibit much false sharing and for which the page granularity of the system matched the granularity at which the shared data was accessed. Several directory management protocols were tested. It was shown that a *dynamic, distributed* scheme, in which the owner of a page of data was allowed to change dynamically as it was accessed, improved performance by exploiting the temporal and spatial locality commonly found in applications. Thus, Munin uses primarily dynamic, distributed protocols.

Clouds performs consistency management on a per-object basis, or in Clouds terminology, on a per-segment basis [DCM+90]. Supporting consistency on a per-object basis gives the user somewhat finer control over the way in which the system maintains consistency. By allocating segments in such a way that they do not overlap, the user can attack the false sharing problem. In addition, Clouds allows a segment to be locked on a processor, which can help avoid the ping pong effects that may result when a single object is accessed by a large number of processors. If the programmer knows that a particular operation will use a given segment for some period of time, the programmer can lock the segment to the local processor throughout that operation. Remote threads attempting to access the segment will not interfere with the local operation and will not impede performance by causing needless migrations.

Mirage attempts to avoid the deleterious effects of false sharing by locking a page on a processor for a certain Δ-time window after it is modified by that processor [FP89]. By doing so, Mirage ensures that at least one thread will make progress over each Δ-time window, which limits the worst case performance. Without this feature, it is possible that a particular page of data could ping pong between processors with very little useful work being accomplished between each movement. This problem was apparent in our first implementation of the conventional protocol, so a similar mechanism was added to Munin. However, Munin's support for multiple concurrent writers does a better job of mitigating the adverse effects of false sharing, without introducing the delays caused by locking a segment to a processor.

Orca is an object-oriented DSM system that supports the *shared data object model* [BT88, Kaa92]. Its consistency management is based on an efficient, reliable, ordered broadcast protocol. For reasons of scalability, Munin does not rely on broadcast, although support for efficient multicast could improve the performance of some aspects of Munin. The Orca language is restricted so that (i) all access to objects is through well-defined per-object operations, (ii) only one operation on an object can be performed at a time, and (iii) there are no global variables or pointers. This programming model allows the compiler to detect all accesses to an object directly without the use of page faults. Programmers must, however, structure their programs so that objects are accessed in a way that does not limit performance. For example, an Orca implementation of SOR requires that the edge elements be specified as shared buffers - the entire array should not be declared to be a single object. However, once a program has been structured appropriately, Orca is able to perform a number of powerful optimizations that improve performance by selecting the appropriate communication style for each operation. For example, Orca can choose whether to replicate an object or force all accesses to be made via RPCs to a master node. If it chooses to replicate an object, it supports both invalidate and update consistency protocols. It remains to be seen how well Orca's optimizations can be integrated into a less restrictive language.

Emerald [BHJL86, BHJ+87, JLHB88] is a distributed, object-oriented system and language. It provides a uniform object model and explicit support for fine-grained object mobility. Unlike the systems discussed above, in Emerald the programmer is responsible for the distribution of data among processors. Objects in Emerald do not migrate except when the programmer specifies that they should, so remote accesses are performed via RPC. Emerald resolves the performance disadvantages of a uniform object model by having the compiler generate code for an object based on its expected use. Objects that are expected to interact locally communicate via shared memory, while objects that are expected to communicate with remote nodes use an RPC interface. Fine-grained mobility allows the programmer to move data from one node to another without having to package the data into a message.

Amber [CAL$^+$89] was derived from Emerald. It provides a shared address space, which alleviates some of the addressing difficulties present in Emerald. It does not attempt to automatically move or replicate data. It is difficult to assess Amber's efficiency, because very few programs have been written using it. Like Orca and Emerald, Amber maintains consistency at the object level, which requires the programmer to carefully decompose objects to reduce contention and thus allow good performance. Good speedups are reported for SOR, although at the expense of requiring the programmer to restructure the SOR program to fit the Amber model. Munin automates many aspects of data distribution, yet remains efficient by asking the programmer to specify the expected access patterns for shared data variables.

