INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6” x 9” black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700  800/521-0600
RICE UNIVERSITY

Efficient Runtime Support for Cluster-Based Distributed Shared Memory Multiprocessors

by

William Evan Speight

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

John K. Bennett, Chairman
Associate Professor of Electrical and Computer Engineering

J. Robertump
Professor of Electrical and Computer Engineering

James B. Sinclair
Associate Professor of Electrical and Computer Engineering

Peter Drusche
Assistant Professor of Computer Science

Houston, Texas
August, 1997
Efficient Runtime Support for Cluster-Based Distributed Shared Memory Multiprocessors

William Evan Speight

Abstract

Distributed shared memory (DSM) systems provide a shared memory programming paradigm on top of a physically distributed network of computers. The DSM system removes the necessity for programmers to move data explicitly between processors. The principle challenge in the development of an efficient DSM system lies in reducing the amount of communication necessary to maintain coherence to an absolute minimum. This thesis presents Brazos, a DSM system for use in an environment of symmetric multiprocessor (SMP) personal computers that are networked together by industry-standard 100 Mbps FastEthernet. Brazos is distinguished by its use of application-level multithreading, selective multicast, adaptive runtime mechanisms, and a unique performance history mechanism. Through the detailed analysis of twelve scientific programs, we show that Brazos outperforms the current state-of-the-art software DSM system by an average of 83%, and outperforms a version of the same DSM system that has been altered to take advantage of SMP personal computers by an average of 32%. Our results indicate that networks of commodity personal computers using available PC networks and operating systems can perform comparably on a wide variety of scientific applications to more traditional networks of high-end engineering workstations.
Acknowledgments

First, I would like to thank my committee members, J.R. Jump, Bart Sinclair, and Peter Druschel, for the help and guidance on my thesis, and for showing me what it takes to produce quality research. I would like to especially thank my advisor, John Bennett, for not only all of the wonderful discussion of off-the-wall ideas, but also for treating me as a colleague long before I finished this work.

I want to also thank all my friends at Rice over the last six years. My original officemates, Kathi Fletcher and Jay Greenwood, ensured that I never got bored during my first three years, and provided me with companionship and help during my transition to graduate life. Ram Rajamony, Mike Wu, Hazim Shafi, and Rob Butera have been particularly fun to be around, mostly because of the incredibly large amounts of time we seem to be able to waste together!

I would like to thank Compaq Computer Corporation and Schlumberger Companies for their generous equipment donations, without which this research would have never been completed.

I must thank my family who have encouraged me to keep up with this madness until the end. Without the support of Mom, Dad, and my brother, I don’t think I would have lasted this long. From my earliest school beginnings, they have always encouraged me to “soar”, and they have always been there for me whenever I fell. I love you all very much.

Finally, I dedicate this thesis to my wife, Audrey, who makes everything worthwhile. She has been my rock throughout my graduate career, and I can’t imagine having gone through the last six years of my life without her guidance, patience, love, and support. Now it’s your turn to lean on me for a while!
Contents

Abstract ii
Acknowledgments iii
List of Illustrations vii
List of Tables ix

1 Introduction 1
   1.1 Background ................................................. 4
   1.1.1 Using Relaxed Consistency Models to Reduce Communication 5
   1.1.2 Using Other Performance-Enhancement Techniques ............ 6
   1.2 The Brazos DSM System .................................... 8
   1.3 Organization .............................................. 9

2 Reducing Communication in Software DSM Systems 10
   2.1 Software Distributed Shared Memory Systems .................. 10
   2.1.1 Consistency Models ....................................... 10
   2.1.2 Differences in Lazy vs. Eager Page Management Techniques . 14
   2.2 Multicast-Based Protocol Design Issues ....................... 17
   2.3 Software Scope Consistency .................................. 20
   2.4 Improving Performance Through Adaptive Techniques ........... 24
      2.4.1 Early Update Protocol ................................... 25
      2.4.2 Dynamic Copyset Reduction ............................ 27
      2.4.3 Adaptive Page-Based Protocol .......................... 27
      2.4.4 Page Management Transitions in Brazos ................ 28
   2.5 Maintaining Histories Across Execution Runs .................. 29
2.5.1 Saving Adaptive Thresholds ........................................ 29
2.5.2 Variable-Specific History ........................................ 30

3 Implementation of Brazos ........................................ 31

3.1 Windows NT and DSM ........................................ 32
   3.1.1 Signals ......................................................... 32
   3.1.2 Structured Exception Handling ............................. 33
   3.1.3 WinSock ....................................................... 35
   3.1.4 Application Performance of Windows NT vs. Solaris .... 36

3.2 Programming for Brazos ........................................ 38

3.3 Components of the Brazos System ............................. 39
   3.3.1 Brazos Graphical User Interface ......................... 39
   3.3.2 Brazos Service ............................................... 42
   3.3.3 Multithreaded Page Access Protection ................... 43

3.4 Brazos Runtime System ......................................... 44
   3.4.1 Initialization ............................................... 44
   3.4.2 Execution .................................................... 51
   3.4.3 Adaptive Protocol Implementation ....................... 56

3.5 Implementation of the Brazos History Mechanism ........... 62
   3.5.1 Collecting Variable Information at Runtime ............ 62
   3.5.2 Obtaining Variable-Specific Runtime Performance Data .. 63

4 Experimental Results ........................................ 66

4.1 Experimental Platform ......................................... 66
4.2 Applications .................................................... 66
4.3 Performance of Brazos ........................................ 74
   4.3.1 Overview .................................................... 75
   4.3.2 Per-Application Analysis ................................. 79
   4.3.3 Evaluation of Adaptive Protocols ................. 101
5 Related Work

5.1 Software DSM Systems ........................................ 123
5.2 Hardware and Hybrid DSM Systems ......................... 128
5.3 Runtime Adaptive Protocols .................................. 130

6 Conclusions and Future Work .................................. 132

Bibliography ......................................................... 137
Illustrations

1.1 Average Request-Reply Round-Trip Latency .................. 3

2.1 Comparison of SC, ERC, and LRC Consistency Models .......... 11
2.2 Page Management in Munin vs. TreadMarks .................. 14
2.3 Multicast Protocol in Brazos ............................... 18
2.4 Messages in LRC and SScC ................................. 21
2.5 Multicast Conflicts .................................. 25
2.6 Page Protocol State Transitions in Brazos .................. 29

3.1 Use of try-except Block in Brazos ............................ 34
3.2 Average Network Throughput in Windows NT .................. 36
3.3 Brazos DSM System Components .............................. 40
3.4 The Brazos GUI ....................................... 41
3.5 Brazos Program Flow Chart ................................. 45
3.6 Relationship Between Page and Diff Structures in Brazos .... 46
3.7 Assigning Multicast Addresses to Communication Sockets .... 48
3.8 Reply Thread Flow Chart .................................. 50
3.9 Application Thread Flow Chart ............................... 52
3.10 Brazos History Mechanism ................................ 63

4.1 LU Problem Decomposition .................................. 69
4.2 Performance of Brazos ..................................... 75
4.3 Application Execution Time Breakdown ........................................ 76
4.4 Breakdown of DSM Overhead ................................................. 78
4.5 Barnes Hut Speedups ......................................................... 80
4.6 Network Usage ............................................................... 81
4.7 Using Multicast to Avoid False Sharing in Barnes Hut .......... 83
4.8 CG Speedups ................................................................. 84
4.9 FFT-3D Speedups ............................................................. 85
4.10 ILINK Speedups ............................................................... 86
4.11 Integer Sort Speedups ....................................................... 88
4.12 LU Speedups ................................................................. 90
4.13 Breakdown of Access Violation Faults in LRC ....................... 91
4.14 MG Speedups ................................................................. 93
4.15 Ocean Speedups ............................................................. 94
4.16 Raytrace Speedups ......................................................... 95
4.17 SOR Speedups ............................................................... 96
4.18 TSP Speedups ............................................................... 98
4.19 Water Speedups ............................................................. 99
4.20 Effect of Dynamic Copyset Reduction Mechanism ................. 103
4.21 Effect of Early Update Mechanism ..................................... 105
4.22 Speedup of Selected Applications on 16 Processors ............... 109
4.23 Effect of Varying Thread Priorities .................................... 113
4.24 SMP Performance vs. Brazos DSM Performance (1 of 2) .. 117
4.25 SMP Performance vs. Brazos DSM Performance (2 of 2) .. 118
4.26 Matrix Multiply Speedups ............................................... 120

6.1 Time from Barrier to Access Violations - ILINK .......... 136
Tables

2.1 Page Management Advantages/Disadvantages .................. 16

3.1 Basic System Timings ...................................... 35
3.2 Comparison of TreadMarks on Solaris and NT ............... 37
3.3 Added Overheads of Adaptive Protocols ..................... 60

4.1 Program Characteristics ...................................... 67
4.2 DSM Systems Used in Results Chapter ....................... 74
4.3 Breakdown of Execution Time ................................ 77
4.4 Breakdown of DSM Overhead .................................. 77
4.5 Indirect Multicast Diff Analysis .............................. 82
4.6 Diff-Related Overheads Observed Per Process ............... 89
4.7 Reduction in Indirect Multicast Diffs with DCR ........... 102
4.9 Effect of Pinning User Threads to Processors ............... 115
4.10 Advantage of Allowing Multiple Outstanding Network Requests ........................................ 116
Chapter 1

Introduction

Recent improvements in the performance of commodity processors and networks have made networks of workstations (NOWs) [5] an increasingly attractive alternative to large bus-based multiprocessors. Although the cost-to-performance ratio of these systems alone is a compelling argument for their use, the issue of raw performance must be satisfactorily addressed before NOWs gain widespread acceptance. Because network interconnections typically offer much lower bandwidth than local processor-memory interconnects, constructing a high-performance multiprocessor from a network of workstations poses special challenges and limits the number of processors that can be productively employed. The use of small-scale SMP (symmetric multiprocessor) bus-based systems allows a larger number of processors to be used in a network of workstations, provided the interconnection network can take advantage of the available shared memory hardware. These systems offer performance comparable to traditional large-scale hardware shared memory systems for those applications that do not exhibit very high communication rates.

While distributed memory systems such as NOWs are easy to build, applications for distributed systems are difficult to develop due to the need to send and receive data explicitly between machines. By providing an abstraction of globally shared memory on top of the physically distributed memories present on networked workstations, it is possible to combine the programming advantages of shared memory and the cost advantages of distributed memory. These distributed shared memory (DSM) runtime systems transparently intercept user accesses to remote memory and translate these accesses into messages appropriate to the underlying communication
media. The programmer is thus given the illusion of a large global address space encompassing all available memory, eliminating the task of explicitly moving data between processes located on separate machines. Both hardware DSM systems (e.g. Alewife [15], DASH [36], FLASH [31]) and software DSM systems (e.g. Ivy [37], Munin [13], TreadMarks [28]) have been implemented.

The majority of software DSM systems use page-based memory protection hardware and the low-level message passing facilities of the host operating system to implement the necessary shared memory abstractions. Programmers write programs using the traditional shared memory parallel programming style, without concern for where data may actually reside in the system. The DSM system intercepts data accesses to remote data, fetches data across the network, and maintains coherence among multiple copies of data resident on different machines. The performance of software DSM systems is challenged by the large size of the unit of sharing (a page) and the high latency associated with accessing remote memory. One of the principal contributions to this latency is the high operating system costs associated with initiating an inter-process network message.

Figure 1.1 depicts a simple request-reply sequence that typifies the type of communication pattern usually seen in DSM systems: one process sends a small request to another process, which in turn responds with a (usually) larger message. Figure 1.1 shows the average round-trip latency of a request-reply message pair between two servers connected by a 100 Mbps FastEthernet. For comparison, the ideal latency (wire time) is also shown.

For the region indicated "Typical Range of DSM Response Messages", we see that there is anywhere from a 60.9% to 99.6% increase in latency for the observed round-trip message as compared to the "wire time" for responses of a few bytes up to a single 4 Kbyte page. This translates into a maximum realized bandwidth of .92 Mbit/sec to 39.2 Mbit/sec for the same response size range. Figure 1.1 shows that the high overhead associated with communication calls can severely increase the
Figure 1.1: Average Request-Reply Round-Trip Latency

latency for each message sent, especially for short messages. Thus, for a DSM system to perform well in this environment, the number of messages must be minimized, and short messages should be combined and sent simultaneously to amortize the high startup cost associated with each message.

To date, most DSM systems have been implemented on Unix-based engineering workstations. DSM runtime systems are generally implemented as a static library that is linked into a parallel program at compile-time. This library contains code to intercept data accesses to stale data through the virtual page protection mechanisms provided by the host operating system; synchronization routines for the programmer to use when processes must synchronize; and functions that use the message passing facilities provided by the OS for the underlying communication media. The majority of these systems have been written to support a single application thread running on
each available processor. Such systems do not fare well on today’s widely available SMP workstations because they can not take advantage of the available hardware coherence mechanisms. Additionally, the use of point-to-point messaging primitives produces excessively high communication rates as the number of participating processes increases.

This thesis describes the design and performance of Brazos, a DSM system that contains native support for multithreaded architectures and operating systems. Brazos allows the operating system and hardware to maintain coherence between threads located on the same multiprocessor, only interfering with a program’s execution when a remote data access must be handled. Brazos also makes extensive use of the hardware multicast features available in modern PC operating systems. Through intelligent use of multicast, the communication required to execute a shared memory program correctly can be reduced substantially by specifying multiple recipients for data messages. However, the naive use of multicast can result in extraneous communication and therefore worsen overall performance. To address this and other problems associated with the use of a multicast-based runtime system, we have derived new adaptive performance-tuning techniques. These techniques further improve overall application performance by tailoring the DSM communication patterns to each individual program. Finally, we have incorporated a history mechanism that retains performance-related information between program executions.

1.1 Background

In this section we present a brief discussion of previous efforts to reduce the amount of communication necessary in software DSM systems. A detailed discussion of related work can be found in Chapter 5.
1.1.1 Using Relaxed Consistency Models to Reduce Communication

Early DSM systems, such as IVY [37], enforced sequential consistency [32] to maintain coherence between processes in a DSM system. Sequential consistency requires that each write be globally performed before the process issuing the write is allowed to proceed. This restriction severely limits the performance achievable by these systems because of the large amount of network traffic necessary to maintain coherence on each write to shared data.

In order to relieve the network pressure caused by sequential consistency, the Munin [13] system used a multiple writers protocol. This protocol allows more than one process to modify a portion of a shared page of data at the same time, as long as the programmer ensures that a single word is not write-shared without an intervening synchronization point. By buffering writes and only propagating them at synchronization points, this relaxed consistency protocol greatly reduces the amount of DSM traffic generated, thus improving overall performance.

Munin is an example of an eager DSM system, in which data is moved in expectation of being needed by other processes. This can be beneficial to performance if the data is actually used. However, the added communication of moving data speculatively can adversely impact performance if the data is never accessed. The TreadMarks [27] system implements a lazy protocol that only moves data when it is absolutely necessary to maintain program correctness. In general, lazy protocols outperform eager protocols, mainly because it is very difficult to predict when data will be needed by another process accurately. However, the memory overhead required by lazy protocols to keep track of information relating to shared pages is much larger than that required by eager protocols, and can adversely affect performance in some instances. The ability to choose either an eager or lazy protocol on a page-by-page basis offers the advantages of both methods.
1.1.2 Using Other Performance-Enhancement Techniques

Static Performance Enhancements Static performance enhancement (SPE) techniques try to alleviate some specific performance problem common to a large class of parallel programs. A performance enhancement is static if the policies dictating when to apply the enhancement are determined before the program is executed, and do not change in response to changes in runtime behavior or measured performance.

The largest class of SPEs include techniques that make use of the compiler to influence the execution behavior of the parallel program. These techniques include compile-time data transformations intended to reduce false sharing [25], compiler-directed prefetching [38], and data restructuring to avoid collision or capacity misses [47].

SPEs can also include user-provided "hints". For example, the Munin system [13] allows users to classify shared variables according to the variable's expected data access behavior. The software mechanism used to maintain coherence is determined by these user-specified classifications. This enhancement is static in that once the program begins execution, the assignment of coherence protocol to data elements does not change.

SPE techniques have two advantages. First, by placing the responsibility of determining when to apply enhancement techniques on the compiler and/or user, there is little added complexity in the runtime system. This reduces the amount of expensive (in overhead) software that must be added to the system in order to achieve performance improvements. Second, because everything is known before runtime, SPEs do not require time to adapt to data access behavior while the program is executing. Therefore, the enhancement is present from the outset of the program and the benefits are immediately evident.

There are also several disadvantages associated with static techniques. In the case of labeling shared data, as is done in Munin, correctly classifying data elements may be difficult, time consuming, or error-prone. Compiler techniques require a non-
standard compiler written with the specific desired performance improvements in mind. Such compilers are difficult to construct and not likely to be widely available. Furthermore, compile-time analysis can not correctly predict runtime-dependent data access behavior.

**Adaptive Performance Enhancements** Adaptive performance enhancement (APE) techniques attempt to affect program behavior at runtime in a manner intended to yield optimum performance. These techniques typically require modifications to the DSM runtime system, such as the update-with-timeout mechanism found in Munin [13]. This adaptive technique propagates updates to remote processes until the remote process “times-out” by not using an update for a specified length of time. At this point, the remote process informs the sender to halt update-forwarding, thereby adapting to the observed runtime behavior.

Unlike SPEs, many adaptive techniques do not require user involvement or the use of a specialized compiler. Instead, the runtime system attempts to make decisions regarding the most appropriate performance enhancement technique for the observed situation. APEs can also adapt to changes in data access behavior that may not be apparent at compile-time, thereby reducing the possibility of mis-managing data accesses.

Adaptive techniques also have several disadvantages. First, the runtime analysis overhead needed to determine when to apply the enhancement may negate the performance benefit achieved. Second, unlike static techniques, adaptive techniques require time to recognize data access behavior initially, during which time data may not be managed in an efficient manner. Finally, even though a program may have been run many times before, existing adaptive protocols start out with no historical information regarding prior program executions. This “cold start” behavior limits the efficacy of adaptive protocols, and is a focus of this research.
1.2 The Brazos DSM System

This thesis develops adaptive runtime techniques that reduce the amount of DSM communication traffic when used in conjunction with multicast-based coherence protocols. We also develop a per-application history of past adaptations across program executions. These techniques have been implemented and evaluated in Brazos, a fully functional DSM runtime system for x86 personal computers running the Windows NT operating system.

Contributions

The principal contributions of this work are demonstrated in the design, implementation, and evaluation of the Brazos Distributed Shared Memory System. Five main contributions are present in Brazos:

- A set of coherence protocols based on the use of multicast. We show that multicast can be used effectively in DSM systems to reduce the amount of communication necessary over that of a system that utilizes point-to-point communication. Several problems encountered by the use of multicast are addressed through innovative runtime adaptive performance mechanisms.

- Adaptive techniques to improve DSM performance. We demonstrate several runtime adaptive techniques to reduce the amount of DSM communication. We implement adaptive versions of three communication-reduction mechanisms: an early update protocol, dynamic copyset reduction, and a mechanism to switch between a home-based and distributed page management strategy on a page-by-page basis.

- Per-application history information. Our work shows that maintaining per-application history information on the success of our adaptive techniques across program executions and re-integrating this information into subsequent application executions can improve overall performance. We develop such a history
mechanism that does not depend on program data sets or user interaction to improve performance.

- **An implementation of software scope-consistency.** In order to address the problem of false sharing commonly found in DSM systems, we introduce a software variation of scope consistency [23] that has a performance advantage for specific classes of parallel programs. Additionally, the use of scope consistency facilitates the use of multicast for those programs that make use of small, independent critical sections.

- **A DSM system for Windows NT.** Brazos represents the first performance-oriented DSM system for use on the Windows NT operating system. We show that on identical hardware, Brazos outperforms the state-of-the-art DSM system based on Unix, despite the higher achievable network throughput under Unix.

### 1.3 Organization

Chapter 2 begins by providing an overview of memory consistency models and page management techniques used in modern DSM systems. We then discuss the use of multicast in DSM systems, software scope consistency, the adaptive performance tuning techniques used in Brazos, and the history mechanism that allows Brazos to save performance-related information across program executions. Chapter 3 describes the differences between a traditional Unix-based DSM system and a Windows NT-based system such as Brazos, including details on the implementation of all features of Brazos. Chapter 4 discusses our experimental platform and the applications that we use to evaluate our system. We then show the relative importance of these performance enhancement techniques on a wide variety of shared memory parallel programs. Chapter 5 relates our work to previous work in the area of distributed shared memory systems, and Chapter 6 provides conclusions and directions for future work.
Chapter 2

Reducing Communication in Software DSM Systems

We begin Chapter 2 with a discussion of the consistency models and page management techniques of two software DSM systems closely related to Brazos: the Munin [13] system, and the TreadMarks [28] system. This section serves as background information for the later sections of this chapter, which describe the communication reduction techniques employed by the Brazos DSM system in detail.

2.1 Software Distributed Shared Memory Systems

2.1.1 Consistency Models

Early software DSM systems such as IVY [37] relied on sequential consistency (SC) to maintain coherence between processes. SC requires that all shared accesses be consistent with some global ordering that does not violate program order:

"[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 this sequence in the order specified by its program" [32].

Sequential consistency limits the performance of shared memory systems because it places severe restrictions on the ordering of accesses to shared data [1]. However, SC is a conceptually simple model for reasoning about parallel programming because all accesses are atomic, and a processor writing to a shared variable is guaranteed to have the only valid copy of the variable in the system. Also, a read is guaranteed to
return the most-recently written value of a variable. However, SC is rarely used in practice in modern software DSM systems because of the large amount of network traffic needed to enforce this strict ordering of accesses*. Figure 2.1 compares SC to two relaxed consistency models, which are discussed below. In Figure 2.1, the red arrows indicate network messages that must be sent in order to maintain sequential consistency. As indicated, invalidation message are required before every shared write, and the processor must stall until the acknowledgment returns. Eager and lazy release consistency help to reduce the amount of time a processors stalls by combining and delaying write notifications as described below.

![Diagram](image)

**Figure 2.1 : Comparison of SC, ERC, and LRC Consistency Models**

*Release consistency [19] (RC) is a relaxed consistency model that only requires data to be made consistent at specified synchronization points in the program. Data is temporarily allowed to become inconsistent between these synchronization points, as long as there are no competing accesses. Competing accesses are two or more accesses (one of which is a write) by more than 1 process to the same memory location.*

*Synchronization accesses are frequently implemented using SC.*
without an intervening synchronization point. Data accesses are divided into special and ordinary accesses, with special accesses further broken down into synchronization accesses called *acquires* and *releases*. An acquire is an operation that grants a process exclusive access to some shared data, and a release allows other processes to access some shared data.

To implement RC, a memory subsystem must ensure:

- Before an ordinary *read* or *write* is allowed to perform with respect to any other process, all previous *acquire* accesses must be performed.

- Before a *release* access is allowed to perform with respect to any other process, all previous *read* and *write* accesses must be performed.

- Synchronization accesses must be sequentially consistent with each other\(^1\).

Munin [13] was the first software DSM system to implement RC. The use of RC allows Munin to buffer writes until required to be visible at the next release point, at which time write notifications can be sent *en masse*. Munin implements a version of RC known as *eager release consistency* (ERC). Under ERC, writes are buffered until a release point occurs. Before the release can complete, all write notifications must be propagated to all other processes in the system. The blue arrows in Figure 2.1 show the messages required under an implementation of ERC, and indicate a significant reduction in the number of messages over SC.

Because of the large granularity of sharing in DSM systems, *false sharing* can be a serious detriment to performance. False sharing occurs when multiple processes access different pieces of the same shared page of data, and at least one process performs a write operation to the page. If processes are required to gain exclusive access to the page before allowing a write to complete, the page may “ping-pong”

\(^1\)It is also possible to allow synchronization accesses to be *processor consistent*, but most RC implementations use sequentially-ordered synchronization variables.
between multiple processes as first one and then another gains exclusive access to the page. This problem is analogous to the “cache thrashing” effect seen in hardware shared memory systems, but the performance impact can be much more pronounced in software DSM systems because of the high communication overhead and large granularity of sharing. The use of a release consistent memory model allows Munin to employ a multiple writers protocol specifically to address the problem of false sharing.

Under the multiple writers protocol, valid copies of a single page are maintained by each process. This is possible because the programmer has ensured that there are no competing accesses to any one memory location on the page, under the rules of release consistency. Changes to the page are only propagated and merged at release points, reducing the opportunity for false sharing to occur.

The TreadMarks DSM system [28] introduced lazy release consistency (LRC) [27], also used in conjunction with the multiple writers protocol. LRC further delays the effect of writes until a process performs an acquire synchronization operation to a variable previously released by another process. The constraints for LRC are the following:

- Before an ordinary read or write access is allowed to perform with respect to another process, all previous acquire accesses must be performed with respect to that other process.

- Before a release access is allowed to perform with respect to any other process, all previous ordinary read and write accesses must be performed with respect to that other process.

- Synchronization accesses are sequentially consistent with respect to one another.

The green arrows in Figure 2.1 show the messages required under LRC. In practice, instead of having a process push invalidations with the release, as in Munin’s eager release consistent model, processes under LRC pull invalidations along with an acquire
message. This delays propagating the effect of shared writes until the last possible moment, without violating the constraints of release consistency.

2.1.2 Differences in Lazy vs. Eager Page Management Techniques

In addition to the coherence model used, software DSM systems differ in the management of shared pages. In the Munin system, each page has a home process, in which the most up-to-date copy of the page always resides. TreadMarks, however, has no such centralized place for processes to get copies of a modified page. Instead, processes maintain dirty pieces of each page they have written, and a process must communicate with all other processes that have dirty portions of the page. Figure 2.2 shows the difference between these two approaches.

![Figure 2.2: Page Management in Munin vs. TreadMarks](image)

Figure 2.2 shows a timeline of accesses to a single page by one process (P0) in a
DSM system. The page begins as invalid at the left side of Figure 2.2. As indicated, the page was invalidated at some time in the past by another process executing a release operation in the case of Munin, and by P0 executing an acquire operation in the case of TreadMarks. At the first read to the page at time t1, Munin sends a single request to the known page manager for this page, and receives the entire page in reply. TreadMarks, on the other hand, sends requests for diffs to other processes that have dirty copies of the page. A diff is a runlength encoding of changes made to a page since the first write to the page. Following the receipt of the diffs, TreadMarks applies each diff to the page to bring it up-to-date. At this point, both systems change the page protection on the shared page to READONLY, and allow computation to proceed. At the first write to the page (time t2), both systems make a copy of the page, known as a twin. The twin represents the state of the page before any modifications have been made. At this point, the original page protection is changed to READWRITE, and the process is free to make any changes to the page without faulting again.

At the next release point (t3), Munin constructs a diff from the current state of the page and the twin, and flushes this diff to the home process. The home process applies all diffs for this page, giving the home process an up-to-date copy of the page that it will provide to other processes in response to requests for data. Conversely, at a release point, a process in TreadMarks does nothing with respect to shared data. It is not until the following acquire that a TreadMarks process informs other processes of modified pages. As shown in Figure 2.2, a diff of the page is made only after it is requested by another process. This delaying of diff creation is known as lazy differencing, and has been shown to have performance benefits for a wide variety of shared memory applications [27] because delaying the creation of diffs can result in the combining of multiple diffs into a single diff, reducing overall communication. However, TreadMarks adds complexity at several levels in exchange for maintaining a distributed page algorithm. First, complexity is added to keep track
of which processes have dirty copies of which pages. This is accomplished by the use of intervals, implemented as vector timestamps that processes maintain for all other processes in the system. The second level of complexity comes in garbage collecting and a higher memory requirement for the maintenance of diffs because diffs must be retained until they have been seen by all processes.

An important thing to note for later reference is that in the TreadMarks case, unless a process has never accessed a page before, the process remains in the copyset (a list of processes who have a copy of the page [8]) for the page even though the page is invalid. This is because the process still may contain dirty portions of the page despite the page being invalid. In Munin, a process with an invalid page is no longer considered in the current copyset because all information regarding the correct values on the page are present only at the home process.

<table>
<thead>
<tr>
<th></th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Munin</strong></td>
<td>Only 1 message request per fault</td>
<td>May transfer extra data</td>
</tr>
<tr>
<td></td>
<td>Known destination for data request</td>
<td>Possible page manager bottleneck</td>
</tr>
<tr>
<td></td>
<td>No garbage collecting</td>
<td>Flushing possibly unused diffs</td>
</tr>
<tr>
<td><strong>TreadMarks</strong></td>
<td>Minimal amount of data transferred</td>
<td>Multiple messages required per fault</td>
</tr>
<tr>
<td></td>
<td>Data sent on demand only</td>
<td>Garbage collecting is difficult</td>
</tr>
<tr>
<td></td>
<td>Lazy diffing can coalesce diffs</td>
<td>High memory overhead for diff maintenance</td>
</tr>
</tbody>
</table>

Table 2.1: Page Management Advantages/Disadvantages

Table 2.1 outlines the advantages and disadvantages of the TreadMarks and Munin approaches to page management. In the next section, we describe how Brazos captures the advantages of both Munin and TreadMarks through the use of communication protocols based on multicast, while avoiding most of the disadvantages.
2.2 Multicast-Based Protocol Design Issues

Although there has been some limited work in using multicast in software DSM systems (see Chapter 5), there has been no work applying multicast to systems specifically to reduce the number of coherence messages required. In fact, there have been several arguments against the use of multicast, which are addressed in this section.

- **Multicast is not scalable.** It would be infeasible to construct a DSM system with hundreds of processes employing multicast communication. However, we contend that software DSM systems based on networks of physically separated computers can not effectively scale past 8 machines with current (1997) processor and network speeds, whether point-to-point or multicast messages are used. Our results support this case even for applications with relatively low amounts of required network communication. However, for medium-scale systems (up to 64 processors), it is quite feasible to connect 8 servers of 8 processors each through a hub or switch, and multicast works well with this number of servers.

- **Multicast is hardware dependent and will not always be available.** On the contrary, we expect the use of multicast to expand in the next several years, especially in the personal computer marketplace. Ethernet is by its nature a broadcast medium, and the predominant form of networks for personal computers will continue to be based on Ethernet-like protocols for the foreseeable future. The establishment of the IP Multicast Initiative [46] further supports the conclusion that multicast will continue to be included in future network hardware and software protocols.

- **Extensive use of multicast is “bad network citizenship”.** The advancements in the development of switches and routers capable of isolating multicast traffic can protect other machines on the network from being saturated by unwanted multicast packets. Additionally, performance-oriented DSM systems
will most likely be implemented on a dedicated network of computers, eliminating their effects from the rest of the network.

To reduce the amount of communication necessary between processes, Brazos makes use of the multicast primitives provided by the WinSock 2.0 library. In an Ethernet network environment, sending a multicast message is no more expensive in terms of used network bandwidth than sending a point-to-point message, and large reductions in the amount of data transferred to maintain coherence can be achieved by specifying multiple recipients for each message. Figure 2.3 shows the differences in communication patterns between a distributed page DSM system that relies on point-to-point communication and one that employs multicast.

![Figure 2.3: Multicast Protocol in Brazos](image)

In Figure 2.3, processes P0, P1, and P2 each write to a different word on the same page, as indicated by Wx, Wy, and Wz. Process P3 does not access this page, but does have a read-only copy of it from a previous access. At the first barrier, all 4 processes will have their copies of the page invalidated. The point-to-point DSM
system requires two separate requests and two separate replies for each of the read accesses by $P0$, $P1$, and $P2$ in order to bring the page up-to-date. In the multicast case, $P2$ requests dirty pieces from $P1$ and $P0$ with a single message (which includes $P2$'s dirty portion), and these processes respond not only to $P2$, but to everyone in the current copyset. Thus, after only 3 messages, all three processes ($P0$, $P1$, and $P2$) are up-to-date with respect to the page, and the read faults by $P0$ and $P1$ are avoided altogether.

Brazos combines the positive benefits of both Munin and TreadMarks through the use of multicast-based coherence protocols. As in Munin, a process only needs to send a single request to bring a page up-to-date, even though we maintain distributed dirty pieces of the page in a manner similar to TreadMarks. This reduces the chance of any one process becoming a bottleneck, as can be the case with a statically assigned home process. Additionally, diffs are not made until requested by at least one other process, providing the performance benefits of lazy diff creation, as in TreadMarks. Finally, we do not have the high memory overhead associated with maintaining a history of past diffs, as in TreadMarks, because through multicast we ensure that once one process has requested a diff for a page, all processes receive the diff. This allows us to maintain at most one outstanding diff per page per process, eliminating the need for explicit garbage collection and reducing overall memory overhead.

However, several issues still hamper the performance of naively-implemented multicast protocols. For example, in Figure 2.3, process $P3$ also receives diffs for the page being accessed, even though these are not used before the next barrier. This results in $P3$'s being interrupted to process diffs that will never be used, detracting from computation time. This problem and others are addressed by the adaptive performance tuning techniques outlined in Section 2.4.
2.3 Software Scope Consistency

Scope consistency was introduced as an enhancement to the SHRIMP AURC system [22], and is a refinement of release consistency. A discussion of scope consistency and its hardware implementation can be found in [23]. Basically, scope consistency divides the execution of a program into scopes, and shared data is only guaranteed to be consistent at the end of a scope. Scopes are characterized as global or local scopes. Global scope delimiters include global synchronization events such as barriers. After a global scope is closed (completed) all shared data in the program is guaranteed to be coherent. In an analogous manner, lock acquire-release operations delimit a local scope. When a process acquires a lock, it enters a new local scope. All changes made until the closing of the local scope (i.e., the lock release) are guaranteed to be visible to the next acquirer of the lock, but not to any other process. Further, any writes that occur before the opening of the local scope are not guaranteed to be visible until the next global synchronization event. This is in direct contrast to release consistency [19], where such writes are guaranteed to be visible after the next release. Thus, global synchronization (a barrier) in scope consistency has the same semantics as the presence of an acquire-release pair in release consistency [19], and programs without local scopes (programs that only use barriers) will behave just as they do under RC or LRC.

To be more precise about the specific conditions required to implement scope consistency for local scopes, the following conditions delineate the differences between release consistency and scope consistency using release-consistency nomenclature. Conditions associated with scope consistency only are shown in [boldface]. Conditions associated with both release consistency and scope consistency are shown in

---

\[1\text{In order to use terminology consistent with previous work in the area of relaxed consistency, "opening" and "closing" a local scope (as defined in [23]) will be considered to be equivalent to an "acquire" and "release" to the same location. "Opening" and "closing" a global scope (as defined in [23]) will be considered to be equivalent to consecutive barrier events.} \]
normal type. The use of the term “performed with respect to” in these conditions is consistent with that of [23].

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

2. Before a release is allowed to perform with respect to any other process, all previous ordinary loads and stores [after the last acquire to the same location as the release] must be performed [with respect to that process].

3. Synchronization accesses (acquires and releases) must be sequentially consistent with one another.

![Diagram](image)

**Figure 2.4 : Messages in LRC and SScC**

Figure 2.4 presents the differences between lazy release consistency (“LRC”, as used in the TreadMarks DSM system) and software scope consistency (“SScC”, as used in Brazos). The figure shows the messages that would be associated with the code fragment shown for each of the consistency models. Two possible execution paths are shown for the last statement in the code fragment. The left-hand (tmp1 = y) path will produce the intended result under release consistency, but will not result
in valid behavior under scope consistency. The value of y is undefined when it is read because scope consistency does not require the effect of the write to y by P0 to be propagated until the next global scope. There are two solutions to this problem. Either a global synchronization has to be inserted prior to this read of y or the write of y needs to be moved inside the critical section. Examples of programs that require these kind of modifications are given in [23].

In Brazos, scope consistency provides two main benefits. First, in the code fragment in Figure 2.4, the new value of x written by P0 is sent along with the lock grant to P1, thereby eliminating a message resulting from the fault of P1 when trying to increment x by five. Second, if we assume that y and z are on the same page (different from the page on which x resides), and that the right-hand path is taken for the last statement (tmp2 = z), LRC will fault on the read of z, even though z was not modified by P0. This is because y and z are falsely shared. SScC removes the effect of this false sharing by not invalidating the page containing y and z when the lock ownership is transferred to P1.

Although scope consistency is Brazos' primary consistency model, Brazos provides two alternate locking mechanisms for situations in which it is inconvenient or not possible to meet the requirements of scope consistency with respect to the placement of shared writes. Brazos provides an eager release locking semantic, in which all invalidations are pushed to all other processes in the system when a process executes a lock release. Brazos also provides a mechanism similar to lazy update, in which a process that grants an acquire to another process sends updates for all writes that have occurred since the last synchronization event. Unlike true LRC, however, this message is multicast to all processes because in order to reduce the amount of diff overhead present in true LRC systems, Brazos does not allow processes to maintain diffs created by other processes, as is necessary in the original LRC implementation.
Software Scope Consistency Implementation Issues

To implement scope consistency in the absence of hardware support such as AURC (automatic update release consistency), a mechanism needs to be in place to detect writes to shared data that occur within a critical section. This matter is complicated in a multithreaded DSM because other threads may be writing to data not protected by the critical section but that resides on the same page as critical section data. We examined several methods of dealing with this problem.

1. **Suspend all other application threads when one thread enters a critical section.** After suspending other threads, the entire shared address space would have to be protected as **READONLY** in order to detect which pages the thread writes to within the critical section affected. Clearly this method would severely limit performance by halting computation of non-involved application threads and changing page protections for the entire address space on each lock acquire.

2. **Explicitly associate variables with locks.** This method is analogous to an implementation of entry consistency [16] applied only to critical sections. The user would have to indicate a mapping that associates shared variables with each lock instance in the program. Then, on a release, these variables are sent with the lock grant to the next acquiring process. However, this mapping between data objects and locks is difficult, and the mapping can easily change during the course of a program.

3. **Annotate shared writes inside a critical section.** This is the method we implemented, mainly because it is fairly easy to write a preprocessor that simply tags all writes between an acquire-release pair as "critical section writes" in the absence of function calls within the critical section. The DSM system determines if the address being written to is indeed in the global shared space. If the write is to a shared variable, the page number, offset, and length of
the variable being written are attached to the DSM lock structure for the lock protecting the critical section, and the write is allowed to proceed. When the lock is granted to the next acquirer, this "critical section diff" list is passed along with the lock grant, and no other coherence information is communicated. The acquiring process applies the writes made by the previous lock holder, and continues.

At the end of the global scope (i.e., the departure from the next barrier instance), processes remove any critical section diffs still remaining. If a process is the last process to acquire a lock before the end of a global scope, it will be left with critical section diffs for the lock. These diffs are merged with regular diffs (diffs of variables on the same page that were not changed inside a critical section), allowing the same shared variables to be handled under the protocol for global and local scopes in the same program.

2.4 Improving Performance Through Adaptive Techniques

We have developed three adaptive runtime mechanisms for the Brazos DSM system. The first two specifically target the side-effects of employing multicast communication: an early update protocol that attempts to send bulk data before it is needed; and a dynamic copyset reduction mechanism whereby a process removes itself from the copyset of a specific page, depending upon whether or not the process wishes to receive multicast diffs for that page. We have also developed a mechanism for switching between a home-based coherence scheme (used in Munin) and a distributed page coherence scheme (used in TreadMarks) on a page-by-page basis. We discuss the rationale behind these three mechanisms in the following paragraphs.
2.4.1 Early Update Protocol

In Brazos, pages are only invalidated at the end of a global scope\(^5\). All processes receive the same list of pages to invalidate from the barrier manager. If these pages are actively shared, it is possible for multiple processes to miss on the same pages at nearly the same time. This leads to multicast conflicts and unnecessary messages, as seen in Figure 2.5.

<table>
<thead>
<tr>
<th></th>
<th>P0</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>Send msg to P2</td>
<td>Send msg to P2</td>
<td></td>
</tr>
<tr>
<td>t1</td>
<td>Wait for reply</td>
<td>Wait for reply</td>
<td>Get request from P0</td>
</tr>
<tr>
<td>t2</td>
<td></td>
<td></td>
<td>Send reply to P0, P1</td>
</tr>
<tr>
<td>t3</td>
<td>Get reply diff</td>
<td>Get “indirect” diff</td>
<td>Get request from P1</td>
</tr>
<tr>
<td>t4</td>
<td>Make page valid</td>
<td>Make page valid</td>
<td></td>
</tr>
<tr>
<td>t5</td>
<td>Program continues</td>
<td></td>
<td>Send NACK to P1</td>
</tr>
<tr>
<td>t6</td>
<td></td>
<td>Get NACK from P2</td>
<td></td>
</tr>
<tr>
<td>t7</td>
<td></td>
<td>Program continues</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.5: Multicast Conflicts

In Figure 2.5, P2 has a dirty copy of the page being accessed, and P0 and P1 each send a request for this data at time t0. P2 receives P0’s request first (or P1’s request first, it does not make a difference), and multicasts its response to both P0 and P1. P0 receives the diff it was expecting, makes the page valid, and continues with program execution. P1, however, gets an indirect diff as a result of the multicast response by P2. P1 knows this is not the response to its request, because

---

\(^5\)For simplicity, we will refer to the end of a global scope as the arrival at the next barrier instance. This is because in Brazos’ intended use of scope consistency, only global synchronization points result in pages being invalidated, and the only global synchronization event supported is the barrier. For programs that use the alternate release consistent locking mechanisms discussed in Section 3.4.2, a global scope may also refer to a release or acquire of a lock, depending on the locking semantics employed.
each request has a sequence number that must be returned with the corresponding response message. However, P1 knows that it needs this diff, because the copy of the page is invalid in memory. The diff is applied to the page, and the page is made valid. At this point (t5), P1 actually can continue program execution, but must wait for the response to the message it sent in t0. P2 responds to P1 with a negative acknowledgment, because the diff has already been sent out in the previous message. P1 receives the NACK and continues execution.

We examined two possible solutions to this problem. The first solution allowed P1 to begin execution in t5, after it received the needed diff indirectly. The NACK from P2 is simply dropped upon arrival at P1. This method allows P1 to begin execution earlier, but does not eliminate the overhead associated with the useless request sent by P1, nor the NACK sent by P2.

The second solution we examined, and finally decided to implement, is called early update. Early update identifies pages for which indirect diffs are received while the process is waiting for a response containing the same diff (the receipt of which is called a multicast conflict). Each process builds a list of these pages, and the list is transferred to the barrier manager upon arrival at the next barrier. The manager then distributes this early update page list to all processes with the barrier release message. Thereafter, when a process arrives at a barrier, it will multicast (in a single message) diffs for pages in the update list to the current aggregate copyset of all pages contained in the message. In the example above, this method not only eliminates both diff requests and responses in return for a single multicast message, but also prevents the faults from occurring in the first place, since updated pages are not invalidated upon a barrier release.