Mether [MF89, MF90] is a DSM system that runs on SUN workstations running the SunOS 4.0 operating system. Mether supports a number of special shared memory segments in fixed locations in the virtual address space of each workstation in the system. Any program that is running on one of these machines can map in an arbitrary piece of the Mether address space and use it to communicate with other programs in the system. Unlike Munin, Mether supports inter-program sharing, and not just intra-program sharing. The tradeoff is that each individual data exchange in Mether is relatively expensive (up to 70 msecs) because of its generality. In an attempt to support efficient memory-based spinlocks, Mether supports several different shared memory segments, each with different protocol characteristics. Two segments are for small objects (up to 32 bytes), while two are for large objects (up to 8192 bytes). One of each pair is "demand-driven", which means that the memory is shipped when it is read, as in a conventional DSM. The other is "data-driven", which means that it is shipped when it is written. A thread that attempts to read it will block until the next thread writes it. This latter form of data can support spinlocks and message-passing fairly effectively. However, Munin's multiple protocols are more closely aligned with the way that shared data is used, and Munin's separate synchronization package removes the need to support data-driven memory.

Mermaid is an Ivy-like DSM system that supports computations that span a heterogeneous computing environment [ZSM90]. Like Munin, Mermaid is implemented as a set of user-level library routines that are linked into the user program. Mermaid maintains sufficient type information about the data that resides in each page of shared memory to perform on-the-fly data conversions when a page is transferred between two machines with different underlying data formats. Their measurements indicate that support for heterogeneity is not only feasible, but can be made almost as efficient as homogeneous DSM.

Memo [KCZ92, DKCZ93] goes a step beyond Munin in its ability to exploit the relaxed rules provided by release consistency. Like Munin, Memo can support update-based, invalidate-based, and hybrid consistency protocols. The primary difference between Munin and Memo is that while Munin updates every cached copy of a data item whenever the modifying thread performs a release operation, Memo exploits the fact that only the thread that next acquires the released lock must be updated. Restricting the number of updates performed reduces the amount of communication performed by the DSM runtime system in the cases when there is a high degree of locality of reference. One potential performance problem that arises in Memo is that when a thread that has a stale copy of a data item needs to obtain an up-to-date copy of the data, it must obtain all of the "diffs" that have been created on any machine since it last received an update. Obtaining the appropriate "diffs" can entail querying a number of nodes before obtaining a fresh copy of the data. Another tradeoff is that the Memo runtime system uses far more memory than Munin, because it must retain "diffs" until it knows that no other node in the system will need them. This need to retain "diffs" introduces a garbage collection problem. Munin's support of the migratory protocol and the update timeout mechanism are simple heuristics that attempt to reduce the number of unnecessary updates performed. Although simulation studies indicate that Memo can reduce the number of message required to maintain consistency when a portion of the synchronization is via locks[11], it is not clear how deeply the added memory and protocol overhead of Memo will cut into this potential advantage.

[11]If synchronization is accomplished exclusively through barriers, Memo acts like Munin because of the global nature of barriers.

Midway [BZS93] proposes a DSM system with *entry consistency*, a memory consistency model weaker than release consistency. The goal of Midway is to minimize communication costs by aggressively exploiting the relationship between shared variables and the synchronization objects that protect them. While Munin supports a suite of consistency protocols, Midway supports a suite of consistency models, ranging from sequential consistency and release consistency, described in Section 2.1, to *entry consistency*, which only guarantees the consistency of a data item when the lock associated with it is acquired. In general, it is not clear how well entry consistency works, nor how hard it is to program. To exploit the power of entry consistency, the programmer must associate each minimal unit of data with a single lock. For some programs, making this association is easy. However, for programs that use nested data structures or arrays, it is not clear if making a one-to-one association is feasible without forcing programmers to completely rewrite their programs. For example, like Amber, the programmer of an entry consistent SOR program would have to hand decompose the shared array to exploit the power of entry consistency. The designers of Midway recognized this problem and gave programmers the ability to increase and decrease the strength of the consistency model supported. Thus, programs for which the data-synchronization association required by entry consistency is convenient can exploit its flexibility, while programs for which this association is inconvenient can use either release consistency (when adequate synchronization is performed) or sequential consistency. Unlike Munin, Midway exploits the power of a sophisticated compiler. The Midway compiler inserts code around data accesses so that the Midway runtime system can determine whether a particular shared variable is present before it is accessed. Thus, Midway is able to detect access violations without taking page faults, which eliminates the time spent handling interrupts.

Hardware DSMs

Recently, a number of designs for hardware distributed shared memory machines have been published [ALKK90, BDGS92, BFKR92, DSF88, LLG+90, WL92, WHL92].

Munin is most related to the DASH project [LLG+90], from which it adopted the concept of release consistency. Unlike Munin, 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 concurrent writers to a single shared object and by using update protocols and migration when appropriate. The differences between DASH's implementation of release consistency and Munin's implementation of release consistency was explained in detail in Section 2.1, and the effect on performance is detailed in Section 4.6.1.