The dynamic copyset reduction mechanism, described in the next section, is used to switch a page back to the default invalidation protocol.
2.4.2 Dynamic Copyset Reduction

As pointed out at the end of Section 2.2, processes receive indirect diffs because they possess an invalid copy of the page, but these diffs may never be used. This takes time away from the compute process by requiring a process to apply incoming diffs that will never be accessed. In order to reduce this effect, we have implemented a dynamic copyset reduction algorithm that seeks to eliminate useless indirect diffs by allowing processes to remove themselves from the current copyset for a specific page.

The mechanism for deciding when to drop out of the copyset is implemented as follows. We associate a counter with each page of shared data. This counter is initially set to an application-specific threshold value derived from past executions of the program (see Section 2.5 for a discussion of the Brazos history mechanism). Whenever the process receives indirect diffs for a page that is not accessed before the next time the page is invalidated, the counter value is decremented by 1. If the process uses the indirect diffs, the counter is reset. If the counter reaches 0, the process removes itself from the copyset at the next global synchronization point.

The dynamic copyset reduction mechanism is also used to transition a page back to an invalidation protocol from the early update protocol described above. In a manner analogous to that described above, we count updates to pages that are not used before the next update arrives. When this counter reaches a certain threshold, a process will request to be dropped from the copyset. When the copyset is reduced to a single process, the page reverts back to an invalidation protocol.

2.4.3 Adaptive Page-Based Protocol

The final adaptive mechanism we employ allows a page to be managed either as a home-based page or a distributed page. The inclusion of this mechanism stems from observations about the wide disparity in the size of diffs, not only between applications, but also within the same application. Situations arise in which several processes may share a page of data, but one process writes to a much larger portion
of the page than any other process. The cost of making this one large diff may be detrimental to performance, especially if this pattern persists throughout a large percentage of the program’s execution. We therefore provide a mechanism whereby processes may flush their diffs to a self-appointed "home process", which applies these diffs to its copy of a page and invalidates other processes’ copies. When processes fault on this page, they simply obtain an up-to-date copy of the entire page from the home process instead of requesting diffs. Thus it is possible to have pages that follow a distributed algorithm and pages that follow this home-based algorithm in the same program.

Changing data access patterns during the course of program execution may dictate that a distributed page management algorithm should be re-instated for a home-based page. We provide a mechanism for this to occur by periodically ensuring that the home process is still writing to a portion of the page larger than the diffs it is receiving for it. If not, the page will revert to a distributed-page management algorithm.

2.4.4 Page Management Transitions in Brazos

These three adaptive mechanisms have been designed to work together. Figure 2.6 shows the transitions between various protocol states that a page may be in. Each state is determined by the adaptive protocol in force at the time:

- MCI - Multicast Invalidate is the base protocol for each page.

- MCEU - Multicast early update refers to the flushing of diffs before the arrival at a global synchronization point, as described in Section 2.4.1.

- MCHB - Multicast home-based refers to a page management scheme in which processes flush diffs to an appointed home-process at each release point.

Chapter 3 gives specific implementation details on all these adaptive techniques, and Chapter 4 examines their performance benefit in the Brazos software distributed shared memory system.
2.5 Maintaining Histories Across Execution Runs

Brazos attempts to preserve history information across program executions to "warm start" the adaptive mechanisms just described. This can be a difficult task, especially when the data access patterns of a particular program change with different input sets or system configurations. In order to make our history mechanism compatible with changing system parameters, we do not save information regarding how specific pages are accessed since the page->data mapping may easily change across executions. Instead, we save general threshold values for the adaptive mechanisms, as well as such information as the runtime system can glean relating to how specific variables or groups of variables are accessed during the course of a program's execution. This section provides a brief overview of the two types of history information saved by Brazos.

2.5.1 Saving Adaptive Thresholds

Brazos uses several threshold and hysteresis values during runtime to adapt the system's performance to the observed data access patterns. Values such as the number
of unused updates or invalidates to receive before deciding to drop out of the copyset, and the size of a diff that must be made before deciding to become a home process for a page are examples of these values. We save an average of the threshold values for all pages for each variable to provide an application-specific starting point for each adaptive mechanism. Additionally, Brazos detects when a particular adaptive protocol is detrimental to performance, and will restrict the protocol's use in the next execution.

2.5.2 Variable-Specific History

Because variable names and their relation to synchronization points do not generally change from execution to execution, information regarding the behavior of application variables can be saved efficiently. Brazos can recognize data access patterns for small variables (i.e., single-word variables or small arrays) from the virtual addresses being faulted on. Information regarding these variables' behavior is saved in the history file and loaded in at the next program execution. The Brazos runtime system constructs a variable$\leftrightarrow$page mapping based on information provided by the user, and the pages for which the runtime system has prior information can be set up accordingly. Section 4.3.5 gives performance-related information regarding the Brazos history mechanism.
Chapter 3

Implementation of Brazos

The majority of existing software DSM systems have been developed for use on high-end engineering workstations using the Unix operating system, mainly due to the higher performance of the operating system, architecture, and networks available on these machines. These DSM systems rely on the message passing facilities available under Unix to communicate between processes, and on the page protection features of the operating system to maintain a coherent view of data. There has been little work to date in the personal computer (PC) environment for large scientific applications. As a vehicle for this research we have developed Brazos, a software DSM system for use on x86 architectures under Windows NT.

Part of our research was to design a DSM system that allowed us to take advantage of the excellent price-to-performance ratio available in SMP personal computers. In deciding on an operating system, we took three main factors into account: low-level system performance, availability, and built-in support for multithreading. We initially compared three operating systems for PC’s: Windows NT 4.0, Linux, and FreeBSD. Of these three, FreeBSD has no built-in kernel support for SMP servers but does have excellent network performance. Linux provides SMP support, but the network performance is hampered by high OS overhead. Windows NT provides network performance comparable to FreeBSD, built-in support for SMP servers, excellent low-level system call performance, and a rich API for dealing with threading issues (thread priority, timing, suspending/resuming threads, etc.). Finally, there have been no attempts to date at designing a performance-oriented DSM system for Windows NT. Therefore, we chose to implement Brazos under the Windows NT operating system.
We also evaluated an x86 version of Solaris, which also provides SMP support, multithreading through the use of the Pthreads package, and excellent network communication latencies. In the following sections, we discuss the implementation and performance differences between a DSM system based on Solaris (e.g., TreadMarks), and a DSM system based on Windows NT.

3.1 Windows NT and DSM

A DSM implementation under Windows NT differs substantially from its Unix counterpart. The major differences that directly affect DSM system design and performance include the lack of BSD-style signals, structured exception handling, and the use of the WinSock communication library. This section describes these differences and provides performance comparisons to Solaris for x86 architectures.

3.1.1 Signals

Unix makes use of *signals* to inform the operating system of asynchronous events that must be handled. When a signal is delivered to a process, the process jumps to a *signal handler*, which can be either user-defined or the default handler provided by the OS. Signals that are used extensively in software DSM systems include SIGIO, indicating that an I/O operation is possible on a file descriptor, and SIGALARM, used as a timing device. The SIGIO signal is generally used to notify a Unix DSM system that an asynchronous message from another process is waiting to be received. An asynchronous message on any socket causes this function to be invoked immediately, unless the user has explicitly blocked the delivery of the signal. The `select()` function must then be used to figure out which socket has available data, and what should be done with it.

Windows NT provides a mechanism to notify threads of asynchronous events at a finer granularity than Unix. When a socket is made asynchronous, Windows NT ties the function that will handle any incoming asynchronous messages on the socket
to the socket instance itself. This allows asynchronous messages to invoke different handler routines depending on the socket on which they are received. Additionally, the receipt of an asynchronous message does not automatically halt other threads (i.e., application threads). This allows multiple, independent threads to process incoming messages concurrently while still allowing computation to proceed, within scheduling guidelines.

### 3.1.2 Structured Exception Handling

In a page-based DSM system, coherence is maintained by setting virtual page protection attributes to indicate the access permissions for a specific shared page. Processes "invalidate" pages in memory by altering the page protection such that any attempt to read or write to the page causes a segmentation fault. Unix-based DSM systems typically install their own interrupt handler that specifies a function to be called when a page-access violation occurs. Requests for page updates are then sent to other processes in the system to bring the invalid page up to date, after which the user program can continue.

Windows NT accomplishes exception handling through a mechanism known as structured exception handling (SEH). SEH allows a greater amount of control over how an exception is to be handled, and also allows exception handlers to be nested. Instead of installing a separate handler for each exception, as is done in Unix, Windows NT implements SEH by bracketing code that users wish to trap exceptions for with the keywords _try and _except().

The _except() keyword takes a function name as an argument, and this function is called when an exception is encountered within the try-except block. The decision is then made whether to handle the exception, pass the exception up to the operating system for handling, or continue execution without addressing the exception. In Brazos, a try-except block is placed around each application thread to catch access violation exceptions arising from accessing pages that have been invalidated. The
following code fragment shows how this is done:

```c
__try {
    UserMain(global_tid, local_tid);
} __except(DSM_AccvioHandler((GetExceptionInfo())->ExceptionRecord) {})"
```

Figure 3.1: Use of try-except Block in Brazos

The function UserMain() is the entry point into application code, and the function DSM_AccvioHandler() is responsible for handling faults caused by accesses to invalid pages. This is transparent to the programmer and does not necessitate altering application code.

Because try-except blocks can be nested, we place another try-except block around the entire DSM subsystem code in order to catch real segmentation violations (e.g., errors, not accesses to invalid pages) that occur within either application code or DSM code. This is in contrast to a Unix DSM system, in which the user-installed segv fault handler is called no matter where in the code the fault occurs. This makes it difficult to indicate to the user whether the fault happened in application code or DSM system code.

Table 3.1 shows data comparing the performance of Windows NT and Solaris for two key performance parameters relating to the handling of segmentation faults: the time needed to change the protection on a virtual page, and the time required to handle a segmentation violation. As Table 3.1 indicates, Windows NT and Solaris take similar amounts of time to protect a page of virtual memory (6.54 μsec and 7.0 μsec, respectively), but SEH under Windows NT takes less than half the time to handle a segmentation fault than the signal mechanism used in Solaris. In practice, however, only those programs with very high access violation rates will see any performance improvement from the faster try-except block in Windows NT.
Table 3.1: Basic System Timings

<table>
<thead>
<tr>
<th></th>
<th>Windows NT</th>
<th>Solaris</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Protect</td>
<td>7.0 μsec</td>
<td>6.54 μsec</td>
</tr>
<tr>
<td>Segv Handler</td>
<td>20 μsec</td>
<td>45 μsec</td>
</tr>
</tbody>
</table>

3.1.3 WinSock

Windows NT uses the Windows Sockets Programming API as implemented in the WinSock runtime library for communication with other processes. Although many of the calls in WinSock mirror their BSD counterparts, differences do exist. In particular, WinSock is implemented as a dynamic linked-library, and all WinSock functions are implemented in this DLL. In Berkeley sockets, some calls are direct system calls into the operating system, while other calls are made to functions in a static library linked at compile-time. Consequently, there is more overhead associated with many of the WinSock calls than with their BSD counterparts, which can result in higher per-message overheads and lower overall throughput.

To demonstrate this, Figure 3.2 shows the average throughput achievable between two servers connected by 100 Mbps FastEthernet. The graph shows throughput for Windows NT/WinSock and Solaris for PCs. This experiment was carried out on the same hardware to rule out any performance variations due to architectural differences. The test being conducted is the same request-reply sequence as that shown in Figure 1.1. Figure 3.2 shows that for small messages (less than 8192 bytes), the higher overhead in WinSock calls means that the maximum achievable throughput for messages under Windows NT is less than that of similarly sized messages under Solaris. For messages over 8 Kbytes, WinSock is able to stream data out more quickly than Solaris after paying the cost of initiating the message. The data in Figure 3.2 underscores the importance of both minimizing the number of messages that must be
sent to maintain coherence and maximizing the number of small messages that can be combined in order to lower the impact of the increased overhead under Windows NT.

3.1.4 Application Performance of Windows NT vs. Solaris

Table 3.2 shows 1- and 8-processor execution times and speedups for 10 of the applications used in Chapter 4 under both Windows NT and Solaris. Both of these implementations use versions of the TreadMarks DSM system, not Brazos, and were run on the same hardware to rule out runtime effects and architectural differences.

Columns 2 and 5 indicate that there is a wide disparity between uniprocessor execution times between Solaris and Windows NT. The reason for this disparity appears to lie in the differences between the two compilers used. The Solaris implementation
<table>
<thead>
<tr>
<th></th>
<th>Solaris</th>
<th></th>
<th>Windows NT</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1P Time</td>
<td>8P Time</td>
<td>Speedup</td>
<td>1P Time</td>
</tr>
<tr>
<td>Barnes</td>
<td>104.9</td>
<td>45.4</td>
<td>2.31</td>
<td>71.1</td>
</tr>
<tr>
<td>CG</td>
<td>22.4</td>
<td>18.0</td>
<td>1.24</td>
<td>15.3</td>
</tr>
<tr>
<td>FFT-3D</td>
<td>47.9</td>
<td>15.1</td>
<td>3.17</td>
<td>34.4</td>
</tr>
<tr>
<td>ILINK</td>
<td>77.2</td>
<td>31.8</td>
<td>2.43</td>
<td>97.9</td>
</tr>
<tr>
<td>LU</td>
<td>124.4</td>
<td>26.2</td>
<td>4.75</td>
<td>111.9</td>
</tr>
<tr>
<td>MG</td>
<td>89.7</td>
<td>44.8</td>
<td>.9</td>
<td>83.0</td>
</tr>
<tr>
<td>Ocean</td>
<td>4.3</td>
<td>11.0</td>
<td>.39</td>
<td>4.9</td>
</tr>
<tr>
<td>SOR</td>
<td>69.6</td>
<td>13.9</td>
<td>5.0</td>
<td>81.2</td>
</tr>
<tr>
<td>TSP</td>
<td>53.9</td>
<td>10.3</td>
<td>5.24</td>
<td>64.8</td>
</tr>
<tr>
<td>Water</td>
<td>135.1</td>
<td>29.8</td>
<td>4.54</td>
<td>71.8</td>
</tr>
</tbody>
</table>

Table 3.2: Comparison of TreadMarks on Solaris and NT

uses the gcc compiler, while the Windows NT implementation uses the Microsoft cl compiler. The latest available gcc compiler for x86 architectures does not make full use of the x86 instruction set. In particular, math functions such as square root and sine have the corresponding hardware instructions FSQRT and FSIN, which the Microsoft compiler makes use of. However, gcc does not use these instructions; rather, gcc inserts calls to the standard C math library, greatly increasing the time required to perform these functions. As a result, programs such as FFT-3D and Water that make frequent use of simple math functions tend to run faster under Windows NT.

The second compiler difference stems from data layout. In general, code generated by Version 4.0 of the Microsoft compiler results in many more mis-aligned data accesses than code generated by gcc, resulting in higher average memory latencies and slower execution [43]. Thus, some of our programs such as SOR and TSP run slower under Windows NT than Solaris.

Looking at the speedups achieved, it is difficult to make a direct comparison because of the differences in the uniprocessor cases just pointed out. With the exception of ILINK, the system with the higher uniprocessor time achieves the better speedup.
Section 3.1.3 showed that Solaris achieves a higher network throughput than Windows NT for most message sizes. As a result, for a DSM system such as TreadMarks, Solaris outperforms Windows NT in terms of speedup for all applications except SOR and TSP, which exchange very little data during execution. However, when we turn off optimizations, the sequential running times are much more closely matched. In this case, we see roughly the same performance between the two implementations. Therefore, in general we believe Windows NT to be competitive with Solaris when utilizing the same hardware and DSM system. Chapter 4 will show how Brazos provides superior performance to TreadMarks under either Solaris or Windows NT.

3.2 Programming for Brazos

Users write programs for Brazos utilizing the shared memory parallel programming paradigm and link with the static library brazos.lib at compile-time. Brazos provides an implementation of the PARMACS [12] macro suite for ease of porting between shared memory systems.

Instead of including a main() function, users specify the function UserMain() as the entry-point to a Brazos parallel application. The program's main() function resides in the Brazos library. The Brazos DSM system will spawn as many application threads as specified by the user, and each one will invoke UserMain() with two arguments: the thread's global identification number (unique among all threads in the DSM system), and the thread's local identification number (unique among all threads running in the current process). We provide the second value (the local id number) because users must follow conventions for programming in a multithreaded environment; namely, changes made to global variables* by one thread in a process

*When discussing multithreaded shared memory, confusion often erupts over the terms "global" and "shared". For the rest of this thesis, "global" will be used to refer to variables that are visible to all threads in a process, but not to threads in a different process. These types of variables include those declared outside any function, and static variables declared inside or outside a function's scope.
will be visible to all threads. The majority of existing shared memory applications currently do not follow this convention, but expect each thread of execution to be running in its own address space. The local identification number can be used to create arrays of global variables that can be indexed by a thread’s local id.

3.3 Components of the Brazos System

The Brazos system consists of three\(^1\) main parts: a library that is linked at compile time to application code; a service that must be run on every machine that will be used to run DSM applications; and a graphical user interface that can be used to monitor, start, and stop DSM applications. Figure 3.3 shows these three components (along with the optional device driver to map two virtual addresses to the same physical address) with arrows to indicate the interaction between them. In the following sections we discuss the specific interaction of these components, as well as detailed information on the implementation of Brazos.

3.3.1 Brazos Graphical User Interface

The Brazos graphical user interface, shown in Figure 3.4, provides a way for users to manage multiple DSM sessions in a coherent manner. The Brazos GUI requires a password for each user wishing to run a DSM application on the system. This password is validated by the remote Brazos service before beginning a program’s execution.

The Brazos GUI also provides feedback on the performance of each DSM session currently under its control by snooping the multicast traffic on the network. Number

\(^{\text{1}}\)We originally implemented Brazos with a kernel mode device driver that allowed us to map two virtual addresses to the same physical page. Recently, we have discovered a way to accomplish this without involving a device driver.
of bytes per second, synchronization operations per second, and other statistics can be viewed by the user while the program is executing.

To begin a Brazos program execution, the user creates a Brazos configuration file through the Brazos GUI. Alternately, a previously defined configuration file can be used. The configuration file specifies the parameters of a DSM run including the machines to use, the number of application threads to start on each machine, the executable to be used, the priorities each process should have, and other options for the program execution. The Brazos GUI also provides a probe function that can be
used to determine the lowest-loaded remote servers on which DSM sessions can be started. After the user has defined the configuration for the Brazos session, the Brazos GUI contacts the Brazos service located on the user-designated root machine. This may be the local server or a remote server. As described in the next session, the service located on the root server spawns a process to execute the specified Brazos DSM program. Messages the user wishes to print to the output are sent via a dedicated socket from the root application process to the Brazos GUI, where they are displayed in a separate window.

Finally, the GUI provides the interface for interacting with features of the remote Brazos services. Through the GUI, users can send messages to query, start, and stop DSM sessions as outlined below.
3.3.2 Brazos Service

Windows NT implements a variety of functions in services. Services are trusted applications that are given access and privileges not normally available to user programs, and serve many of the same functions as Unix daemons. Services are started at boot-time, and implement things such as the at scheduler (analogous to the cron daemon in Unix), telnet, and ftp. Many common Unix functions are not provided in the standard release of Windows NT. Examples of these include reexec, rlogin, and telnet. DSM systems require a method of starting an application on a remote machine. Therefore, we developed the Brazos service to provide this functionality. The functionality includes:

1. Resource Management - The service is responsible for ensuring that the local server does not become overloaded with DSM sessions. A user-defined maximum number of DSM sessions can be specified when the service is first installed. In addition, the Brazos GUI allows a privileged user to change the maximum number of DSM sessions from a remote process.

2. Query Resource Availability - Users may query a remote service from a Brazos GUI as to the service's ability to accept another DSM application.

3. Starting Remote Processes - Through the GUI, users can request that a remote service start a DSM application on the service's local machine. This functionality mirrors that of the rexec command in Unix.

4. Terminating Remote Processes - Users may also send a message to the service requesting that a specific DSM application be terminated before completion. This is useful for stopping runaway DSM processes without having to be logged into the remote computer's console. There is normally no method for terminating a remote process under Windows NT.
When a Brazos GUI contacts the Brazos service to begin a new DSM program, the service first checks to see if the system can accommodate the request. If the service can accommodate the request, the specified executable is started. This executable will then interact with other remote DSM services to start another instance of itself on remote machines, differing only in the argument given each instance specifying its unique process number. From this point on, the DSM processes communicate with each other directly without intervention from the Brazos service.

3.3.3 Multithreaded Page Access Protection

In a multithreaded DSM system, it is often necessary to allow one thread to modify a page of shared data while restricting access to the page by another thread. For instance, when the DSM system reply thread receives a diff that must be applied to a page of data, application threads should not be allowed to access that page until the update is completed, or they may read stale data.

One solution to this problem is to suspend all threads that may access incorrect data when shared data on a page needs to be changed atomically. This can incur a severe performance penalty, especially if the suspended threads would not have accessed the page in question.

A better approach is to provide two virtual addresses that map to the same physical page, as can be done with the `mmap()` call in Unix. We originally modified an existing device driver to perform this task. At the beginning of a program's execution, the DSM system invoked this device driver to provide an alternate set of virtual addresses for each page of shared data. These addresses were used by the DSM system threads only. After a system thread finished updating a shared page through these alternate virtual addresses, the page protection on the original virtual address is changed from NOACCESS to READONLY or READWRITE to allow application threads to access the data.