The Willow multiprocessor [BDGS92] is a scalable shared memory architecture designed to support over a thousand commercial microprocessors. It attacks the bottlenecks often found in large scale shared-memory multiprocessor systems (inefficient synchronization, memory latency and bandwidth limitations, bus contention, cache inclusion, and limited I/O bandwidth) via several major architectural innovations. It provides a hierarchical memory, cache, and synchronization design that exploits program locality at every level in the hierarchy. One of Willow's innovative features, support for adaptive cache coherence, was derived from our preliminary results with Munin. A companion memory access evaluation tool, Paraview [Spe93, SFGB93], is used to identify the correct protocol to use for each cache line based on simulating the performance of the program using address traces. The preliminary performance of Willow indicates that the ideas developed in the Munin project will translate effectively to hardware implementation.

The GalacticaNet scalable DSM architecture introduced a novel technique for maintaining the consistency of shared data caches based on a hardware-supported but software-controlled update- and invalidate-based protocols [WL92]. Like Munin, the GalacticaNet system showed that support for an update-based protocol that exploits the flexibility of a relaxed consistency protocol can improve the performance of a scalable distributed shared memory system by reducing the effect of performance inhibiting read misses (and processor stalls). Also like Munin, the GalacticaNet design includes a provision to time out updates to stale data, which is shown to have a significant effect on the performance of the system when there is a large number of processors.

The APRIL machine addresses the problem of high latencies in distributed shared memory multiprocessors in a different way [ALKK90]. APRIL provides sequential consistency, but relies on extremely fast processor switching to overlap memory latency with computation. For APRIL to be successful at reducing the impact of read misses, there must be several threads ready to run on each processor at a time, which is not always feasible or which may cause the program granularity to become so small that it affects performance. Because APRIL performs many low-level consistency operations in very fast trap handling software, it would be possible to adopt several of Munin's optimizations to their hardware cache consistency mechanism.

Sesame is a fast hardware network interface that supports distributed shared memory on a network of high-speed workstations [WHL92]. Like Munin, Sesame aggressively attacks the problem of communication latency caused by demand-driven data transfers by selectively sending updates to shared data in advance of when they are requested, so-called *eager sharing*. Their results indicate that eager sharing allows a hardware DSM system to scale well under circumstances that limit conventional demand-driven consistency protocols to an order of magnitude less performance.

MemNet [DSF88] is a memory-mapped network interface that supports a contiguous 2-megabyte region of memory that is shared by the processors connected to its high-speed token ring network. Sharing is supported by dedicated memory management hardware built into the network interface. Any process can map a portion of the MemNet address space into its address space and use this memory to share data with any other process in the system. When a processor attempts to access a piece of data not present in the local network interface, the MemNet controller uses a snoopy cache algorithm to obtain the data from a remote interface that is caching it. MemNet was one of the first hardware implementations of shared memory for workstations connected via a network, and performed well for many programs. However, it could not readily support programs that shared data at a fine-grained level, because it could not tolerate hotspots. When there was much contention for a single 32-byte page, the network saturated and the processors spent most of their time idle. In addition, MemNet required special network interfaces, which were not widely available, so its impact was limited. However, as the first system of its kind, it helped later designers develop more scalable systems.

The Data Diffusion Machine (DDM) [WH88] was the first hardware DSM to support an *all-cache* model for memory, wherein the entire memory of the system is treated as a large cache of the global virtual address space. Preliminary results suggest that the all-cache architecture model can scale effectively for several classes of programs, but the system has not been adequately tested to determine the extent of its scalability or applicability.

The PLUS multiprocessor uses specialized hardware to support word-granularity updates to replicated pages. A distributed hardware directory is used to maintain a linked list of the processors with a copy of each replicated page. A directory entry consists of a *head* and *next* field. Each time a processor writes to a replicated page, it sends an update message to the *head* of the list of processors with a copy of the page, which in turn forwards the update to the processor represented by the *next* field.

After each processor on the chain incorporates the update, it forwards the update using the *next* field. This scheme guarantees that all updates to a given word are seen in the same order on all nodes.

The BBN Butterfly [BBN85] family of commercial multiprocessors supports a cacheless page-granularity NUMA shared memory. Any processor can access any page of shared memory, but because there are no per-processor caches, non-local accesses are substantially slower than local accesses. This requires the programmer or operating system to carefully allocate pages to processors based on how it is being accessed or to support replication in software [BFS89, CF89].