In the current Brazos implementation, DSM and application threads map the same
region of address space to two virtual addresses through the `MapViewOfFile()` call, originally intended as a way for processes to share data.

3.4 Brazos Runtime System

This section provides implementation details on the runtime aspects of the Brazos DSM system. Figure 3.5 depicts the program flow of the main thread of every Brazos program. The green boxes indicate elements executed by both the root and remote DSM processes, the orange boxes show root-only code, and the blue boxes show remote-only code. Except for startup, there is no difference between the execution paths taken by the root and remote processes. We discuss each portion of Figure 3.5 in the following sections.

3.4.1 Initialization

The end of the initialization phase is indicated by the horizontal black line in Figure 3.5. The main phases of initialization are DSM runtime startup initialization, network initialization, and thread creation.

Brazos Runtime Startup Initialization

The startup initialization phase covers the first two green boxes at the top of Figure 3.5. During this phase of the execution, application code and DSM program arguments are parsed, and the user-specified configuration file is read and appropriate variables set. Startup values for adaptive mechanism thresholds that may have been gleaned from previous executions of the program are also read in from the configuration file, and the appropriate global variables are set.

The next phase allocates and initializes the DSM runtime data structures. There are four main data structures in Brazos:

- Shared Page Structure. A page structure keeps information relating to each
shared page of data presently allocated in the system. The page's virtual address, current copyset and page management model, and the threshold values for use in the adaptive tuning mechanisms are examples of the information maintained in this structure. The root process begins with all pages valid and exclusively held. Slave processes initially protect the entire shared address space, forcing them to retrieve an initial page from the page manager when the process first accesses the page.

- Diff Structure. Diff structures keep information such as the diff type, diff size.
and diff data, and are associated with a page structure. A pointer from the page structure provides access to the diff, as shown in Figure 3.6. Brazos uses two different diff structures: a runlength encoding whose format is that of

\(<\# \text{ bytes in diff}<\text{ offset into page}<\text{ data}<\text{ data+1}\>\ldots<0>\)

and an address-value pair whose format is

\(<\text{ offset into page}<\text{ data}<\text{ offset into page}\>\ldots<\text{ data}<\text{-1}>\)

In general, the runlength encoding format requires less storage overhead, whereas diffs of the address-value pair are more easily merged together. Diffs in Brazos begin as runlength encoded diffs, and are converted to address-value pair diffs before being merged with other diffs. We merge diffs to save on overall storage requirements. Because diffs are made on a word-length basis, users must avoid allocating shared data at a granularity less than that of the size of a word.

![Diagram of page structure and shared page array](image)

Figure 3.6: Relationship Between Page and Diff Structures in Brazos
- Lock Structure. Lock structures keep information about each application-level lock, including current lock holder, the lock manager, and diffs for pages modified within a critical section delimited by the lock.

- Barrier Structure. Barrier structures maintain the statically-assigned barrier manager's process number, the number of processes currently at the barrier, and a Windows NT mutex variable to implement a local barrier for threads located in the same process.

Network Initialization

During the network initialization phase, processes allocate communication sockets and join multicast groups. There are only two sockets needed in each Brazos execution: one for sending asynchronous requests and receiving replies to them, and one for receiving asynchronous requests and responding to them. Processes bind each of these sockets to an arbitrary port number selected by the WinSock communication stack, and begin registering to receive multicast packets sent to specific multicast addresses on each port.

A base multicast address for the DSM session is chosen by the service on the root server (e.g., “234.5.6” in Figure 3.7). Processes then register to receive multicast packets sent on addresses which have the bit corresponding to their process id set in the last 8 bits of the address, as shown in Figure 3.7. This registration is for multicast receiving only since nothing special has to be done to send on any desired multicast address. The combination of the two port numbers with the \(2^n - 2\) addresses needed are saved in an array of remote addresses. Then it is a simple matter for a process to use a byte-length bit mask (i.e., the copyset of a specific page) to send to whatever subset of possible processes is desired. Note that the multicast addresses ending with 0 and \(2^n - 1\) are not used because processes do not send to the empty set, nor do they send to a set of processes that includes themselves.

The next step is to establish communication with other processes. As indicated by
base_addr = "234.5.6"
for(i = 1; i < (1 << NumNodes) - 1; i++) {
    sprintf(buf, "%s.%d", base_addr, i);
    addr.sin_addr.s_addr = inet_addr(buf);
    if(i & (1 << MyNodeId))
        WSAJoinLeaf(Socket, addr, ...)
}

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
<th>Process 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>234.5.6.1</td>
<td>234.5.6.2</td>
<td>234.5.6.4</td>
<td>234.5.6.8</td>
</tr>
<tr>
<td>234.5.6.3</td>
<td>234.5.6.3</td>
<td>234.5.6.5</td>
<td>234.5.6.9</td>
</tr>
<tr>
<td>234.5.6.5</td>
<td>234.5.6.6</td>
<td>234.5.6.6</td>
<td>234.5.6.10</td>
</tr>
<tr>
<td>234.5.6.7</td>
<td>234.5.6.7</td>
<td>234.5.6.7</td>
<td>234.5.6.11</td>
</tr>
<tr>
<td>234.5.6.9</td>
<td>234.5.6.10</td>
<td>234.5.6.12</td>
<td>234.5.6.12</td>
</tr>
<tr>
<td>234.5.6.11</td>
<td>234.5.6.11</td>
<td>234.5.6.13</td>
<td>234.5.6.13</td>
</tr>
<tr>
<td>234.5.6.13</td>
<td>234.5.6.14</td>
<td>234.5.6.14</td>
<td>234.5.6.14</td>
</tr>
</tbody>
</table>

Figure 3.7: Assigning Multicast Addresses to Communication Sockets

the orange and blue boxes in Figure 3.5, this is the only point in a program's execution where the root process behaves differently than the remote processes. The root process starts remote processes by contacting the services on the remote machines, which in turn spawn a process to run the specified executable. The newly-created process sends a message to the root process indicating that it has been successfully started, and specifying the port numbers to use for communication with it. After the root process has received start-up messages from all remote processes, a single multicast message is sent to indicate that all processes should continue to the thread creation portion of program execution.

DSM System Thread Creation

In the final initialization phase, each process creates the threads needed for the DSM system and user-code. Two DSM system threads are created first: one to receive asynchronous requests for data from remote processes, and one to receive replies to
requests previously sent out by the local process.

**Asynchronous Request Thread** The request thread is responsible for quickly responding to asynchronous messages from other processes. This thread waits for an incoming request from another process and routes the request to the appropriate request handler. The asynchronous request thread also uses the overlapped-IO features in Windows NT. The thread posts \( N \) receive buffers to the OS, where \( N \) is the number of remote threads that could possibly send a request. Additionally, an NT event is specified for each buffer. Threads will block on an event if it has been reset, and they will wake up and proceed when the event becomes signaled or set. The asynchronous request thread initially enters a wait state by calling the NT system call `WaitForMultipleObjects`, specifying the array of events associated with the buffers as the objects for which to wait. When a message arrives, the buffer's event is signaled, and the thread is re-scheduled and processes the message. After the message has been processed, the buffer is posted again, and the thread re-enters the wait state for until the arrival of another incoming message. An advantage to overlapped IO is that the OS will route incoming messages directly into the user-supplied buffers. This avoids a copy that usually occurs from internal OS buffers to user space by transferring contents of the message directly from network buffers to the supplied application-level buffers.

Because Brazos only provides for one asynchronous request to be handled at any given time, we have found performance to be best when this thread runs at the highest possible thread priority in order to service requests quickly (see Section 4.3.6).

**Reply Thread** The second DSM system thread handles replies to requests previously sent by application threads in the process, as shown in Figure 3.8. After receiving a reply, this thread checks to be sure the message has the correct sequence number for the application thread that issued the original request. If the sequence number is wrong, the message is dropped. This usually only happens when a retrans-
mission has occurred, and the first message has already been handled. Otherwise, the appropriate reply handler is called. Finally, the reply thread signals the application thread waiting for this reply, indicating successful receipt of the data requested.

![Flow Chart](image)

**Figure 3.8 : Reply Thread Flow Chart**

The alternative to allocating a thread solely for waiting for replies is to have the application thread that issued the original request simply perform a blocking receive operation until the response returns. However, because Brazos allows multiple application threads to have simultaneous outstanding requests, a thread that blocks on a receive will unblock when the first data on the socket is made available, whether the incoming message is intended for that thread or another thread. If the thread has received a response to a request it did not issue, either the message must be dropped, in which case the sending process will retry and hopefully the correct thread will receive it the next time, or some mechanism for communicating between application threads must be implemented. The first case clearly will result in poor performance, and the second is likely to be inefficient because it requires additional synchronization between application threads in the same process. Additionally, indirect multicast diffs are received by the reply thread, something that would also be complicated without a dedicated reply handling thread.
Application-Level Threads

Coherence between application threads in the same process is maintained through the available hardware coherence mechanisms. Synchronization between these threads is provided by embedding native Windows NT synchronization commands inside the Brazos synchronization primitives. This allows the same executable to run either in one process as a shared memory program, or between processes as a DSM program without any changes to the application-level code.

Brazos provides interfaces for the user to specify a wide variety of options for application-level threads. These include pinning threads to a subset of available processors; providing for a dedicated protocol processor that does not execute application code but handles all communication with other processes; and setting relative thread priorities. Additionally, the entire set of Win32 API threading functions is available for programmers to use on Brazos application-level threads. Section 4.3.6 addresses the performance impact of some of these options.

3.4.2 Execution

After the application-level threads have been created, the original DSM thread of execution waits for the end of the program, which is indicated by the exiting of the application threads just created. After the application threads have exited, data structures are de-allocated, WinSock is shut down, and the program exits.

User Program Flow

Application-level threads interact with the DSM system at three points in a parallel program: on a page-access violation, during synchronization, and when an application thread wishes to output information to the user. Figure 3.9 shows the execution path of an application-level thread in Brazos as it interacts with the page-access violation handler and synchronization primitives.

When an access violation occurs, the application thread enters a handler routine
in the DSM system library. At this point, the thread must acquire two internal DSM locks: a page-specific mutex that ensures that multiple application threads do not send requests for updates to the same page simultaneously, and a lock to protect internal DSM data structures from other application threads and DSM system threads. The application thread then checks to see if the page is marked as “valid” in the page structure. A thread will fault on a valid page at the first write to the page. When this occurs, a twin is made of the page (the page is copied and its contents saved for later comparison with the modified page [14]), the protection is changed to read/write, and the application thread is allowed to continue. The twin will be later used to determine the words in the page that have been modified in order to update other processes in the system. If the page is not valid upon entering the access violation handler, the thread must communicate with other processes in order to bring its copy of the page
up-to-date. If the miss is a cold miss (e.g., the page has never been accessed before by this process), the thread must first request the entire page from the current page manager. After the page is received from the manager, or if the miss resulted from the page having been invalidated previously in the program, the thread sends a single multicast request to the current invset for the page in order to obtain diffs from other processes. The invset is a bit-mask maintained by each process for every shared page in the system, and indicates which processes have dirty pieces of a page. The invset for each invalid page is received by the process at a barrier release. When the thread receives the appropriate diffs from other processes, the diffs are applied to the page to bring it up to date with respect to the other processes in the system. After this, a twin is made if the miss occurred on a write access, and the handler exits, allowing computation to proceed.

**Barriers**

Barriers are used by the programmer to synchronize all threads. Each barrier instance has a statically assigned barrier manager associated with it. Threads arriving at a barrier must go through two levels of barrier synchronization: a local barrier for all threads in the process, and a global barrier for all processes in the system. Only the last thread to arrive at a local barrier will interact with other processes in the system.

The last thread in the process to arrive at the barrier sends a message to the barrier manager, indicating that all threads in the process have arrived at the barrier. This message is divided into five sections, with each section providing information to the manager on pages written, current copysets for each page, and information used by the adaptive mechanisms. Specifically, the barrier arrival message indicates:

- **Dirty Pages.** A list of pages that have been written to since the last barrier arrival.

- **Copyset Changes.** The manager of each page sends the new copyset for each page whose copyset has changed since the last barrier arrival. Because Brazos
uses the copyset to respond to diff requests and depends on all processes seeing
the response for correctness, it is important that all processes always have the
correct copyset for each page.

- **New Early Update Pages.** Each process includes in the barrier arrival mes-
  sage a list of pages that it has detected should be placed on the global early
  update list. The manager then distributes this list to all processes at the barrier
  release.

- **New Home-Based Pages.** Similarly, if a process believes that it has written
to enough of a page to qualify as the “home process” for the page. this is also
  indicated in the barrier arrival.

- **Copyset Removal Requests.** When a process decides that it must remove
  itself from the current copyset, the manager is informed in this section of the
  barrier arrival message. The new copyset will then be distributed to other
  processes at the barrier release.

The barrier manager collates information from each other process in the system
with information from the local process. and sends a single multicast message to
release processes from the barrier.

In terms of coherence, barriers in Brazos behave similarly to barriers in an eager
or lazy release consistent memory model. Specifically, all global data in the system is
guaranteed to be coherent after processes leave a barrier instance. Thus. the use of
scope consistency in Brazos makes no difference in programs that use only barriers for
synchronization. Differences in performance for barrier-only programs results from
the use of multicast and adaptive performance techniques.

**Locks**

Locks are used to synchronize accesses to shared data between two processes. Users
can make calls to the Brazos-provided lock primitives without regard as to where
the synchronizing processors are located (i.e., they may be in the same process or
different processes), as Windows NT synchronization calls are embedded within the
Brazos call. The lock mechanism used is a simple queue-based locking scheme, similar
to that used in the Munin [13] and TreadMarks [28] systems.

Each lock is statically assigned a lock manager. All processes initially request the
lock from the manager, which forwards the request to the last requester of the lock.
Each process keeps track of the next process after itself to request the lock, thereby
implementing a distributed FIFO queue of processes waiting for the lock.

Brazos provides three different lock semantics. The first uses locks to implement
scope consistency as described in Section 2.3. When using scope consistency, users
must ensure that no race conditions exist between data accessed on either side of a
critical section, since only changes made within the critical section are transferred to
the next lock acquirer.

To implement scope consistency, we use user-annotated critical section writes.
Users replace writes to shared data within a critical section with the macro CRIT-
ICAL SECTION WRITE and specify the variable being written and the new
value. This allows the Brazos system to keep what was written inside a critical sec-
tion separate from changes to data made outside a critical section. The Brazos system
attaches the value and address written inside each critical section to the DSM system
lock structure for the lock currently held by the writing thread. This information is
then passed to the next lock acquirer with the lock grant. Thus, the use of the scope
consistency lock macros LOCK and UNLOCK are best suited for small critical
sections that may introduce unnecessary false sharing.

Some applications may have large or complicated critical sections, or have data
dependences across critical sections that can not be easily removed (e.g., the TSP
application used in Chapter 4). For these instances, Brazos provides two other lock
semantics: an eager release-style lock mechanism, and an lazy update release mech-
anism. The eager release macros RCLOCK and RCUNLOCK work as locks do
in Munin: on each RCUNLOCK, all pending write notifications are flushed to all processes prior to allowing the release to complete. The macros LRCLOCK and LRCUNLOCK provide mechanisms similar to lazy update release locks, with one important difference. When a process sends a lock grant to another requesting process, the response is multicast to all processes. Included in this message are updates for all pages the providing process has modified since the last synchronization event. In true LRC, updates are only sent to the next acquirer. However, because we do not permit processes to forward diffs from other processes, we must multicast the updates.

It should be noted that all information transferred between processes at a barrier point is also exchanged when an application employs the two release consistent lock mechanisms just described. This is because these mechanisms are also global synchronization points, as all processes are affected when they are executed.

3.4.3 Adaptive Protocol Implementation

In this section we describe the implementation details of the adaptive performance tuning protocols used in the Brazos DSM system. The first two of these protocols deal with side-effects of employing a multicast protocol: extraneous multicast diffs and extra network communication. The third protocol addresses the fact that different pages in a single DSM application may benefit more from a specific page management algorithm, and allows a page to switch between a distributed page management algorithm and a home-based protocol. A breakdown of the added overheads for each of the three adaptive techniques is presented at the end of this section.

Dynamic Copyset Reduction

As described in Section 2.4.2, processes may continue to receive diffs for a page that will not be used before the page next gets invalidated or updated. By causing the reply threads to be scheduled, receive these diffs, and apply them to the page, compu-
tation time is taken away from application threads. We have therefore implemented a dynamic copyset reduction mechanism to allow processes to drop out of the copyset for pages they are not actively using.

We implement this mechanism by associating a counter with each page of shared data. When a shared page is brought up-to-date through the receipt of indirect diffs, this counter is reset to an application-specific threshold determined by a recorded history of past executions. In order to detect whether a process uses this page, the page protection is augmented with a PAGE_GUARD status flag. The guard flag causes the thread to take a guard page exception the next time the page is accessed. When the exception occurs, the OS immediately removes the guard status for the page. In this way, guard pages act as a "one-shot" alarm. Inside the handler for the page guard exception, Brazos notes that the page was accessed and resets the counter associated with that page. Conversely, when the page is next invalidated, the current protection of the page is checked. If the guard flag is still set, the page has not been used, and the counter is decremented by 1. When the counter reaches 0, the process will remove itself from the current copyset of the page to avoid receiving more unused indirect diffs.

If a page previously dropped from the copyset is accessed again by a thread in the local process, the access is treated as "cold miss" as described in Section 3.4.1. However, the counter for that page is set to the current threshold for the page plus a hysteresis factor. The hysteresis factor means that more unused indirect diffs must be received than last time in order for the page to drop from the copyset. We include this factor because retrieving a page across the network due to a premature drop is much more expensive than continuing to receive unused diffs.

Early Update Mechanism

We have implemented an early update mechanism to reduce the effects of multicast conflicts. As described in Section 2.4.1, a process may receive indirect multicast
diffs while waiting for a reply for diffs to the same page. This increases network communication because the request and the replies are unnecessary.

A Brazos process will place a page on the early update list if an indirect diff is received and the page mutex variable discussed in Section 3.4.1 is held by an application thread. If the page mutex is held, it indicates that a thread has already faulted on the page and is in the process of requesting changes to the page. Processes send their list of new early update pages to the barrier manager at a barrier arrival, and the manager distributes the new list of early update pages to all processes at the barrier release. Thereafter, processes will flush all changes they have made to early update pages to other processes immediately before arriving at a barrier, eliminating further multicast conflicts and access violations. Additionally, early updates are sent en masse by each process, making better use of the available network bandwidth by combining diff messages into one message.

Keeping pages on the early update list too long can have two detrimental effects. First, continuing to place pages on the early update list can cause the size of the pre-barrier early update flush to exceed the maximum allowable message size (64 Kbytes under WinSock UDP). This will result in multiple rounds being necessary for each process at a barrier arrival, severely hampering performance. To address this, if a process is required to send multiple update messages before reaching a barrier to flush all updates, the update list is emptied, and all pages revert back to the standard multicast invalidation protocol. In practice, this does not occur frequently.

The second detriment to performance occurs from the overhead of making the early update diffs. Consider a situation where the sharing patterns change during a program's execution, and only one process is writing to a page of data. Even if this data is not used by other processes, the data will continue to be unnecessarily flushed at each barrier, increasing both network usage and diff creation time. Therefore, we use the dynamic copyset reduction mechanism described above to allow processes to drop from a copyset for an early update page for which the updates are not used. We
again make use of the guard flag to accomplish this by guarding updated pages and checking the guard status when the next update for the page arrives. When there is only a single process left in the copyset, the page will be taken off the early update list, and the page will revert back to the invalidation protocol. Pages that are put back on the early update list will have a hysteresis factor added to the new threshold, because similar to the receiving of useless indirect diffs, receiving too many early updates is less expensive than fetching an entire page of data across the network.

**Adaptive Page Management Mechanism**

The final adaptive protocol used in Brazos is an adaptive page management mechanism. Some previous studies have shown that distributed page based systems outperform home-based page systems [27], while yet other studies have argued that home-based page systems offer superior performance [53]. We have elected to provide an adaptive mechanism whereby those pages better suited to a home-based management technique can switch from the default distributed algorithm.

To determine when a page should change page management mechanisms, processes continually monitor the size of the diffs being created in response to requests for data. When this size reaches a certain threshold, which we conservatively choose as \((\frac{3}{4} \times \text{PageSize})\), the process will request that it be made the home process for the page. Thereafter, processes will flush changes made to this page to the new home process before each release operation. When another process misses on the page, the entire up-to-date page is retrieved from the home process.

The adaptive page management algorithm closely resembles the early update mechanism just described. However, in the situation where one process writes to most or all of an entire page, and other processes simply read from the page, the ability to change to a home-based protocol eliminates the overhead of the writing process having to create and flush a large diff. Instead, processes that need the page simply retrieve the entire page from the home process.
There are two ways in which a process may cease to be the page manager for a home-based page. First, if a home process notices that a diff flushed to it by another process exceeds the threshold, the home process will transfer ownership of the page to the process that sent the large diff. Second, the home process will periodically make a diff for each page that it manages. If the diffs turn out to be consistently small, the page will revert back to a distributed page management protocol.

**Overheads Associated with Adaptive Protocols**

There are several sources of overhead associated with the use of the three adaptive protocols described above. These overheads can be broken down into four areas: storage requirements, additional access violation processing, extra page protection operations, and communication. Table 3.3 shows the overheads for each of the four protocols.

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Dynamic Copysets</th>
<th>Early Updates</th>
<th>Adaptive Page Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extra Storage Required (per shared page)</td>
<td>16 bytes</td>
<td>28 bytes</td>
<td>12 bytes</td>
</tr>
<tr>
<td>Additional Access Violation Handling</td>
<td>1 segv each used page</td>
<td>1 segv each used page</td>
<td>–</td>
</tr>
<tr>
<td>Extra Page Protections (N receiving processes)</td>
<td>–</td>
<td>N per updated page</td>
<td>1 per flushed page</td>
</tr>
<tr>
<td>Network Communication</td>
<td>–</td>
<td>1 msg per update</td>
<td>1 msg per flush</td>
</tr>
</tbody>
</table>

Table 3.3: Added Overheads of Adaptive Protocols

It is difficult to assess the performance impact on the first overhead, extra storage requirements, because it does not directly affect a program's running time. Rather, in the limit, it can effect the maximum program size that can be executed by taking up memory that could otherwise be allocated as shared data. The three protocols use a total of 56 bytes of extra storage per shared page of data to keep track of
the thresholds, hysteresis factor, and performance counters associated with these mechanisms. We typically use 16,384 shared pages of data, yielding an overhead of 896 Kbytes for a 64 Mbyte shared address space, or a 1.3% overhead.

Two protocols add extra access violation processing because of the use of guard flags on virtual pages. The dynamic copyset reduction mechanism and the early update protocol each use guard flags to keep track of the success rate of the protocol. This allows the Brazos runtime system to alter the protocol's behavior. Each page that is used after either an early update or an indirect multicast diff to that page will incur an additional segmentation violation to handle the guard exception. Each such exception requires approximately 20 μsec to handle.

Adaptive page management and early updates require 1 and \( N \) extra page protection operations, respectively, where \( N \) is the number of processes that receive the flush or update. This is because both of these algorithms flush changes to shared pages at each release point in the program, after which the protections on each dirty page must be changed to \texttt{READONLY}. Additionally, in the early update protocol, the protection on each updated page must be changed to include the guard bit used to indicate if the updated page was used. Each page protection operation requires 7.0 μsec.

Finally, the early update and adaptive page management protocols require "eager" network communication that moves data earlier than the base invalidation with multicast protocol. At each release in the early update algorithm, each of \( N \) processes must flush updates for every modified shared page in a single multicast diff, where \( N \) is the number of processes with dirty portions of the pages in the message. These updates require \( P \) point-to-point acknowledgments, where \( P \) is the number of processes in the aggregate copyset of all pages in the update message. Realistically, programs that make use of the early update protocol end up with all processes having to flush at least one shared page's changes, and each process having to respond to all other processes, requiring \((\text{NumProcesses})^2\) messages. Similarly, the adaptive page
management protocol requires a single multicast message from $N$ processes who have dirty page portions of pages to the aggregate set of page managers. Each manager responds with one acknowledgment, typically resulting again with $(NumProcesses)^2$ messages.

Of the four overheads discussed above, the only overhead with a significant impact on performance is the additional network communication required by the early update and adaptive page management protocols. In Chapter 4, we examine how effective these protocols are at reducing the overall communication traffic.

### 3.5 Implementation of the Brazos History Mechanism

Brazos employs a history mechanism to retain performance-related data across program executions. As outlined in Section 2.5, the history mechanism saves information about each adaptive protocol on a variable-specific basis to allow past performance information to be used even when problem sizes or system configurations change.

#### 3.5.1 Collecting Variable Information at Runtime

Figure 3.10 shows how information on shared variables is collected and stored in Brazos. When users allocate regions of shared data, they must do so through the macro \texttt{G\_MALLOC}. As a second argument to this macro, the user passes a unique label, which the runtime system associates with the region of shared memory that is allocated as shown in Figure 3.10. Each entry in the page information structure contains a linked list of variable-related structures that contain details on all variables that reside in whole or in part on the page. In this way, the runtime system obtains information on the size, starting page, and user-provided label for all regions of shared data allocated by the user program.
3.5.2 Obtaining Variable-Specific Runtime Performance Data

The early update protocol is the only protocol that uses information regarding variable placement during a program's execution. Recall from Section 3.4.3 that a page is placed on the early update list when a process detects that it has received an indirect multicast diff while an application thread is blocked waiting for a response to a request for diffs for the same page. The addition of the history mechanism enhances the early update protocol in the following way: when a diff request is sent, the address on which the thread faulted is included in the request. The replying process then includes this address in the multicast diff that is sent to the page's current copyset. Therefore, when another process receives this indirect diff, it knows the address of the variable on which the original application thread faulted. By comparing this address with the
addresses of the local threads currently waiting for responses to diff requests, and combining this information with the variable information obtained as outlined above, it is possible to determine the exact relationship between the variables involved in the multicast conflict.

Depending on this relationship, the runtime system may choose to do one of three things in addition to placing the page on the current early update list:

1. **Place the entire variable on the “permanent” update list.** Brazos maintains two types of early updates, *temporary*, which do not remain on the update list across executions; and *permanent*, which will be “warm-started” as using the early update protocol at the beginning of the next program execution. Variables are immediately placed on the permanent update list if the conflicts occurred to the same variable on the page and the variable is less than one page in length.

2. **Evaluate the conflicts to the variable at program’s end.** If the runtime system detects multicast conflicts for a variable larger than a page, the percentage of pages that the variable covers is examined at the end of the program to see how many of these pages experience multicast conflicts. If this percentage is higher than a threshold value (we use 50% currently), the entire variable will be placed on the permanent list.

3. **Take no permanent action.** If the multicast conflict occurs to different variables on the same page, no action past placing the page on the temporary update list is taken.

At the end of the program execution, the list of variable structures associated with each page are searched to determine which variables should be saved to the configuration file as permanent update variables.

In addition to the early update mechanism’s construction of the permanent update list as described above, the information gleaned by the history mechanism is used to determine the thresholds used by the adaptive protocols on a variable-specific basis.
At the end of a program's execution, each process traverses through the list of shared variables. For each shared variable, the average of each threshold value on all pages containing the variable is determined. This average threshold is saved in the Brazos configuration file along with the label for the shared variable as originally passed in through the G_MALLOC macro. On the next execution, the process is reversed. Variables and the threshold values associated with them are read in from this file, and once the runtime system determines the page\rightleftharpoons variable mappings, the pages' initial thresholds are set as an average of the thresholds of the variables contained on the page. In this manner, the history mechanism can be used across program executions, even when the data input set and/or system configuration changes.
Chapter 4

Experimental Results

4.1 Experimental Platform

Our experimental platform consists of four Compaq Proliant 1500 personal computer workstations, each with two 200 MHz Pentium Pro processors and 192 MBytes of main memory. These four machines run Microsoft Windows NT 4.0, and are connected with 100 Mbps FastEthernet through a managed switch. We use the switch to isolate Brazos multicast traffic from the rest of the departmental network.

4.2 Applications

We use a variety of applications to evaluate the performance of Brazos, ranging from complex "real-world" applications to kernels of code used in a large number of shared memory applications. We have also included several applications that do not perform well on software DSM systems because we believe that an evaluation of DSM should include not only those applications that run well, but a representative of all types of shared memory applications. We have taken representative applications from three well-known benchmark suites (SPLASH, SPLASH-2, and NAS). Two other applications were used in previous DSM studies evaluating the Munin and TreadMarks systems. Finally, ILINK is a genetics application that was recently used to find the gene responsible for Parkinson's disease. Descriptions of the SPLASH-2 applications appear in [50, 44, 51, 49]; NAS benchmarks are outlined in [6]; the two SPLASH applications are described in [45]; the Munin programs are described in [14]; and a detailed description of ILINK can be found in [17]. Here, we briefly summarize the
applications studied. Table 4.1 gives relevant information about each of the twelve applications used.

<table>
<thead>
<tr>
<th>Synchronization</th>
<th>Shared Memory Size (KBytes)</th>
<th>Data Set</th>
<th>Sequential Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes Hut</td>
<td>barriers</td>
<td>5,378</td>
<td>32K bodies</td>
</tr>
<tr>
<td>CG</td>
<td>barriers</td>
<td>109</td>
<td>14,000 matrix</td>
</tr>
<tr>
<td>FFT-3D</td>
<td>barriers</td>
<td>24,576</td>
<td>$2^6 \times 2^7 \times 2^6$ array</td>
</tr>
<tr>
<td>ILINK</td>
<td>barriers</td>
<td>29,885</td>
<td>$clp$ input set</td>
</tr>
<tr>
<td>IS</td>
<td>barriers</td>
<td>64</td>
<td>$2^{23}$ ints, $B_{max} = 17$</td>
</tr>
<tr>
<td>LU</td>
<td>barriers</td>
<td>32,800</td>
<td>2048 matrix, $b = 32$</td>
</tr>
<tr>
<td>MG</td>
<td>barriers</td>
<td>39,525</td>
<td>$2^7 \times 2^7 \times 2^7$ grid</td>
</tr>
<tr>
<td>Ocean</td>
<td>barriers, locks</td>
<td>57,189</td>
<td>258 X 258 grid</td>
</tr>
<tr>
<td>Raytrace</td>
<td>barriers, locks</td>
<td>3,527</td>
<td>balls4 data set</td>
</tr>
<tr>
<td>SOR</td>
<td>barriers</td>
<td>16,432</td>
<td>2048 X 2048 matrix</td>
</tr>
<tr>
<td>TSP</td>
<td>locks</td>
<td>785</td>
<td>19 cities, $R = 12$</td>
</tr>
<tr>
<td>Water</td>
<td>barriers, locks</td>
<td>478</td>
<td>729 mols, 10 steps</td>
</tr>
</tbody>
</table>

Table 4.1: Program Characteristics

SPLASH-2 Applications

Ocean simulates eddy currents in an ocean basin, including effects of wind stress and the impact of friction with the ocean walls and floor. The implementation included in the SPLASH-2 distribution uses a red-black Gauss-Seidel multigrid equation solver to solve a set of spatial partial differential equations each time step. The multigrid solver acts on a hierarchy of grids during each step, with boundary values maintained in a padded array.

Processes stream through many different portions of the grid data structure in different phases of Ocean, resulting in very high communication rates and low overall performance on software DSM systems. Synchronization in Ocean is through locks, which protect single-write updates to shared variables in three places in the program; and barriers, which divide the different portions of the computation into 8 phases,
which are fully described in [45]. We ran the Ocean program with a grid size of 258 × 258.

**Raytrace** renders a three-dimensional scene by tracing light rays through each pixel in an image plane containing the scene to be rendered. When a ray encounters an object in the scene, it will reflect in unpredictable ways off the object and produced multiple new rays. This results in in a “ray tree” per pixel when each ray’s interaction with the object is recursively determined.

The data for the image is stored in a hierarchical uniform grid and is read-only. A distributed task queue is implemented to specify the work required by each thread, and task stealing is employed to maintain a good load balance. Raytrace uses only locks for synchronization, and these locks protect the only write-shared data in the program. We use the *balls4* input set.

**LU Decomposition** splits a matrix A into upper-triangular (U) and lower-triangular (L) matrices such that L * U = A. A triangular set of equations can then be trivially solved using L and U with forward substitution or back-substitution. The version of LU in the SPLASH-2 suite blocks the matrix into b × b size blocks, where b is the *blocking factor*. We use a blocking factor of 32 on a 2048 × 2048 element matrix, resulting in 4096 blocks. The data in each block is allocated contiguously to reduce false sharing and enhance temporal locality, and the blocks are evenly distributed among the available threads in a 2-D scatter decomposition as shown in Figure 4.1.

The basic algorithm in LU consists of factoring the current diagonal block, updating all perimeter blocks using the just calculated diagonal, and updating interior blocks using the corresponding perimeter blocks. This final step is the most computationally-intensive, involving a dense matrix multiplication of two blocks.

**Modified SPLASH Applications**

*Water* is an N-body molecular dynamics application that calculates forces and potentials over a user-specified number of time steps in a system of water molecules
Figure 4.1: LU Problem Decomposition

contained in a cubical box. During each time step, Newtonian equations of motion for water molecules are solved, and the inter-molecular and intra-molecular potentials are summed to provide the total potential of the system. In order to calculate all potentials, $\frac{n^2}{2}$ interactions would have to be evaluated each time step. To reduce this computation, an approximation is made that only evaluates interactions between particles that fall within a half-box length of each other (the spherical cutoff radius).

The program flow in each time step consists of the following: calculate predicted values of atomic variables, compute intra-molecular forces, compute inter-molecular forces, correct predicted values based on calculated forces, enforce boundary conditions imposed by the containing box, compute the kinetic energy of the system, and calculate and print the potential energy of the system. Nearly the entire execution time is taken up by the computation of the inter-molecular forces between molecules. Water has been slightly modified from the original release by combining multiple lock acquisitions into a single acquisition and accumulating force parameters into local variables before updating the global array. These changes reduce the communication
from that in the original implementation.

Synchronization in Water occurs at barriers between the force calculation stages and at the beginning and end of each time step. Additionally, each molecule is protected by a lock, which is necessary because the same molecule may be involved in multiple force calculations simultaneously. We run a 729-molecule execution for 10 time steps. We ignore the timings for the first iteration to remove cold-start behavior from our performance data.

**Barnes Hut** is a hierarchical N-body problem that simulates the interaction of a system of bodies in three dimensional space over a number of time-steps. The information on the bodies under study is maintained in an octree data structure, with the leaves maintaining information on each body, and the internal nodes representing space cells. In order to compute the forces on each body, the program must partially traverse the octree once for each body, resulting in highly unpredictable access patterns. We run Barnes Hut with an input of 32,768 particles.

Barnes Hut was locally modified by eliminating two critical sections. Instead of threads accumulating their particle information directly into shared global structures within these critical sections, computed values are accumulated into local variables, which are then used to update the global data structures.

**NAS Applications**

**Fast Fourier Transform - 3D** FFT-3D numerically solves a partial differential equation using forward and inverse fast fourier transforms. The input to the algorithm is a complex array $A$ of size $n_1 \times n_2 \times n_3$. The data is distributed such that for each $k$ and $l$, all elements of $(A_{j,k,l}, 0 \leq j < n_1)$ are owned by a single thread of execution. A series of 1-D FFT's are performed on the input array. First, a $n_3$-point FFT is performed on each $n_1n_2$ complex vector. Next, a $n_2$-point FFT is performed on each $n_1n_3$ vector. Both of these one-dimensional transforms can be performed with no communication. A transpose phase then follows where each thread must communicate
its results to the thread that will next use that portion of the input array. During this transpose, the algorithm requires a large amount of communication, which limits the speedup of the application. The final step consists of performing a $n_1$-point FFT on the set of $n_2n_3$ vectors.

Because of memory limitations, we were not able to run the full benchmark size of $n_1 = 256$, $n_2 = 256$, and $n_3 = 128$. Instead we use an input set with $n_1 = 64$, $n_2 = 128$, and $n_3 = 64$ in our experiments.

**Conjugate Gradient** (CG) finds an estimate of the smallest eigenvalue of a symmetric positive definite sparse matrix ($A$) that has been filled with a random pattern of nonzeros. The algorithm uses the power method to approximate the reciprocal of the largest eigenvalue of $A^{-1}$ iteratively, and is divided into two main loops. The inner loop approximates the solution $z$ to the linear system $Az = x$ using the conjugate gradient method. The exact implementation used is defined in [6]. The outer loop implements the rest of the power method to arrive at the final eigenvalue solution to $A$. We use a matrix of order 14,000 with 1,853,104 nonzero elements, the size specified in the benchmark documentation.

**MultiGrid** (MG) is a 3D multigrid benchmark that obtains a solution $u$ to the discrete Poisson problem $\nabla^2 u = v$, where $v$ is a $N \times N \times N$ grid. Multigrid kernels are used in many simulations of large scale natural phenomena, and use an iterative finite differencing method to arrive at the approximate solution to a system of partial differential equations. MG applies a multigrid cycle to a hierarchy of grid points that represent the physical domain until a convergence condition has been satisfied. See [6] for further details. We ran MG with $N = 128$ for 20 multigrid cycles.

**Integer Sort** (IS) sorts $N$ keys (or elements) in parallel through a process known as ranking. Each key has a rank, which corresponds to the index value the key would have if the entire sequence of keys were sorted, and the computation performed by the application is to determine the ranking of all $N$ keys through a bucket sorting algorithm.
The key values are randomly determined at the start of the program, and all keys fall into the range \((0, B_{\text{max}})\). The only shared data structure in the algorithm consists of a single array of \(B_{\text{max}}\) integers, representing all possible values of each key. This array must be communicated between processes each iteration, and therefore the communication required and subsequent performance of the algorithm depends heavily on the value of \(B_{\text{max}}\) chosen. We use \(N = 23\) and \(B_{\text{max}} = 17\), and our speedups are substantially lower than some previous studies with this application (most notably [27], which used a \(B_{\text{max}}\) of 7).

Other Applications

The version of **Successive Over Relaxation** (SOR) used here is taken from the TreadMarks [28] distribution of sample DSM applications. SOR is a nearest-neighbor algorithm used to solve partial differential equations. The implementation we use is a so-called “red-black” SOR, because the source matrix is broken up into two separate input matrices in a checkerboard pattern. During the first half of an iteration, each value in the “black” matrix is computed at the average of the four values from the “red” matrix that appear around it. During the second part of an iteration, the “red” values are computed from the corresponding value taken from the “black” matrix.

The matrix is distributed among available threads by allocating \(\frac{\text{NumRows}}{\text{NumThreads}}\) contiguous rows to each thread. Thus, the only values that are shared throughout the computation are those along the border between two threads. We use an input matrix size of 2048 X 2048 integers, and iterate over the matrix for 50 iterations. To eliminate cold start behavior, we discard statistics from the first iteration.

The **Traveling Salesman Problem** (TSP) algorithm takes an input set consisting of \(N\) cities with weighted path lengths connecting each city. The problem is to find the path with the minimum total length that passes through each city exactly once, returning to the original city at the end. A complete route through all cities is called a **tour**. The program maintains the length of the current minimum tour found
so far in a global variable protected by a lock. This allows the search space to be pruned by abandoning any partially evaluated tours that exceed this global minimum.

The program proceeds as follows. Each thread acquires the lock that protects the global list of partially evaluated tours. This list is maintained as a sorted heap, with the shortest partially evaluated tours at the head of the queue. Thus, shortest complete tours have a better chance of being found early in the computation, reducing unnecessary tour evaluation. When a thread retrieves a tour from the list, the tour is evaluated until it is within a set number ($R$, the recursion level) of cities from the end, after which the tour lock is released and the tour is evaluated locally. We use an input set with $N = 19$ and $R = 12$.

**ILINK** [20] is part of a parallel implementation of the LINKAGE [33, 34, 35] package, which uses family pedigree information to map human genes and locate disease-related genes in the human genome. The goal in linkage analysis is to compute the probability that a recombination between two genes will occur. ILINK begins with a single initial estimate of the recombination probability and uses an iterative method to find a maximum likelihood estimate of a recombination occurring.

The major data structure in the program is the *genarray* structure, which stores the probability that an individual has a specific genotype, based on the recombination factor and the genotypes of family members already analyzed. The *gene* structure is used to accumulate the contributions from each pair of parental genotypes. The parallel version of ILINK replicates this *gene* structure across all processes, and local updates that will eventually be placed in the *genarray* structure are made to the local *gene* structure. When all threads have finished an iteration, one thread sums the contributions to *gene* from each thread and updates the global *genarray* structure. This sequential portion of the algorithm reduces the performance achieved in a parallel environment. We ran ILINK with the *clp* input set, which provides data on 12 families with autosomal dominant nonsyndromic cleft lip and palate.
4.3 Performance of Brazos

In this section we present performance numbers for several DSM systems, as outlined in Table 4.2. LRC is a version of TreadMarks 1.0.0 that we have ported to Windows NT, and only supports a single application thread per process. LRC+MT is a modification to LRC that allows multiple application threads per process in order to take advantage of the hardware coherence mechanisms available on our SMP machines. LRC+MT is based on the release of TreadMarks for use in the PThreads threading environment available for Unix systems that support lightweight threads. The MC+SScC configuration is the Brazos DSM system without the adaptive features enabled. Finally, Brazos refers to the full implementation of Brazos, with all adaptive features enabled.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Multithreading?</th>
<th>Communication Primitives Used</th>
<th>Consistency Model Used</th>
<th>Adaptive Mechanisms?</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRC</td>
<td>No</td>
<td>Point-to-Point</td>
<td>Lazy RC</td>
<td>N/A</td>
</tr>
<tr>
<td>LRC+MT</td>
<td>Yes</td>
<td>Point-to-Point</td>
<td>Lazy RC</td>
<td>N/A</td>
</tr>
<tr>
<td>MC+SScC</td>
<td>Yes</td>
<td>Multicast</td>
<td>Scope Consistency</td>
<td>No</td>
</tr>
<tr>
<td>Brazos</td>
<td>Yes</td>
<td>Multicast</td>
<td>Scope Consistency</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 4.2: DSM Systems Used in Results Chapter

Figure 4.2 shows the speedups on 8 processors for the twelve applications described above, and compares the overall performance of Brazos with two baseline systems, LRC and LRC+MT. Overall, Figure 4.2 shows that the Brazos DSM system outperforms the LRC implementation for all programs except LU. The average performance improvement is 83% for all 12 applications. Brazos also performs better than LRC+MT for most applications. In the following sections, we examine each application in detail and isolate those aspects of Brazos that are most beneficial to performance.
4.3.1 Overview

In this section, we analyze the performance advantages of using multithreading, multicast, and software scope consistency by comparing a base version of Brazos (MC+SScC, which does not contain any adaptive mechanisms; the performance effect of the adaptive mechanisms will be examined in Section 4.3.3) with LRC and LRC+MT, two DSM systems employing lazy release consistency as implemented in the TreadMarks DSM system. Note that for $N$ application threads, LRC will have $N$ single-application-thread processes that must communicate through network primitives, even if two processes are located on the same machine. LRC+MT and MC+SScC will have $N/M$ processes that must communicate, where $M$ is the number of processors per machine.
Figure 4.3 gives the breakdown of each application’s execution in terms of time spent in the NT kernel, time spent executing DSM code, and time spent idle. The “DSM Time” portion of this graph contains time spent executing both user (e.g., application) and DSM system code. These measurements were obtained by using thread statistics API’s available in Windows NT, and are averaged over all application threads in each program. Table 4.3 gives the average percentages for all applications for each DSM implementation.