Finally, the KSR-1 [BFKR92] is a scalable shared memory multiprocessor that, like the DDM, supports an all-cache model for memory. Thus, it incorporates specialized hardware support for maintaining consistency across multiple rings of processors. The KSR-1 supports a fairly conventional directory-based write-invalidate consistency protocol, which limits its scalability when there is frequent write sharing. Unlike the hardware systems discussed above, it is in commercial production and being used by real users. Most of the systems that have been sold are 32-node (single-ring) systems, so it is not clear how scalable the implementation is.

Chapter 6

Conclusions

The goal of the Munin project was to develop techniques for providing efficient support for the execution of shared memory parallel programs on distributed memory multiprocessors, without requiring programmers to make extensive modifications to their shared memory programs. We have achieved this goal. Munin's shared memory is different from "real" shared memory only in that it provides a release-consistent memory interface, and the shared variables must be 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 proven to be only a minor chore. Programming under Munin proved to be easier than programming using message passing, and we have achieved a level of performance comparable to well-written message passing implementations of the same applications.

The following points summarize the "lessons learned" from our experience with Munin.

- When there is little sharing, conventional DSM performs adequately.

- When there is a significant amount of sharing at a fine granularity, or sharing that involves multiple concurrent writers to a single page of data, programs run under Munin achieve from 25% to over 100% higher speedups than their conventional DSM counterparts. The presence of multiple concurrent writers can result in very poor performance in conventional DSM systems.

- False sharing is very common in real programs when the granularity of coherence is a page or full data object, so support for multiple concurrent writers to a single block of data (page or data object) should be considered a requirement for future DSM systems. Munin's *write-shared* protocol is an example of a protocol that provides such support.

- Update-based consistency protocols are useful for handling false sharing and producer-consumer data, but mechanisms that limit the amount of communication that these protocols induce are necessary to ensure that continuous updates do not overwhelm the network. Munin introduces two such mechanisms: delayed updates and an update timeout mechanism.

- Invalidation-based consistency protocols perform well in cases when there is little false sharing and a high degree of access locality, but mechanisms that limit the frequency of page "bouncing", and the resulting high latency read misses, are important in a DSM system.

- Function shipping can sometimes dramatically improve performance, so its inclusion in the design of a DSM system should be considered.

The rest of this section contains recommendations for how to improve the performance of existing and future distributed shared memory systems. These suggestions are based on our extensive experience with the Munin prototype, and are intended to guide future system implementors.

The evaluation of TSP and QSORT presented in Section 4.7 illustrates the potential of integrating *function shipping* into a DSM system. If the amount of computation performed per operation is small compared to the cost of locating and transmitting the data, it is clearly preferable to ship a request to a node with a copy of the data to have it perform the operation, modify the data, and supply the result. DSMs have been designed and implemented that use this as the sole means of communication [BHJ+87, CAL+89]. However, most often it is preferable to replicate or migrate data on demand. Supporting replication enables parallelism and keeps the single copy of the data from becoming a bottleneck. Thus, we recommend that future DSM systems support both data shipping and function shipping, which is a natural outgrowth of our belief that the runtime system should support the most common methods by which shared data is accessed.

Object-oriented DSMs encapsulate most accesses to shared data within interface functions, which allows the compiler to insert code to check to see if an object is present before it is accessed, and invoke the DSM layer to acquire a copy if it is not [BKT92, BHJ+87, CAL+89]. This can lower the overhead of DSM by avoiding costly page faults, and allows the system to dynamically choose between data shipping and function shipping [BKT92]. However, existing object-oriented DSMs restrict the use of arbitrary pointers and maintain consistency on a purely object basis, which requires programmers to restructure their programs and hand decompose shared data into objects that are not falsely shared. The benefits of object-oriented DSMs (data encapsulation and abstraction) are worth pursuing, but we recommend that the programming restrictions of existing object-oriented DSMs be eliminated in the future through use of mechanisms like those introduced in Munin (e.g., using a write-shared protocol to support falsely-shared objects).

More and better tools are needed to improve the performance of parallel programs if we are to make effective use of multiprocessing for more than trivially parallelizable scientific programs. This is especially true for distributed shared memory systems, because the overheads associated with poor programming decisions or avoidable bottlenecks are much higher than on shared-memory multiprocessors. Thus, we recommend that future DSMs be embedded in parallel programming systems that include (i) an optimizing *compiler*, (ii) a powerful runtime *linker*, (iii) a *DSM runtime system*, and (iv) *performance analysis and visualization tools*. The compiler should identify common constructs to be parallelized directly using message passing and guide the runtime system's behavior. The linker should be capable of transparently reordering and inserting code optimized for DSM, such as replacing data shipping consistency protocols with function shipping when appropriate. Given the static nature of compile-time analysis, purely compiler-based parallelization [BFKK90, ZBG88] is currently restricted to numerical computations with static shared memory access patterns. However, the combination of a parallelizing compiler and a DSM runtime system could remove this restriction. Finally, performance analysis and visualization tools will help users detect sources of overhead in their applications using profiling information and the performance statistics collected by the DSM runtime system. Simple tools of this form proved very useful during the evaluation of the Munin prototype.

Finally, many of the ideas that were effective for improving the performance of software DSM should be applicable to hardware DSM. The key issues are implementation complexity, since hardware is harder to build than software and impossible to modify once built, the amount of state required to maintain consistency, because a hardware cache controller has a limited amount of expensive static RAM, and performance. While not all of the techniques that were developed in Munin will translate effectively to hardware implementation, we recommend that the following techniques be evaluated in the context of hardware DSM: (i) support for multiple consistency protocols, and specifically both a write-invalidate and write-update protocol, (ii) support for delayed updates/invalidates and multiple concurrent writers, and (iii) the use of an update timeout mechanism to limit the impact of updates to stale data.

Bibliography

[AB86] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[ABB+86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proceedings of the Summer Usenix Conference*, July 1986.

[Adv] Advanced Micro Devices. Am 7990: Local area network controller for LANCE. Programmer's Manual.

[AG88] 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.

[AHJ91] 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.

[ALKK90] 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.

[ALL89] T.E. Anderson, E.D. Lazowska, and H.M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '89)*, pages 49–60, May 1989.

[And90] T.E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[ARS89] V. Abrossimov, M. Rozier, and M. Shapiro. Generic virtual memory management in operating systems kernels. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 123–136, December 1989.

[ASHH88] 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.

[BBN85] BBN. Butterfly parallel processor overview. Cambridge, MA, 1985.

[BCZ90a] 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.

[BCZ90b] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 168–176, March 1990.

[BDGS92] J.K. Bennett, S. Dwarkadas, J.A. Greenwood, and E. Speight. Willow: A scalable shared memory multiprocessor. In *Proceedings of Supercomputing '92*, pages 336–345, November 1992.

[BFKK90] 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.

[BFKR92] H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie. Overview of the KSR1 computer system. Technical Report KSR-TR-9002001, Kendall Square Research, February 1992.

[BFS89] W.J. Bolosky, R.P. Fitzgerald, and M.L. Scott. Simple but effective techniques for NUMA memory management. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 19–31, December 1989.

[BHJ+87] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–74, January 1987.

[BHJL86] 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. Special Issue of SIGPLAN Notices, Volume 21, Number 11, November, 1986.

[BKT92] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, pages 190–205, March 1992.

[BLL88] 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.

[BP85] C. S. Burrus and T. W. Parks. DFT/FFT and Convolution Algorithms. *Wiley-Interscience*, 1985.

[BR90] 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.

[BT88] 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.

[BZS93] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway distributed shared memory system. In *COMPCON '93*, pages 528–537, February 1993.

[CAL+89] 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.

[CBZ91] 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.

[CD88] E.C. Cooper and R.P. Draves. C threads. Technical Report CS-88-154, Carnegie Mellon University, June 1988.

[CF78] L. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[CF89] A.L. Cox and R.J. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with PLATINUM. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 32–44, December 1989.

[CF93] A.L. Cox and R.J. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 98–108, May 1993.

[CKA91] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, April 1991.

[CZ83] 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.

[DCM+90] P. Dasgupta, R.C. Chen, S. Menon, M. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. LeBlanc Jr., W. Applebe, J.M. Bernabeu-Auban, P.W. Hutto, M.Y.A. Khalidi, and C.J. Wileknloh. The design and implementation of the Clouds distributed operating system. *Computing Systems Journal*, 3, Winter 1990.

[DKCZ93] S. Dwarkadas, P. Keleher, A.L. Cox, and W. Zwaenepoel. Evaluation of release consistent software distributed shared memory on emerging network technology. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 144–155, June 1993.

[DL92] C. Dubnicki and T. LeBlanc. Adjustable block size coherent caches. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 170–180, May 1992.

[DS90] M. Dubois and C. Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Transactions on Computers*, 16(6):660–673, June 1990.

[DSB86] M. Dubois, C. Scheurich, and F.A. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, May 1986.