![Application Execution Time Breakdown](image)

Figure 4.3: Application Execution Time Breakdown

In general, the kernel time requirements of all applications are low for the three implementations, although several applications spend greater than 10% of their execution time in the NT kernel. The average idle time is over 50% for the LRC and LRC+MT implementations, as indicated by Table 4.3, while the average idle time
is just under 48% for MC+SScC. In general, those applications in Figure 4.3 with a high percentage of idle time will not achieve good speedups.

<table>
<thead>
<tr>
<th></th>
<th>DSM Time</th>
<th>Kernel Time</th>
<th>Idle Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRC</td>
<td>40.33</td>
<td>6.78</td>
<td>52.88</td>
</tr>
<tr>
<td>LRC+MT</td>
<td>44.56</td>
<td>5.13</td>
<td>50.30</td>
</tr>
<tr>
<td>MC+SScC</td>
<td>49.13</td>
<td>3.39</td>
<td>47.47</td>
</tr>
</tbody>
</table>

Table 4.3: Breakdown of Execution Time

Figure 4.4 shows the overheads in the three DSM systems broken down into four categories: time spent in the access violation handler code, time spent in lock code, time spent in barrier code, and time spent executing application code. Obviously there is a strong correlation between the “Application Code” category and the performance of the application. Table 4.4 gives average values across all applications for each category and DSM system shown in Figure 4.4.

The access violation category in Figure 4.4 shows the percentage of time that threads execute DSM code to retrieve pages and diffs from remote processes. This includes time spent sending diff and page requests and waiting for responses. In general, we expect LRC to spend more time in the access violation handler than the two hybrid hardware/software systems because LRC must communicate with potentially twice as many other processes than either LRC+MT or MC+SScC. Table 4.4 shows that the average access violation time is over 23% for LRC, but is only 14.9% for LRC+MT and 13.3% for MC+SScC.

<table>
<thead>
<tr>
<th></th>
<th>Application</th>
<th>Barrier</th>
<th>Lock</th>
<th>Access Violation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRC</td>
<td>44.75</td>
<td>22.24</td>
<td>9.61</td>
<td>23.40</td>
</tr>
<tr>
<td>LRC+MT</td>
<td>52.46</td>
<td>23.91</td>
<td>8.75</td>
<td>14.88</td>
</tr>
<tr>
<td>MC+SScC</td>
<td>58.26</td>
<td>21.27</td>
<td>7.18</td>
<td>13.29</td>
</tr>
</tbody>
</table>

Table 4.4: Breakdown of DSM Overhead
Figure 4.4: Breakdown of DSM Overhead

The blue portion of each bar in Figure 4.4 shows the time spent in barrier code for each application. On average, as much or more time is spent in barrier code than is spent retrieving modified portions of shared pages, as shown in Columns 3 and 5 of Table 4.4. Further analysis of the barrier code time portion of Figure 4.4 reveals that most of the time in the barrier category is spent actually waiting to be released from the barrier, not executing barrier-related code. Applications for which the barrier code portion makes up a significant percentage of execution time (such as Barnes-Hut) may suffer from a load-balance problem.

Four applications make use of lock variables for synchronization (Ocean, Raytrace, TSP, and Water). For TSP and Water, the lock portion of the execution time is moderate, and Ocean spends minimal time executing lock-related code for all three implementations. Raytrace's execution time is heavily dominated by lock-related code.
execution, comprising an average of 74.9% of the execution time for \textbf{LRC}, 73.1% for \textbf{LRC+MT}, and 59.2% for \textbf{MC+SScC}.

### 4.3.2 Per-Application Analysis

In order to isolate where particular performance benefits come into play, we present an application-by-application analysis of the above three systems in the following sections. For each application, we examine the performance benefits of application-level multithreading by comparing the performance of \textbf{LRC} with that of \textbf{LRC+MT}. We then compare the performance of \textbf{LRC+MT} with that of \textbf{MC+SScC} in order to clarify the improvements to performance by employing multicast communication and scope consistency. Unless otherwise noted, the detailed discussion and accompanying data refer only to the 8-processor case. However, the speedup graphs also show speedups for 2, 4, and 6 processors. With 2 processors, the \textbf{LRC+MT} and \textbf{MC+SScC} implementations should perform identically since only hardware mechanisms are used. Likewise, the 4-processor versions of these two implementations should be fairly close in performance because \textbf{MC+SScC} will not employ multicast communication until there are more than 2 processes. Section 4.3.3 examines the effects of adding our runtime adaptive mechanisms and maintaining per-application history information across program executions.

**Barnes Hut**

Figure 4.5 indicates that the speedup for Barnes Hut increases from 1.94 under \textbf{LRC} to 2.84 under \textbf{LRC+MT} for 8 processors. The high communication rate (shown as messages per second in Figure 4.6) in Barnes Hut causes the rolloff in speedup for \textbf{LRC} seen in Figure 4.5. Additionally, the high overhead of diff storage and garbage collection limit the performance of the \textbf{LRC} algorithm for this application (see the discussion on IS later in this section for further details about diff overhead). By making use of available hardware coherence mechanisms, the message rate drops
from 7263 under LRC to 3849 messages per second under LRC+MT. The majority of the messages avoided by LRC+MT are false sharing misses as threads traverse and update elements in the octree each time step [27]. Making use of the available hardware coherence mechanisms allows the effects of false sharing between threads in the same process to be minimized.

MC+SScC improves the Barnes Hut speedup from 2.84 under LRC+MT to 3.91. The performance improvement of MC+SScC over LRC+MT is also related to the high false sharing rates found in Barnes Hut.

Table 4.5 shows that Barnes Hut uses just under 80% of all multicast diffs. This statistic does not mean that processes necessarily use data modified by other processes in the multicast diffs, but only that they use some data on the page before it is next invalidated. In the case of Barnes Hut, processes write to their portion of the global
data structure at a very fine granularity. perform a barrier operation to make sure all processes are finished with the current timestep, and then access their own modified portions of the global data structure on the other side of the barrier. As Figure 4.7 shows, if these processes receive indirect diffs for falsely shared data before the process attempts to access the page, the harmful effects of false sharing can be obviated just as in the case of true sharing misses. The reduction in the effects of false sharing leads to the dramatic drop in the message rate under $\text{MC+SScC}$ as shown in Figure 4.6.

**CG**

Figure 4.8 shows that CG achieves a low speedup overall because of the high communication rate (over 4000 messages sent per second for LRC, as indicated in Figure 4.6). Providing support for application-level multithreading in $\text{LRC+MT}$ improves per
<table>
<thead>
<tr>
<th></th>
<th>Indirect Multicast</th>
<th>% Used</th>
<th>% Unused</th>
<th>% Conflicts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Diffs Received</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Barnes Hut</td>
<td>34,563</td>
<td>79.42</td>
<td>20.1</td>
<td>.48</td>
</tr>
<tr>
<td>CG</td>
<td>28,504</td>
<td>34.49</td>
<td>1.5</td>
<td>64.01</td>
</tr>
<tr>
<td>FFT-3D</td>
<td>5906</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>ILINK</td>
<td>31317</td>
<td>31.85</td>
<td>45.33</td>
<td>22.82</td>
</tr>
<tr>
<td>IS</td>
<td>1456</td>
<td>8.79</td>
<td>78.44</td>
<td>12.77</td>
</tr>
<tr>
<td>LU</td>
<td>4193</td>
<td>0</td>
<td>99.19</td>
<td>.81</td>
</tr>
<tr>
<td>MG</td>
<td>14,769</td>
<td>1.25</td>
<td>96.62</td>
<td>2.13</td>
</tr>
<tr>
<td>Ocean</td>
<td>11,975</td>
<td>16.73</td>
<td>55.51</td>
<td>27.76</td>
</tr>
<tr>
<td>Raytrace</td>
<td>288</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>SOR</td>
<td>594</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>TSP</td>
<td>6</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>Water</td>
<td>5,969</td>
<td>57.14</td>
<td>40.5</td>
<td>2.36</td>
</tr>
</tbody>
</table>

Table 4.5: Indirect Multicast Diff Analysis

formance by reducing the number of messages per second by half. However, parallelism in the multithreaded implementation is limited because threads tend to fault on the same page simultaneously. This leads to a situation in which both threads wait for a single page of data, limiting the computation/communication overlap and reducing potential performance improvements. In CG, 42.9% of all access violations in LRC+MT resulted in a thread suspending inside the access violation handler due to another thread already waiting for updates to the same page. These access violations are referred to as “useless access violations”.

MC+SScC improves the speedup of CG from 2.43 to 3.62 because over 34% of the indirect multicast diffs received are used, as indicated by Table 4.5. Roughly 40% of all access violations are still “useless” under MC+SScC due to threads faulting on the same page simultaneously, however. In addition, threads in different processes also simultaneously fault on the same page. This leads to “multicast conflicts” as described in Section 2.4.1, and results in 64% of all multicast diffs being discarded (Column 5 in Table 4.5). The early update mechanism presented in Section 4.3.3
substantially improves CG's performance over the results shown in Figure 4.8 by eliminating these multicast conflicts.

**FFT-3D**

Figure 4.9 shows the speedups for FFT-3D. All three implementations achieve low speedup for this application because of the large amount of data that must be transferred during the transpose phase of each iteration. After the transpose, threads communicate their results to the new owners of their segment of the array. A thread will over-write entire pages of data, and the large diffs for each of these pages must be computed, sent, and applied during each iteration.

Comparing the performance of **LRC** and **LRC+MT**, Figure 4.9 indicates that the software-only implementation runs 20% faster at the 8-processor point, despite the
obvious speed advantage of the multithreaded implementation (this is evidenced by the large performance difference between LRC+MT and LRC at the two-processor mark, where LRC+MT is only using hardware coherence mechanisms). The reason for this counter-intuitive result lies in the competition among threads for shared resources, specifically access to the network. In the LRC+MT implementation, only one thread is allowed to have a network request outstanding at any one time, and other faulting threads must spin-wait on a mutex variable until the first thread's request is satisfied. In LRC, where every thread is running in its own process address space, there is no such restriction. As a result, the total time spent in the access violation handler increases from 44.0 seconds under LRC to 56.4 seconds under LRC+MT.

LRC and LRC+MT both outperform MC+SScC for FFT-3D. This is because of FFT-3D's sharing patterns, in which all communication occurs between two pro-
processors at a time. This eliminates any possible performance advantage of multicast.

Even though the sharing patterns in FFT-3D can not make use of multicast, Table 4.5 shows that FFT-3D does receive nearly 6000 indirect multicast diffs. Closer examination reveals that over 99% of the indirect diffs are received by Process 0. This stems from the fact that Process 0 is the default initial page manager for all shared pages. Therefore, this process will always be in the copyset for every page of shared data and receive all indirect multicast diffs, even if most of them are never used.

This situation leads to a load imbalance, in which Process 0 arrives at each barrier instance well behind other processes due to a slowdown in computation caused by the processing of unused multicast diffs. In LRC and LRC+MT, all processes wait approximately the same total amount of time at barriers, indicating a good work load balance. In MC+SScC, however, Processes 1, 2, and 3 wait more than twice as long.
for barrier releases as Process 0 due to the load imbalance created by the processing of unused multicast diffs. In Section 2.4, we show how the dynamic copyset reduction mechanism can alleviate this effect.

Figure 4.10: ILINK Speedups

ILINK

Figure 4.10 shows that the performance of ILINK is good for both hybrid DSM implementations (LRC+MT and MC+SScC). The communication rate for ILINK is one of the lowest shown in Figure 4.6, and application-level multithreading is able to reduce the rate from 1568 messages per second in LRC to only 743 in LRC+MT. This reduction allows the LRC+MT implementation to run 21% faster than LRC.

MC+SScC further reduces the message rate to just 545 messages per second because ILINK uses 32% of the received indirect diffs, which in turn eliminates al
most 33% of the diff requests sent in the LRC+MT implementation. Despite this, the overall performance does not improve substantially because 45% of the indirect multicast diffs go unused, and almost 23% result in multicast conflicts. Early updates and the dynamic copyset reduction mechanism will help marginally to improve the performance of ILINK as we will see in Section 4.3.3.

**Integer Sort**

Figure 4.11 shows the speedups obtained for IS. The single-thread LRC implementation does not perform well on this application, as the 8-processor version runs almost twice as slowly as the uniprocessor version. Figure 4.4 shows that IS spends the least percentage of its execution time in user code of all 12 applications under the LRC protocol, with roughly 85% of the execution time being split evenly between barrier time and access violation time.

The reason for this lies in the way diffs are managed in the lazy release protocol. In IS, processes respond to requests for diffs with multiple large diffs that have been received from other processes for the same page. Because the LRC protocols do not allow these diffs to be merged, all of these large diffs must be sent when a request for the page is received. This is inefficient because only the last diff is strictly needed, since information in earlier diffs will be overwritten. Computation time is wasted in creating, sending, and applying these extraneous diffs at both the sending and receiving side. As evidence of this, the average diff reply message under LRC consists of 4 diffs, each of size 4104 bytes (the diff size needed when every word on a shared page has been modified). Because LRC+MT only has 4 processes instead of LRC's 8, the average diff response for LRC+MT is 2 diffs, each of size 4104 bytes.

**MC+SScC** improves the performance of IS by 43% to a speedup of 2.92 on 8 processors. The main reason for this lies in the way Brazos manages diffs. Processes do not cache diffs sent by other processes in Brazos, in order to allow diffs to be merged. Through multicast, Brazos ensures that once one process has requested a
diff, all processes receive it, eliminating the need for diff forwarding as is done in the LRC protocols. Therefore, diffs do not have to be maintained past the point where they are requested by one other process. The average response for a diff request in MC+SScC is only 4104 bytes, the maximum size of a single diff.

IS shows another advantage of Brazos over the LRC-based protocols: diff overhead and garbage collection. As described above, Brazos only has to maintain diffs until they are requested by one other process, whereas LRC-based protocols must maintain diffs until they are seen by all other processes, or until the diff heap grows beyond a certain point and a repo operation is required at the next barrier [27]. A repo operation causes all processes to collect any outstanding diffs for all shared pages at a barrier in order to bring all processes up to date. This garbage collection mechanism is expensive because the time spent collecting outstanding diffs and sending them
can be costly. Table 4.6 shows the *maximum* required diff storage space observed (in megabytes) during the execution of our 12 programs, as well as the percentage of execution time spent by the **LRC** protocols performing garbage collecting via *repo* mechanisms.

<table>
<thead>
<tr>
<th></th>
<th>LRC</th>
<th></th>
<th>LRC+MT</th>
<th></th>
<th>MC+SSeC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Overhead</td>
<td>Repo Time</td>
<td>Overhead</td>
<td>Repo Time</td>
<td>Overhead</td>
</tr>
<tr>
<td>Barnes Hut</td>
<td>4.86 MB</td>
<td>22.75%</td>
<td>5.92 MB</td>
<td>7.59%</td>
<td>6.06 MB</td>
</tr>
<tr>
<td>CG</td>
<td>7.63 MB</td>
<td>.70%</td>
<td>3.79 MB</td>
<td>.87%</td>
<td>.06 MB</td>
</tr>
<tr>
<td>FFT-3D</td>
<td>3.64 MB</td>
<td>9.82%</td>
<td>5.03 MB</td>
<td>7.03%</td>
<td>3.01 MB</td>
</tr>
<tr>
<td>ILINK</td>
<td>1.37 MB</td>
<td>3.34%</td>
<td>1.35 MB</td>
<td>1.48%</td>
<td>.37 MB</td>
</tr>
<tr>
<td>IS</td>
<td>8.06 MB</td>
<td>.73%</td>
<td>3.53 MB</td>
<td>1.00%</td>
<td>.25 MB</td>
</tr>
<tr>
<td>LU</td>
<td>2.27 MB</td>
<td>1.24%</td>
<td>1.58 MB</td>
<td>.56%</td>
<td>.71 MB</td>
</tr>
<tr>
<td>MG</td>
<td>2.62 MB</td>
<td>13.77%</td>
<td>2.41 MB</td>
<td>5.44%</td>
<td>6.28 MB</td>
</tr>
<tr>
<td>Ocean</td>
<td>2.67 MB</td>
<td>9.05%</td>
<td>2.08 MB</td>
<td>4.24%</td>
<td>.16 MB</td>
</tr>
<tr>
<td>Raytrace</td>
<td>7.51 MB</td>
<td>.15%</td>
<td>7.28 MB</td>
<td>.07%</td>
<td>.41 MB</td>
</tr>
<tr>
<td>SOR</td>
<td>.32 MB</td>
<td>0%</td>
<td>.24 MB</td>
<td>0%</td>
<td>.02 MB</td>
</tr>
<tr>
<td>TSP</td>
<td>.37 MB</td>
<td>0%</td>
<td>.36 MB</td>
<td>0%</td>
<td>.02 MB</td>
</tr>
<tr>
<td>Water</td>
<td>4.99 MB</td>
<td>.43%</td>
<td>3.11 MB</td>
<td>.40%</td>
<td>.71 MB</td>
</tr>
</tbody>
</table>

Table 4.6: **Diff-Related Overheads Observed Per Process**

Table 4.6 shows that Brazos requires less diff storage than either **LRC** protocols for all applications except Barnes Hut and MG, both of which make a large quantity of diffs in a short period of time immediately after a barrier, leading to a spike in the amount of diff overhead during the program execution. In MG, Brazos requires more maximum diff storage because diff structures contain enough storage space to hold the largest possible diff, regardless of the actual size of the diff. In the **LRC** protocols, only as much diff space as is needed is allocated for diff storage. Brazos quickly reuses diff storage space because a past history of diffs does not have to be maintained. Diff merging also reduces the overhead requirements.

Finally, Table 4.6 shows that the time spent executing *repo* code can be a substantial percentage of overall execution time. This overhead can also be seen in the
larger barrier code percentages noted in Figure 4.4, since the repo operations occur within barrier code. In Brazos, a diff structure can be placed back on the free list as soon as the diff is requested by one other process, eliminating the need for explicit garbage collection of used diff structures through a mechanism such as a repo.

Figure 4.6 shows that the average number of messages per second increases for IS when moving from LRC to LRC+MT. In actuality, the total number of messages drops nearly in half (from 38,976 messages to 20,220 messages), but the execution time drops by more than a factor of 3 (from 57 seconds to 17 seconds) due to the reduction in diff management overhead seen in Table 4.6. In a similar manner, the number of messages sent drops by 30% when moving from LRC+MT to MC+SScC, but the execution time drops by almost 45%. This causes the message rate to increase, even though the overall amount of communication and execution time have been reduced.

![Speedup Chart](image)

**Figure 4.12: LU Speedups**
LU

LU benefits slightly from multithreading, with the speedup improving from 4.13 on 8 processors for LRC to 4.59 for LRC+MT. Figure 4.6 indicates that the communication rate drops in half (from 1,531 to 737 messages per second); however, this drop is not as a result of a reduction in consistency-related misses, as in the case of most of the other applications. Figure 4.13 shows that consistency misses make up a very small percentage of all access violations in LU for the LRC implementation, while the number of cold misses dominates.

![Graph showing % of all segmentation faults for various applications]

**Figure 4.13 : Breakdown of Access Violation Faults in LRC**

Cold misses occur when a process accesses a page for the first time and must retrieve the entire page from the page manager. The majority of our applications are iterative in nature, and we ignore the first iteration to reduce start-up effects.
on parallel performance evaluation. Therefore, cold misses show up as a very small percentage of overall misses for most of the programs, as shown in Figure 4.13. In LU, however, cold misses occur throughout the application’s execution and are important to overall program performance, even at large problem sizes.

LRC+MT reduces the number of cold misses in LU by 50.3% (from 16,230 to 8,068), leading to a slight 3.9% performance improvement. To understand this, consider the cold misses to a single shared page of data. In the software-only system, each thread other than the page’s manager will fault on the page and incur a cold miss. This results in 7 cold misses for each page, assuming 8 processes. In the hybrid systems, each process that is not the page’s manager will cold-miss on the page only once for all threads in the process. Thus, only 3 cold misses will result for an 8-processor machine consisting of four 2-application-thread processes. If every thread were to require every page of shared data, this translates into a maximum $\frac{7-3}{7} = 57.1\%$ reduction in cold misses over the course of a program’s execution, which corresponds well with our observed results.