[DSF88] G.S. Delp, A.S. Sethi, and D.J. Farber. An analysis of MemNet: An experiment in high-speed shared-memory local networking. In *Proceedings of the Sigcomm '88 Symposium*, pages 165–174, August 1988.

[Dwa92] S. Dwarkadas. *Synchronization, Coherence, and Consistency for High Performance Shared-Memory Multiprocessing.* PhD thesis, Rice University, September 1992.

[EK88a] 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.

[EK88b] S.J. Eggers and R.H. Katz. Evaluating the performance of four snooping cache coherency protocols. Technical Report UCB/CSD 88/478, University of California at Berkeley, December 1988.

[EZL89] D.L. Eager, J. Zahorjan, and E.D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, March 1989.

[FP89] B. Fleisch and G. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 211–223, December 1989.

[GGH91] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluations of memory consistency models for shared-memory multiprocessors. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, April 1991.

[GLL+90] 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.

[Goo91] J.R. Goodman. Cache consistency and sequential consistency. Technical Report CS-1006, University of Wisconsin-Madison, February 1991.

[GVW89] J. R. Goodman, M. K. Vernon, and P.J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessor. In *Proceedings of the 3rd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, April 1989.

[HFM88] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, January 1988.

[Int90] Intel Corporation. i860 64-bit microprocessor programmer's manual. Santa Clara, California, 1990.

[JLHB88] 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.

[Kaa92] M.F. Kaashoek. *Group Communication in Distributed Computer Systems*. PhD thesis, Vrije Universiteit, Amsterdam, December 1992.

[KCZ92] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[KEW+85] R. Katz, S. Eggers, D. Wood, C.L. Perkins, and R. Sheldon. Implementing a cache consistency protocol. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 276–283, June 1985.

[KMRS86] A.R. Karlin, M.S. Manasse, L. Rudolph, and D.D. Sleator. Competitive snoopy caching. In *Proceedings of the 16th Annual IEEE Symposium on the Foundations of Computer Science*, pages 244–254, 1986.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LH89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Li86] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, September 1986.

[LLG+90] 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.

[LMKQ89] S.J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman. *The Design And Implementation of the 4.3BSD Unix Operating System*. Addison-Wesley, Reading, MA, 1989.

[LR80] B.W. Lampson and D.D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[LS88] R.J. Lipton and J.S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, September 1988.

[LT88] T. Lovett and S. Thakkar. The Symmetry multiprocessor system. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, August 1988.

[MB76] R.M. Metcalfe and D.R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[MCS91] J.M. Mellor-Crummey and M.L. Scott. Synchronization without contention. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 269–278, April 1991.

[MF89] 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.

[MF90] R.G. Minnich and D.J. Farber. Reducing host load, network load, and latency in a distributed shared memory. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 468–475, May 1990.

[Mot] Motorola. MC68020 32-bit microprocessor. User's Manual.

[Myr90] Myrias Corporation. System overview. Edmonton, Alberta, 1990.

[NL91] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[PP84] M. Papamarcos and J. Patel. A low overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348–354, May 1984.

[RAK88] 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.

[SA88] R.L. Sites and A.Agarwal. Multiprocessor cache analysis using ATUM. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 186–195, June 1988.

[SBS93] P. Stenström, M. Brorsson, and L. Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[SFGB93] E. Speight, K. Fletcher, J. Greenwood, and J.K. Bennett. ParaView: Performance debugging through visualization of shared data. August 1993. Submitted for publication.

[Spe93] E. Speight. ParaView: Performance debugging through visualization of shared data. Master's thesis, Rice University, August 1993.

[SZ90] M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, 24(5):54–64, May 1990.

[TSS88] C.P. Thacker, L.C. Stewart, and E.H. Satterthewaite, Jr. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[VF92] 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.

[WG89] 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.

[WH88] D.H.D. Warren and S. Haridi. The Data Diffusion machine - A shared virtual memory architecture for parallel execution of logic programs. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, pages 943–952, Tokyo, Japan, December 1988.

[WHL92] L.D. Wittie, G. Hermannsson, and A. Li. Eager sharing for efficient massive parallelism. In *1992 International Conference on Parallel Processing*, pages 251–255, St. Charles, IL, August 1992.

[WL92] A. Wilson and R. LaRowe. Hiding shared memory reference latency on the GalacticaNet distributed shared memory architecture. *Journal of Parallel and Distributed Computing*, 15(4):351–367, August 1992.

[YTR+87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 63–76, October 1987.

[ZB92] R.N. Zucker and J.-L. Baer. A performance study of memory consistency models. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 2–12, May 1992.

[ZBG88] H.P. Zima, H.J. Bast, and M. Gerndt. Superb: A tool for semi-automatic SIMD/MIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

[ZSM90] S. Zhou, M. Stumm, and T. McInerney. Extending distributed shared memory to heterogeneous environments. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 30–37, May 1990.

Appendix A

Example Munin Program

```
#include <stdio.h>
#include <diff.h>

shared write_shared ELEMENT elements[MATRIX_SIZE][MATRIX_SIZE];
ELEMENT scratch [MATRIX_SIZE][MATRIX_SIZE];
ELEMENT edge_init;

SynchId barrier;

unsigned num_hosts, iterations, dump, sequential;

SystemCode error;

ProcessId RootPid;
```

```
/*
 *  By default, user_init is called before the remote hosts are spawned,
 * so any changes to "non-shared" data items that are performed here get
 * propagated to the children.
 *
 * Note: you CAN NOT create threads here, because the remote hosts have
 * not been created.  Basically, this should contain the sequential
 * initialization code for the program.
 */
user_init(argc,argv)
    unsigned argc;
    char **argv;
{
    int i, j, done;
    char *cp, hostname[128];
    File *fp;

    num_hosts = DEFAULT_NUM_HOSTS;
    iterations = DEFAULT_ITERATIONS;
    edge_init = (ELEMENT) DEFAULT_EDGE_VALUE;
    sequential = 0;

    for (i = 1; i < argc; i++) {
        done = 0;
        if (argv[i][0] != '-') {
            fprintf(stderr,"Bad option %d: <%s>.\n", i, argv[i]);
            continue;
        }
        for (cp = &argv[i][1]; *cp != NULL && !done; cp++) {
            switch(*cp) {
                case 'D': dump = 1; break;
                case 'n': cp++; sscanf(cp, "%d", &num_hosts); done = 1; break;
                case 'i': cp++; sscanf(cp, "%d", &iterations); done = 1; break;
                case 'e': cp++; sscanf(cp, "%d", &j);
                          edge_init = (ELEMENT) j; done = 1;break;
                default:  fprintf(stderr, "Bad option: [%c]\n", *cp); break;
            }
        }
    }
```

```
    if (num_hosts == 1) sequential = 1;
    else {
        /* Read in the hosts to use. */
        fp = fopen(".hostlist", "r");
        if (fp == NULL) {
            fprintf(stderr, "Can't open host table file\n");
            exit(1);
        }
        for (; machine_count < num_hosts; machine_count++) {
            if (fgets(hostname, sizeof(hostname), fp) == NULL) {
                fprintf(stderr, "no more host names in file\n");
                exit(1);
            }
            i = strlen(hostname) - 1;
            hostname[i] = '\0';
            MachineName[machine_count] = (char *) malloc(i);
            strcpy(MachineName[machine_count],hostname);
        }
    }

    /* Initialize the external edge elements. */
    for (i = 0; i < MATRIX_SIZE; i++)
        elements[0][i] = elements[i][0] =
        elements[MATRIX_SIZE-1][i] = elements[i][MATRIX_SIZE-1] = edge_init;

    /* Initialize the internal elements. */
    for (i = 1; i < MATRIX_SIZE - 1; i++) {
        for (j = 1; j < MATRIX_SIZE - 1; j++) {
            elements[i][j] = (ELEMENT) 0;
        }
    }

    /* Create a barrier at which all the children and the root can join. */
    if (!sequential) barrier = CreateBarrier(num_hosts);

    RootPid = MyPid;
}
```

```
main(argc,argv)
    unsigned argc;
    char **argv;
{
    extern RootMessage *RootMsg;
    register int i, j;
    int rows, start, end, root_rows, extras;
    double total_time;

    rows = MATRIX_SIZE / num_hosts;     /* Number of row/thread, normally. */
    extras = MATRIX_SIZE % num_hosts;   /* Extra rows if evenly divided.   */

    /* Start handing out rows, starting from the end.  */
    /* Remember that the first one is on an edge.      */
    end = MATRIX_SIZE - 2;
    start = MATRIX_SIZE - rows;

    /* Give the root the first extra row, since it'd   */
    /* otherwise get one less row, being on an edge.   */
    if (sequential) root_rows = MATRIX_SIZE - 2;
    else if (extras) {
        root_rows = rows;
        extras--;
    }
    else root_rows = rows - 1;

    for (i = 1; i < num_hosts; i++) {
        /* Create and start num_hosts Worker processes, the last of which */