MC+SScC performs worse for LU than either of the LRC-based protocols past 4 processors. This can be attributed to the use of multicast, because it is only when there are three processes or more (e.g., 6 processors or more in our configuration) that multicast is even used. As Table 4.5 indicates, LU does not make use of any indirect multicast diffs received. As in the case of FFT-3D, the only reason any process in LU receives indirect multicast diffs is because all pages initially begin with Process 0 as the owner.

MG

MG performs poorly on LRC because of the large amount of sharing and resultant communication. Less than 40% of MG’s execution time is spent executing user code, as indicated in Figure 4.4, with large percentages of time spent in both barrier code and access violation handler code. The large percentage of time spent executing repo
code (Table 4.6) contributes to the long barrier delays for this application.

**LRC+MT** provides a near doubling in performance over **LRC** for MG at 8 processors. This improvement arises because the number of messages sent falls from 133,512 to 67,250 due to the use of the hardware coherence mechanisms available on the multithreaded hybrid system.

**MC+SScC** is unable to help performance significantly because only 1.15% of the multicast diffs are used before being invalidated, as indicated in Table 4.5. However, the addition of the adaptive home-based page management algorithm described in Section 4.3.3 will give the **MC+SScC** implementation a slight performance increase by lowering the overhead associated with repeatedly creating, sending, and applying large diffs.
Ocean

Ocean has been included in this study because it represents a class of applications that are not well-suited for use in a software DSM environment. The extremely high communication rates shown in Figure 4.6 do not allow Ocean to achieve a speedup above 1 for any implementation studied, as indicated in Figure 4.15. Figure 4.15 shows that Ocean spends less than 27% of its execution time processing user code for all three implementations. The two hybrid systems perform better than the software-only system simply because of the reduction in the amount of communication in these systems. Ocean requires a much faster network protocol and communication media to achieve satisfactory performance. Section 4.3.7 describes Ocean's performance on a hardware-only system.
Raytrace

Similar to Ocean, Raytrace is an example of an application whose communication requirements make it difficult to achieve good speedup on a software DSM system. In Raytrace, the majority of the communication requirements stem from the passing of lock grants and requests between processes. Synchronization-related messages make up over 50% of all messages for this application. To make matters worse, the average diff response is only 13 bytes long, which means there is a very high overhead for each diff response sent relative to the amount of data transferred.

Figure 4.16 shows that LRC never achieves a speedup greater than 1 for Raytrace due to the high communication requirements discussed above. The speedup on LRC+MT is 1.7 on 2 processors, but this configuration uses only hardware princi
tives between the only two processing threads. As soon as network communication is required (at four processors), the speedup for LRC+MT drops below 1.

MC+SScC, on the other hand, achieves a speedup of about 1.6 for 2 to 8 threads. This performance improvement is a direct result of the use of software scope consistency. The only data that is write-shared in Raytrace occurs within short critical sections. SScC communicates this data along with the lock grant message, and does not invalidate data outside of the critical sections, as is done by the two LRC protocols. Therefore, the sending of nearly all modified data with lock responses and the reduction in false sharing reduce the number of non-synchronization messages from 258,158 in LRC+MT to only 192 in MC+SScC. However, the number of synchronization messages required remains the same, dominating execution time and limiting the speedup of Raytrace to just 1.64 on 8 processors.

![Figure 4.17: SOR Speedups](image-url)
SOR

The speedups for SOR are 5.4 for LRC and 5.7 for LRC+MT and MC+SScC. In our implementation of SOR, only data along the row boundary between two processors is shared, resulting in very high computation to communication ratios and good speedup. LRC+MT improves performance by eliminating the network communication between threads co-located on the same machine. This has the effect of cutting the communication rate in half, as indicated in Figure 4.6.

Because of the pair-wise sharing found in SOR, the use of multicast in MC+SScC does not offer any performance benefit. Table 4.5 shows that 0% of the 594 multicast diffs are used before being invalidated, as expected. As in the case of LU and FFT-3D, all of these multicast diffs are sent only because Process 0 starts out as the page manager for the entire address space. This situation is addressed in Section 4.3.3, but the communication rates are already so low that the impact on performance of the adaptive techniques is negligible.

TSP

Figure 4.18 shows that TSP gets speedups of 5.6 for both LRC and LRC+MT. Most of the messages in TSP are diff request and responses, making up over 85% of all messages. Additionally, these messages are small, with the average diff response being only 168 bytes long. Small messages of this kind degrade performance because of their high overhead, as shown in Figure 3.2. The reason that LRC+MT does not achieve higher performance on TSP, despite the 40% reduction in the message rate shown in Figure 4.6, again results from the contention for access to the network that was observed with FFT-3D. The total time spent in the access violation handler increases from 3.6 seconds in LRC to 6.4 seconds in LRC+MT.

The performance of two Brazos versions are shown in Figure 4.18: MC+LRC and MC+RC. The main critical section in TSP contains a significant amount of work to select a new tour and update the priority queue, and therefore TSP represents an ap-
Figure 4.18: TSP Speedups

Application that is difficult to convert to SSeC semantics. The MC+LRC curve shows the base Brazos configuration using the lazy release consistent lock-style semantics, and the MC+RC curve shows Brazos with release consistent lock semantics. Recall from Section 3.4.2 that the LRC-type locks in Brazos will multicast updates for all pages modified since the last acquire operation when a process relinquishes a lock to another process, and the RC-style locks invalidate all copies of modified data before allowing a release to complete.

From Figure 4.18, it is clear the MC+LRC implementation works best for this application. The number of messages drops from 12,000 in LRC+MT to just under 2,000 in MC+LRC because nearly all data messages are piggybacked on lock grant messages. This style of lock mechanism works for TSP because the amount of data that has to be transferred at each lock grant is small. Otherwise, multiple-
messages may be required, which would severely hamper performance.

MC+RC does not fare as well in TSP as MC+LRC, only achieving a speedup of 5.8 for 8 processors. This is because the MC+RC locking primitives invalidate pages at each release, for which separate diff requests must be sent when threads next fault on the invalid pages. Because updates are used in MC+LRC, these faults and subsequent messages are avoided. MC+RC performs slightly better than either LRC implementations because 20% of the indirect multicast diffs sent are used, reducing the number of messages and access violation faults.

![Graph showing speedup vs number of processors for different schemes.](image)

Figure 4.19: Water Speedups

Water

The speedups for Water are 3.28 for LRC and 3.78 for LRC+MT. This improvement results from a decrease in both required data and synchronization messages. Over 70%
of all messages in Water are synchronization-related, and the majority of these arise from the sending and receiving of lock grants and requests. By co-locating threads in the same process, LRC+MT reduces the number of synchronization messages sent from 88,169 to 48,708 because some lock grants can be handled exclusively through Windows NT primitives when the two threads involved in the transfer are located on the same machine. As a result, the average lock wait time drops from 4.44 msec to 2.15 msec.

**MC+SSC** performs slightly better than LRC, but worse than the LRC+MT protocol for this application, only achieving a speedup of 3.41 on 8 processors. The main reason for this lies in the overhead required to implement scope consistency, and the negligible benefits of SSC for this application.

SSC does not provide benefit if local scope delimiters (i.e., critical sections) appear immediately before global scope points (i.e., barriers). This is because the savings in SSC over LRC arises from not invalidating data modified outside the local scope in order to reduce false sharing after the lock release. However, if there is a global scope immediately after the local scope, all shared data will be made coherent there, resulting in little behavior difference between the LRC protocols and SSC.

Additionally, there is overhead associated with implementing SSC without hardware support that is not present in the LRC-based protocols. When data is modified inside a local scope, the DSM runtime system has to add the value and address of the variable to a linked list attached to the lock currently held. This list will then be passed to the next lock acquirer. Maintaining this list is expensive relative to performing the shared write, as is done in LRC. This is especially true if there are many writes within the same critical section in a single global scope, as the length of the list grows with each new write to a different memory location. In Water, these factors remove any possible performance benefit from the use of scope consistency. The five critical sections in Water all appear immediately before a barrier, eliminating any false sharing reduction that might have been achieved through the use of SSC.
Additionally, one of the critical sections inside of the inter-molecular force calculation step contains a substantial number of writes, increasing the overhead of maintaining the list of critical section-modified data. For these reasons, **MC+SScC** performs worse than **LRC+MT** for this application.

### 4.3.3 Evaluation of Adaptive Protocols

In this section we examine the performance benefits of the three runtime adaptive protocols described in Chapter 2: dynamic copyset reduction, an early update protocol, and an adaptive page-management protocol. For each of these three mechanisms, we examine the applications for which a performance impact was observed.

#### Dynamic Copyset Reduction

The dynamic copyset reduction (DCR) mechanism was developed to provide processes with a way to stop receiving useless indirect multicast diffs. We saw how this effect can adversely impact performance in our discussions of FFT-3D, LU, SOR, and Water. Table 4.5 shows that all applications (with the exception of CG) do not use 20% or more of received indirect multicast diffs.

As described in Section 2.4.2, the DCR protocol associates a counter with each page of shared data. The counter is preset to either an application-specific value contained in the history file, or set to an initial value if the program has never been run before. Each time an indirect diff is received that is not used before the next invalidation of the page, the counter is decremented. If a page is used, the counter is reset to the initial value. If the counter reaches zero, the process removes itself from the copyset, ensuring that no further indirect diffs for that page will be sent.

If the process should fault on the page sometime in the future, the entire page is brought in from the manager. Because receiving indirect diffs is an inexpensive operation relative to having to retrieve an entire page of data across the network, we reset the counter value to the initial value plus a hysteresis factor each time the page
is erroneously dropped from the copyset.

First we look at how successful DCR is at reducing the number of unused indirect multicast diffs received. Table 4.7 shows the reduction in the number of indirect diffs received, as well as how the percentage of used indirect diffs changes as a result of using the DCR mechanism.

<table>
<thead>
<tr>
<th></th>
<th>Original Num Multicast Diffs</th>
<th>Num Multicast Diffs with DCR</th>
<th>% Reduction</th>
<th>Old Usage</th>
<th>New Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes Hut</td>
<td>34,564</td>
<td>34,448</td>
<td>.3%</td>
<td>79.42%</td>
<td>79.67%</td>
</tr>
<tr>
<td>CG</td>
<td>28,504</td>
<td>28,504</td>
<td>0%</td>
<td>34.49%</td>
<td>34.33%</td>
</tr>
<tr>
<td>FFT-3D</td>
<td>5,987</td>
<td>3,138</td>
<td>47.6%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>ILINK</td>
<td>31,315</td>
<td>28,659</td>
<td>8.5%</td>
<td>31.85%</td>
<td>34.70%</td>
</tr>
<tr>
<td>IS</td>
<td>11,648</td>
<td>11,648</td>
<td>0%</td>
<td>11.15%</td>
<td>11.15%</td>
</tr>
<tr>
<td>LU</td>
<td>4,190</td>
<td>4,190</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>MG</td>
<td>14,784</td>
<td>2,034</td>
<td>86.2%</td>
<td>1.25%</td>
<td>10.37%</td>
</tr>
<tr>
<td>Ocean</td>
<td>11,965</td>
<td>6,965</td>
<td>41.8%</td>
<td>16.73%</td>
<td>28.01%</td>
</tr>
<tr>
<td>Raytrace</td>
<td>288</td>
<td>288</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>SOR</td>
<td>594</td>
<td>18</td>
<td>97.0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>TSP</td>
<td>6</td>
<td>6</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Water</td>
<td>5,975</td>
<td>4,950</td>
<td>17.2%</td>
<td>57.14%</td>
<td>69.11%</td>
</tr>
</tbody>
</table>

Table 4.7: Reduction in Indirect Multicast Diffs with DCR

For FFT-3D, MG, Ocean, and SOR, a large percentage of the original indirect multicast diffs are eliminated. SOR sees the largest reduction, with the DCR implementation receiving 97% fewer indirect diffs than the original implementation. MG and Ocean also use a higher percentage of the remaining indirect diffs as a result of dropping unused pages from the copyset.

The impact on performance of the reductions in indirect diffs received is shown in Figure 4.20, which compares the performance of Brazos with and without DCR for FFT-3D, LU, MG, Ocean, SOR, and Water. In general, the performance benefits of the DCR mechanism are slight, except for FFT-3D in which a 15% reduction in execution time was observed. In LU, unlike FFT-3D, each of the unused multicast
diffs is only sent once per page, meaning that the DCR mechanism does not receive enough samples to change the runtime behavior. Consequently, DCR does not help the performance of LU. SOR sees a decrease of only 3% in execution time despite the 97% reduction in the number of unused indirect diffs received. Because the number of indirect multicast diffs received by processes in this application is small, the impact on performance of eliminating almost all of them is negligible.

![Speedup Chart](image)

**Figure 4.20**: Effect of Dynamic Copyset Reduction Mechanism

There are two reasons why the DCR protocol did not have a larger impact on performance for application such as MG. First, our hardware configuration of fast processors relative to network speed reduces the overhead associated with receiving and applying indirect diffs. In a system with a higher network/processor performance ratio, we expect the DCR protocol to more positively impact the overall execution time of these programs. Second, if the DCR protocol removes a process from a copyset
too soon, the process may incur a new cold miss when the page is next accessed. This means that the entire page must be retrieved from the manager, and outstanding diffs from other processes must also be obtained before the faulting thread can continue. Programs that receive more unused multicast diffs between places where the page is used will therefore benefit more from the DCR protocol. For example, MG receives an average of 10.6 unused multicast diffs for each time a shared page is actually used, whereas ILINK only receives an average of 3.4 unused multicast diffs. Thus it is more advantageous for MG to drop pages from the copyset than ILINK, assuming that the time to process 10 incoming diffs plus the lost computation time while doing so is more than the time to retrieve a page and outstanding diffs from remote processes.

Early Update Mechanism

As described in Section 2.4.1, the early update mechanism helps to eliminate multicast conflicts. As in the case of the DCR algorithm, a counter is associated with each page of shared data and set to an application-specific threshold. Each time a multicast conflict for a page is detected by the runtime system, the counter is decremented. A page is placed on the global early update list when the counter reaches zero. Thereafter, before processes reach a barrier, changes to all pages on the early update list are sent in a single bulk-transfer message to all processes in the aggregate copyset. The recipients of this message store these diffs and apply them after leaving the barrier. This mechanism has three beneficial aspects:

1. Multicast conflicts are avoided, which reduces the amount of overall communication.

2. The access violation faults that would have occured after each barrier are avoided altogether, allowing computation to proceed uninterrupted.

3. Diffs for all early update pages are sent in a few (usually just one) bulk messages, better utilizing the network throughput characteristics of Windows NT as shown
in Figure 3.2.

Pages are able to switch back from an EU protocol to the default invalidation protocol. Processes keep track of unused early updates, and will drop out of the copyset of a page for which a certain number of early updates are not used. When only one process remains in the copyset, the page reverts back to an invalidation protocol.

![Graph showing % of Execution Time](image)

**Figure 4.21: Effect of Early Update Mechanism**

Figure 4.21 shows the performance gains of the EU protocol for ILINK, IS, and CG, which were the three applications that experienced multicast conflicts (Table 1.5). The speedups observed are labeled at the top of each column in Figure 4.21. CG shows the largest improvement with EU, as the speedup improves from 3.62 to 1.64. Figure 4.21 shows that nearly all of the access violation time has been eliminated.
from CG.

ILINK improves by a modest 6% with the addition of EU, which also eliminated many of the access violation faults. Early updates do not provide more benefit to ILINK for two main reasons. First, during each iteration, so many pages are identified as early update that it requires two multicast messages to send the updates before arriving at each barrier instance. Second, only 49% of the early updates received are used before a new update arrives, as opposed to over 89% for CG.

IS does not gain any performance benefit from EU. Only 4% of the received updates are used. This is because we use previous global scope information to determine runtime behavior, and the multicast conflicts in IS do not last over one or two global scopes.

Adaptive Page-Management Protocol

As observed in FFT-3D and MG, processes may write to entire pages of data during the course of one global scope. This may be particularly true in the case of data that exhibits producer-consumer or migratory sharing patterns. In a distributed page-management protocol, diffs for these pages must still be made, even though the diff itself may be as large as the shared page. Protocol and network overhead can be reduced by allowing processes that write to large portions of a page to "own" the page. Other processes that write to the page flush their (presumably) small diffs to the new page owner, and the entire page is retrieved after an invalidation instead of collecting distributed diffs from many processes. This is the default behavior for the Munin [13] DSM system.

We have implemented an adaptive page-management protocol that allows pages to be managed with either a distributed or "home-based" page management protocol, depending on observed data access patterns. The exact methodology used is outlined in Section 3.4.3.

Table 4.8 gives information on the performance of the adaptive page management
<table>
<thead>
<tr>
<th></th>
<th>Execution Time Reduction</th>
<th>Time in Segv Handler</th>
<th>First Write Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Brazos - APM</td>
<td>Brazos</td>
</tr>
<tr>
<td>FFT-3D</td>
<td>10.7%</td>
<td>63</td>
<td>45</td>
</tr>
<tr>
<td>MG</td>
<td>7.8%</td>
<td>62</td>
<td>56</td>
</tr>
</tbody>
</table>

Table 4.8: Effect of Adaptive Page-Management Protocol

protocol (APM) for FFT-3D and MG. In Table 4.8, Brazos - APM refers to Brazos without the adaptive page management protocol. Column 2 gives the percentage reduction in execution time, with FFT-3D and MG gaining a 10.7%, and 7.8% performance improvement, respectively. Columns 3 and 4 show the time spent by each configuration inside the access violation handler. Because only one message has to be received for each page that is managed with a home-based protocol, the total time spent by threads in the handler routine drops by 10-40%, providing the performance improvement. The last two columns show a detrimental effect of employing a home-based protocol. The number of first-write faults goes up by factors of 1.75 and 2.41, adding from .4 to 1.4 seconds to overall execution time. This occurs because processes flush their changes to the home process at every release, and then must change the page protection back to READONLY to catch writes during the subsequent global scopes. The lazy protocol does not have to do this, only changing the page protection when the page is requested by another process or invalidated.

4.3.4 Scalability Issues

In this section, we present results for a DSM system consisting of 16 processors to examine the effect on scalability of the features used in Brazos. The system configuration used is identical to the one used in the experiments in Section 4.3 with the addition of two 4-processor Proliant 5000 servers. The Proliant 5000 servers contain identical processors to those in the 1500, and each has 256 Mbytes of main memory.

Figure 4.22 shows the speedups for Barnes Hut, CG, ILINK, SOR, and Water for
the LRC+MT and MC+SScC protocols, with and without the adaptive runtime features. We used only these 5 applications because they achieved some speedup past 8 processors on at least one DSM implementation. We do not show results for the LRC implementation for two reasons. First, using LRC on the 4-processor 5000's requires starting four separate processes, each with its own shared address space, DSM system overhead, etc. The amount of memory required for four instances of the DSM program exceeds the available physical memory in the 5000's for several of our applications. Secondly, no application that would actually run showed a speedup greater than 1 for LRC past 8 processors.
Figure 4.22: Speedup of Selected Applications on 16 Processors

Figure 4.22 shows that Barnes Hut and CG do not show a performance improvement past 8 processors for the LRC+MT implementation, while the reduction in false sharing communication (in the case of Barnes Hut) and the early update protocol (in the case of CG) allow the multicast-based DSM systems to continue to improve performance.

All three systems show increased performance at 16 processors for both ILINK.
and SOR. In SOR, the three protocols continue to perform nearly identically, but the addition of the two 4-processors machines only improves the overall speedup to 9.1 due to contention for the internal bus (see Section 4.3.7). ILINK also shows an improved performance for all three systems. The effects of the multicast protocol and the early update mechanism allow MC+SScC and Brazos to widen the difference in performance that was seen at 8 processors when 8 additional processors are added to the system.

Finally, Water continues to perform better under LRC+MT due to the overheads imposed by implementing scope consistency.

4.3.5 Maintaining Per-Application Execution Histories

The history mechanism in Brazos can be used to capture performance-related data during runtime at a program-variable granularity, as described in Section 3.5. Brazos currently maintains two types of information in a per-application history file: threshold values derived from previous executions, and generalized performance “hints” to be used during the next execution. In this section, we look how the information produced by the history mechanism can be used to reduce the number of unused multicast diffs for FFT-3D.

The sharing patterns in FFT-3D were shown in Section 4.3 to be unsuitable for a multicast based protocol. One of the performance problems encountered was the receipt of 6000 unused multicast diffs by Process 0 because Process 0 is the initial page manager for all shared pages. In Section 4.3.3, it was shown that the DCR protocol can significantly reduce the number of these unused multicast diffs sent to Process 0. The history mechanism can be used to reduce the number of unused multicast diffs further in the following manner.

First, the program is run under the Brazos system with no previous history information available. The runtime thresholds are set to the default values for a new program execution. We will focus on the threshold value indicating when a page
should be dropped from the current copyset because this is the value of interest for this application. Receiving useless indirect diffs for a page is not ideal, but it is preferable to dropping from a page’s copyset prematurely and then having to re-acquire it across the network. Therefore, the initial threshold value used in Section 4.3.3 was set to the default value of 6. As shown in Table 4.7, the number of received multicast diffs dropped by a factor of 1.91 to 3138 diffs with the addition of the DCR protocol.

After the program is finished, the Brazos post-mortem analysis program examines the number of multicast diffs sent, the number used, and the current thresholds for each variable. Because no indirect diffs were used, and the threshold values are at the threshold limit, a note will be placed in the history file that the threshold should be lowered by half during the next execution. After the next program execution, the number of multicast diffs has dropped 838 diffs. Upon analysis, it is again discovered that no multicast diff was used, so the threshold is lowered again by half (rounded down to 1, which is the lowest allowable threshold value). This produces a drop to just 64 unused diffs during the next program execution. Thus, the total received unused multicast diffs dropped from the initial value of 5987 to just 64.