        /* will be on the same host as the root process.  Note, we can't  */
        /* have any thread try to perform the SOR on the edge elements,   */
        /* which is why the start and rows parts have conditionals.       */

        if (extras) {
            extras--;
            start--;
        }
```

```
        /* The arguments to CreateThread() are the address of the routine  */
        /* that the thread is to execute, the number of the node on which  */
        /* it is to execute, the number of arguments to pass to the thread */
        /* and the thread arguments themselves.                            */

        CreateThread(Worker, num_hosts-i, 3, num_hosts-i, start, end);
        end = start - 1;
        start -= rows;
    }

    StartTimer();

    Worker(0, 1, root_rows); /* Call Worker() on the root node.    */

    /* Read the entire result matrix back to the root node to avoid   */
    /* cheating and leaving the result scattered throughout the       */
    /* system.  Don't include I/O time to dump matrix in elapsed time */

    for (i = 0; i < MATRIX_SIZE; i++) start = elements[i][i];

    total_time = GetElapsedTime();

    printf("\n-----------------\n\n");
    for (i = 0; i < argc; i++) printf("%s ", argv[i]);
    printf("\n\n    Total : %4.2f secs\n", total_time);

    if (dump) {
        sprintf(dumpfile, "DumpFile.s%d.i%d.n%d",
                MATRIX_SIZE, iterations, num_hosts);
        unlink(dumpfile);
        PrintMatrix(iterations, 0, MATRIX_SIZE - 1, dumpfile);
    }
}
```

```
Worker(wnum, first, last)
   unsigned wnum, first, last;
{
    register ELEMENT sum;
    register ELEMENT *cur, *res, *start, *end;
    register int row, col, i;

    start = &(elements[first][0]);
    end   = &(elements[last][MATRIX_SIZE]);

    if (!sequential) WaitAtBarrier(barrier);/* Wait for the go ahead. */

    for (i = 0; i < iterations; i++) {
        /* Perform the finite differencing operation. */
        res = &(scratch[0][0]);
        for (row = first; row <= last; row++, res++) {
            *res = elements[row][0]; /* Copy left edge element. */
            res++;
            for (col = 1; col < MATRIX_SIZE - 1; col++, res++) {
                *res = (elements[row-1][col] + elements[row+1][col] +
                        elements[row][col-1] + elements[row][col+1])/4;
            } /* Traverse this row. */
            *res = elements[row][MATRIX_SIZE-1]; /* Copy right edge element. */
        } /* Traverse rows for which we're responsible. */

        /* Wait until all worker threads finish computing. */
        if (!sequential) WaitAtBarrier(barrier);

        /* Copy the new copies into the main array. */
        res = &(scratch[0][0]);
        for (row = first; row <= last; row++) {
            for (col = 0; col < MATRIX_SIZE; col++, res++) {
                elements[row][col]= *res;
            }
        }

        /* Wait until all worker threads finish copying. */
        if (!sequential) WaitAtBarrier(barrier);
    }
}
```

```
/*
 * PrintMatrix():
 *
 *        Pretty prints the global elements matrix.
 *
 */
PrintMatrix(phase, start_row, last_row, filename)
    int phase, start_row, last_row;
    char *filename;
{
    int i,j;
    File *dumpf;

    dumpf = Open(filename, FAPPEND, &error);
    if (error != OK) {
       fprintf(stderr,"Couldn't open %s: %s.\n", filename, ErrorString(error));
       Flush(stderr);
       exit(1);
    }
    fprintf(dumpf,"\nPass %d:\n", phase);
    for (i = start_row; i <= last_row; i++) {
       fprintf(dumpf,"%3d:", i);
       for (j = 0; j+8 <= MATRIX_SIZE; j += 8)
           fprintf(dumpf,"\t%5.0f %5.0f %5.0f %5.0f %5.0f %5.0f %5.0f %5.0f\n",
               (double) elements[i][ j ], (double) elements[i][j+1],
               (double) elements[i][j+2], (double) elements[i][j+3],
               (double) elements[i][j+4], (double) elements[i][j+5],
               (double) elements[i][j+6], (double) elements[i][j+7]);
       if (j != MATRIX_SIZE) fprintf(dumpf, "\t");
       for (; j < MATRIX_SIZE; j++)
           fprintf(dumpf, "%5.0f ", (double) elements[i][j]);
       if (j != MATRIX_SIZE) fprintf(dumpf,"\n");
    }
    Flush(dumpf);
    Close(dumpf);
}
```