The runtime system will not adjust the threshold value just discussed down during a program execution because that would favor the more expensive option of dropping out of the copyset early. That is, if the page were accessed again, the more expensive option of having to re-acquire the page across the network would have been favored by the lowering of the threshold. After the program has finished, the decision to lower the threshold can be made with a higher degree of confidence. If, in the next execution the access patterns were to change (i.e., the problem size or configuration changes), the threshold will quickly adjust back up due to the hysteresis factor we used for all threshold values.
4.3.6 Threading Issues

In this section we examine the effect of three thread management techniques on the performance of Brazos parallel programs: thread priority, pinning application threads to processors, and the effects of allowing multiple outstanding network requests per process.

DSM System Thread Priority

In a pre-emptive thread environment such as Windows NT, it is possible to assign different thread priorities to each thread in the system. Here we briefly evaluate the effect on overall performance of changing the thread priority of Brazos threads.

The two Brazos system threads were described in Section 3.4.1. The request thread is responsible for responding to asynchronous requests for data from remote processes, while the reply thread receives responses to requests previously sent out by threads in the local process. In addition, the receipt and processing of indirect multicast diffs is handled by the reply thread.

In general, we have observed little impact on performance of altering the priority of the reply DSM system thread. However, we have observed variations in execution time when the relative priorities between the application threads and asynchronous request thread are changed. Figure 4.23 shows the impact on performance for the 12 applications of varying the priority of the asynchronous response thread and the application threads. The bars show execution time normalized to an execution of the application where all threads (application and DSM system threads) are run at the same priority. The red bars show the affect on performance of lowering the asynchronous request system thread priority. The green bars show what happens when the DSM request thread runs at the highest possible priority, and the blue bars show performance when application threads are given higher priority.

Looking first at the red bars, we see that the majority of the applications perform worse when the request thread is run at a lower priority than the reply and application
Figure 4.23: Effect of Varying Thread Priorities

threads. When the asynchronous request thread is run at a lower priority, remote processes must wait longer for their requests to be satisfied. This can lead to remote threads' timing out and being forced to resend their request if the local OS can not schedule the request thread before the timeout period expires. Alternately, the green bars show that running the request thread at the highest priority benefits a few applications substantially (IS and LU), and minimally impacts the performance of the other applications. The blue bars indicate that raising the application-level threads' above the priority of both DSM system threads negatively impacts all applications by an average of 15%. Brazos uses the configuration corresponding to the green bars in Figure 4.23. However, although users can not change the DSM system thread priorities, they can use the standard Windows API to changes application-thread priorities to any level they wish.
Pinning Threads to Processors

Windows NT allows the user to specify a thread affinity mask for each thread created by the user program. This affinity mask is a bit mask specifying the processors on which the thread is allowed to be scheduled. We have examined the impact of pinning compute threads to specific processors in order to eliminate the increased cache and TLB misses that may result from allowing the OS to schedule a single application thread on both processors in a dual-processor system. A disadvantage to this is that it restricts where the OS can schedule an application thread to be run. Because the DSM asynchronous request thread runs at a higher priority, it will preempt one of the two application threads (Thread A) when an asynchronous message arrives. If the non-preempted application thread (Thread B) must block, the OS will not be able to re-schedule Thread A on the other processor, potentially losing available parallelism.

As the data in Table 4.9 indicates, we generally found a slight performance improvement by explicitly pinning threads to processors. For FFT-3D, this provided a 16% performance improvement, the largest observed. On the other hand, IS and Water performed 6.9% and 5.3% worse, respectively, by restricting the OS in this manner. Our conclusions are that overall, this is not a critical factor to DSM performance. However, with more application threads (i.e., 4 or 8 threads per process), the effects of this may be magnified as the amount of potentially lost parallelism increases. Brazos does not pin application threads to specific processors, although users may choose to do this through the Win32 API.

Allowing Multiple Outstanding Network Requests per Process

In this section we look at the effect on performance of allowing multiple threads per process to have outstanding requests for data. Recall from Section 3.4.1 that this is one of the reasons that Brazos uses a separate reply thread to wait for incoming responses to requests. For performance to improve, the increase in parallelism afforded by allowing multiple outstanding network requests must outweigh the overhead of
<table>
<thead>
<tr>
<th></th>
<th>Non-Pinned Speedup</th>
<th>Pinned Speedup</th>
<th>% Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes Hut</td>
<td>3.91</td>
<td>3.97</td>
<td>+1.5%</td>
</tr>
<tr>
<td>CG</td>
<td>3.62</td>
<td>3.60</td>
<td>-0.6%</td>
</tr>
<tr>
<td>FFT-3D</td>
<td>1.76</td>
<td>2.04</td>
<td>+15.9%</td>
</tr>
<tr>
<td>ILINK</td>
<td>5.65</td>
<td>5.71</td>
<td>+1.1%</td>
</tr>
<tr>
<td>IS</td>
<td>2.92</td>
<td>2.72</td>
<td>-6.9%</td>
</tr>
<tr>
<td>LU</td>
<td>4.0</td>
<td>4.06</td>
<td>+1.5%</td>
</tr>
<tr>
<td>MG</td>
<td>2.40</td>
<td>2.43</td>
<td>+1.3%</td>
</tr>
<tr>
<td>Ocean</td>
<td>0.90</td>
<td>0.94</td>
<td>+4.4%</td>
</tr>
<tr>
<td>Raytrace</td>
<td>1.52</td>
<td>1.53</td>
<td>+0.7%</td>
</tr>
<tr>
<td>SOR</td>
<td>5.68</td>
<td>5.67</td>
<td>-0.2%</td>
</tr>
<tr>
<td>TSP</td>
<td>7.7</td>
<td>7.8</td>
<td>+1.3%</td>
</tr>
<tr>
<td>Water</td>
<td>3.41</td>
<td>3.23</td>
<td>-5.3%</td>
</tr>
</tbody>
</table>

Table 4.9: Effect of Pinning User Threads to Processors

maintaining and scheduling another thread to implement this feature. Table 4.10 summarizes the execution ratios between a Brazos implementation that allows overlapped requests and one that does not.

Table 4.10 indicates a slight to moderate performance advantage to allowing overlapped network requests, especially in the cases of FFT-3D, CG, LU, and Ocean. Several applications, such as Water, Raytrace, and Barnes Hut, actually do better when overlapped requests are not allowed due to a reduction in contention for access to the network interface card. The execution times of ILINK, TSP, and SOR are not affected by this optimization because of the low network requirements of these applications. Unlike the implementation of LRC+MT, Brazos does provide for simultaneous multiple outstanding network requests.

4.3.7 Hardware SMP Performance vs. Brazos DSM Performance

In this section, we use a 4-processor Compaq Proliant 5000 server to compare the performance of three different shared memory configurations on four processors: a hardware implementation that uses the Proliant 5000's hardware cache coherence
<table>
<thead>
<tr>
<th></th>
<th>Execution Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Overlapped</td>
</tr>
<tr>
<td></td>
<td>Non-Overlapped</td>
</tr>
<tr>
<td>Barnes Hut</td>
<td>1.02</td>
</tr>
<tr>
<td>CG</td>
<td>.90</td>
</tr>
<tr>
<td>FFT-3D</td>
<td>.87</td>
</tr>
<tr>
<td>ILINK</td>
<td>1.00</td>
</tr>
<tr>
<td>IS</td>
<td>.99</td>
</tr>
<tr>
<td>LU</td>
<td>.91</td>
</tr>
<tr>
<td>MG</td>
<td>.97</td>
</tr>
<tr>
<td>Ocean</td>
<td>.95</td>
</tr>
<tr>
<td>Raytrace</td>
<td>1.04</td>
</tr>
<tr>
<td>SOR</td>
<td>1.01</td>
</tr>
<tr>
<td>TSP</td>
<td>1.00</td>
</tr>
<tr>
<td>Water</td>
<td>1.02</td>
</tr>
</tbody>
</table>

Table 4.10: Advantage of Allowing Multiple Outstanding Network Requests

mechanism only; a software-only implementation that uses a single processor on each of four 2-processor Compaq Proliant 1500 machines, with Brazos providing coherence between machines; and a hybrid version that uses two Proliant 1500’s with two processors each, maintaining on-machine coherence through hardware and between-machine coherence with the Brazos DSM system. The Proliant 5000 has the same processors as the 1500’s (200 MHz Pentium Pro), the same amount of L2 cache (512 Kbytes) per processor, and 256 Mbytes of main memory. Thus, the differences in speedups presented here are almost exclusively due to the mechanism used to maintain shared memory consistency. Figures 4.24 and 4.25 show the speedup for the three shared memory implementations on the twelve benchmark applications.
Figure 4.24: SMP Performance vs. Brazos DSM Performance (1 of 2)
Figure 4.25: SMP Performance vs. Brazos DSM Performance (2 of 2)

We would expect the hardware version to perform substantially better than either the software-only version or the hybrid implementation because of the high communication costs associated with network transactions. The hybrid implementation can take advantage of local coherence mechanisms for co-located threads, but still must communicate across the network with remote threads. In the software-only version, all inter-thread communication is across the network, but the use of multicast proto-
cols can favorably impact performance over the hybrid system.

For FFT-3D, Barnes Hut, IS, MG, Ocean, Raytrace, and Water, the hardware version performs substantially (> 20%) better than either the software-only or hybrid implementations. Not surprisingly, these applications show the highest message rate as indicated in Figure 4.6, and therefore the increased communication speed of the cache coherent hardware has the biggest impact. For Barnes Hut and Water, the software-only version outperforms the hybrid version slightly (by 5.6% and 8.7%, respectively) because these programs were shown to be able to take advantage of multicast communication. The hybrid version, since there are only 2 processes, does not use multicast communication. The hybrid version performs better than the software-only version for FFT-3D, IS, MG, Ocean, and Raytrace.

In CG, ILINK, LU, and TSP, the hardware version performs slightly to marginally better than the other two implementations. ILINK, LU, and TSP have very low communication requirements (see Figure 4.6), which enables the Brazos versions to compare favorably with the hardware-only system. In ILINK and TSP, the difference between the software-only version and the hybrid version is small, whereas the reduction in cold misses described in Section 4.3 for LU enables the hybrid version to outperform the software-only implementation by 30%. CG is the only application for which the software-only system substantially outperforms the hybrid system. The use of the early update adaptive mechanism combined with multicast provides a 21% improvement in this application.

In SOR, the software-only version executes 15% faster than the hardware-only version, and 6% faster than the hybrid version. This effect has been observed before for this application [27], with the reduction in the amount of data moved cited as a possible reason. Because DSM systems move data based on diffs, if only a portion of a page has been modified, the unmodified portion will not be sent to a requesting process. In a hardware system, entire cache lines are transferred regardless of what has been modified within the cache line. It was postulated that because the interior
elements of the matrix in SOR change very slowly, extra data was being transferred by the hardware system. We tested this claim by duplicating the experiment performed in [27]. We filled the entire matrix with non-zero numbers so that all elements would change with each iteration. No effect on the relative execution times between the three implementations was observed.

A more extreme example of this phenomenon can be seen with matrix multiply. Matrix multiply is trivially parallelizable, and our version of the application (taken from the Munin [14] suite) has no communication between threads other than cold-start misses. We altered Brazos to allow all threads to begin with all pages valid, thereby eliminating even these cold-start misses. Figure 4.26 shows the speedups for the hardware-only, software-only, and hybrid shared memory implementations.

![Figure 4.26: Matrix Multiply Speedups](image)

As indicated by Figure 4.26, the software-only version obtained a perfect speedup.
of 4.0, as expected. However, the hardware-only implementation achieved a speedup of only 1.5 on 4 processors, and the hybrid system obtained a speedup of 2.0 on 4 processors.

The reason for this counter-intuitive result stems from the architecture of the Pentium Pro and the Intel memory bus. The matrix multiply algorithm we use is not blocked, and therefore threads do not enjoy any cache reuse because the cache is fully swept through during each $Row \times Column$ operation. Therefore, there is a substantial amount of bursty traffic that must be handled by the memory bus. The Pentium Pro core clocks at 200 MHz and can support up to 4 simultaneous outstanding bus transactions. The memory bus is 64 bits wide and clocked at only 66 MHz, yielding a peak bandwidth of 528 Mbytes/second, giving enough bandwidth to handle the amount of traffic generated by matrix multiply easily. However, the problem lies in the fact that the bus can only support a total of 8 outstanding transactions from all processors attached to it. Thus, if the program makes little use of the cache, 2 processors can conceivably fill all available buffer space, resulting in delays while accessing main memory. Attaching 4 processors to the bus, as in the Proliant 5000, compounds the problem. The memory bus contention does not exist in the software-only system, and explains the perfect speedup observed.

4.3.8 Summary

The results in this chapter clearly show the performance benefit of using application-level multithreading and selective multicast in cluster-based software DSM systems. Ten out of twelve of the applications studied benefited from the use of multithreading, while eight applications were able to take advantage of selective multicast. The performance benefit of supplying adaptive protocols to ameliorate some adverse effects of multicast was also demonstrated, although the effect on performance was not as pronounced as hoped because of the mismatch between processor and network speeds in the current Brazos implementation. Finally, the results in this chapter have shown
that software scope consistency can significantly reduce the network demands for a specific type of program, although the widespread applicability of scope consistency is not yet widely understood.
Chapter 5

Related Work

5.1 Software DSM Systems

Ivy [37] was the first software-only DSM implementation. Ivy used a single-writer, invalidation page-based, sequentially consistent coherence protocol to maintain coherence between processing nodes, and was prone to high false sharing rates and large amounts of communication. However, Ivy demonstrated that software DSM systems could be a viable alternate to large-scale shared memory multiprocessors.

The Clouds [39] system also utilized sequential consistency, but coherence was maintained on a per-object or per-segment basis. By associating the coherence protocols on a granularity recognizable to the application programmer, Clouds allowed more user interaction with the methods used to provide consistency throughout the system. Objects could be attached to a specific processing node through user-supplied “pin” and “unpin” operations, providing an experienced programmer the opportunity to improve performance through intelligent placement of data objects. Brazos maintains its history mechanism on a variable basis for many of the same reasons.

Another object-based DSM system was Emerald [21]. The Emerald system did not migrate or replicate data objects across processing nodes unless instructed to do so by the programmer. Remote accesses were accomplished through RPC calls, while local communication occurred via shared memory. The use of the RPC call mechanism allowed Emerald to provide fine-grained object control [10], unlike more traditional page-based DSM systems. However, Emerald required a much higher level of programmer involvement in data distribution and movement than other DSM systems, detracting from part of the rationale for the advantage of DSM over straight
message passing.

Amber [16] was derived from Emerald, and provided a shared address space to eliminate some of the complexities of programming present in the Emerald system. Amber was object-based, and used a combination of RPC and data replication to move data between processing nodes. However, programmers still had to ensure a proper decomposition of shared data into objects in order to achieve good performance.

Orca [26, 7] was an object-oriented DSM system that made use of group communication through an efficient, ordered broadcast mechanism. Orca used multicast to provide transparent replication of data objects on different processing nodes. Data was encapsulated in objects specified by the programmer, and no global shared data was allowed. The runtime system detected writes to these data objects, removing the necessity for access violation faults or page protection operations. Brazos uses multicast in much the same way to reduce the amount of network communication necessary to maintain coherence between machines. However, Brazos’ support of the standard shared memory parallel programming paradigm makes for easier porting of existing programs to Brazos as well as providing a programming environment more familiar to most programmers.

Brazos combines the use of data objects and page-based DSM coherence protocols. Data objects are the granularity used for the history mechanism, because these tend not to change between program executions. During runtime, maintaining an encapsulation of these object structures and managing coherence between them is difficult, so Brazos uses page-based coherence protocols similar to DSM systems such as TreadMarks [27] and Munin [14].

The Midway [9] system introduced two new features: entry consistency and software dirty bits. Entry consistency is a relaxed consistency model that attempts to avoid unnecessary data motion by explicitly associating every shared variable with a particular synchronization object, allowing the runtime system to keep coherent only those objects affected by a specific synchronization point in the program. This
can reduce the amount of communication traffic necessary to maintain coherence in a manner similar to scope consistency, but at an added cost of requiring the programmer to provide the mapping between shared data and synchronization. Scope consistency has two advantages over entry consistency. First, the mapping between variables and synchronization is not required to be determined by the programmer. Instead, scope consistency relates synchronization to sections of user code rather than to specific synchronization. Also, the mappings between data and synchronization may change over the course of a program’s execution, further complicating the programmer’s job when using entry consistency. Second, points in a program where the user does wish all data to be made coherent are handled transparently by scope consistency, while the programmer must explicitly specify this desire in entry consistency.

The use of a sophisticated compiler allowed Midway to implement software dirty bits to detect shared writes at the granularity of a cache line. Code was placed around each shared write to determine whether a shared variable is present locally before the write occurs, thus eliminating the need to take segmentation faults and providing a finer granularity of write-detection. There are several disadvantages to this approach, however. First, there is an overhead associated with executing the write-detection code that the compiler places around each write. Furthermore, this overhead is present in every write to shared data, even if the variable is currently only being accessed by a single process in the system. Second, if multiple writes to the same word are made between synchronization points, each write incurs this code overhead, whereas diff creation only captures the last write to each memory location. Other systems after Midway have used software dirty bits to provide fine grained access control [41, 42].

Munin [8, 13, 14] was the first software DSM system to make use of a relaxed consistency model. As outlined in Section 2.1.1, Munin’s buffering of writes before a release operation resulted in substantial communication reduction over a similar sequentially consistent DSM system. Munin also incorporated a number of static and
adaptive performance tuning mechanisms in an attempt to provide the most efficient page management protocol for each page of shared data. Unfortunately, many of Munin's protocols relied on identification of the sharing pattern by the user, limiting its usability. Similar to Munin, Brazos employs performance tuning techniques to tailor program data management to the sharing patterns found in a specific shared memory programs. However, Brazos implements all tuning mechanisms adaptively at runtime, eliminating the requirement for involvement on the part of the user.

TreadMarks [27, 28] introduced the concept of lazy release consistency (LRC), a further refinement to the release consistent protocol introduced by Munin. Instead of pushing invalidation or update messages on each release as is done in Munin, LRC delays the propagation of coherence effects until the following acquire operation. Additionally, TreadMarks makes use of a distributed page management protocol to eliminate the cost of maintaining an always up-to-date copy of each page at each "home node". In general, TreadMark's lazy protocols slightly outperform protocols based on release consistency [27]. Brazos uses a combination of lazy and eager protocols through multicast. Updates to pages are not made until requested, as in TreadMarks, but these updates are multicast to all processes with a copy of the page to bring them up-to-date before they incur an access violation on the page, thus introducing an "eager" aspect to Brazos' behavior.

CVM [29] was one of the first DSM systems to explore the benefits of application-level multithreading and relaxed consistency protocols. CVM is closely related to TreadMarks. CVM employs a multiple-writer, lazy release consistent protocol as the default coherence mechanism, but CVM has been written to allow the easy performance comparison of different protocols. In [48], Thitikamol and Keleher examine the performance impact of application-level multithreading on an SMP implementation of CVM. However, only one processor per machine was used to reduce the amount of contention for the network interface. Unlike Brazos, which implements a hybrid hardware/software system by using available hardware coherence mechanisms, CVM
currently only uses multithreading as a latency-hiding technique, switching application threads when a network operation is required. Results in [48] show that while multithreading used in this manner can be beneficial, good performance is not always achievable for non-trivial applications.

The Millipede [24] project provides a simple programming interface and portability, while implementing adaptive measures such as thread migration and load balancing. Millipede is the only other software DSM system we are aware of that is currently designed for use under the Windows NT operating system, although there are several other such systems currently under development. Millipede has been designed to use several coherence protocols, including release consistency, lazy release consistency, and sequential consistency. The majority of the focus in the Millipede project has been on minimizing the effect of running DSM systems as background jobs, without disrupting the reaction time to foreground jobs that may be running by users logged on to the system. Brazos has instead focused on achieving optimum performance on a dedicated network of PC’s.

The Shasta [41] DSM system allows access to data at a fine granularity, and allows the coherence granularity to vary across shared data structures. Shasta implements its version of fine-grained access control by inserting code before each load or store to check if the data is available locally, communicating with other processes if necessary, as is done in Blizzard-S [42]. Several techniques have been added to the basic check to reduce the runtime overhead, including careful layout of the shared address space, explicit instruction scheduling of the checking code, and batching together checks for a number of loads and stores simultaneously. Even though fine grained access control provides a partial solution to the problem of false sharing, recent studies have shown that while the extra number of bytes sent as a result of false sharing may be high, the increase in the overall number of messages is relatively small [4].
5.2 Hardware and Hybrid DSM Systems

Release consistency was developed as part of the DASH multiprocessor project [36]. In DASH, writes are pipelined to *mask the latency of write operations*. In contrast, most software DSM systems such as Brazos, Munin, and TreadMarks buffer writes until the next synchronization to *reduce the number of required messages*. DASH is organized as a network of multiprocessors nodes, with coherence between the nodes maintained by a directory mechanism located in the network interface of each node. Consistency is on the basis of cache lines, and therefore DASH does not suffer the same potential performance problems associated with false sharing seen in page-based DSM systems.

The FLASH [31] architecture supports a variety of communication models including shared memory and message passing through the use of the MAGIC (Memory And General Interconnect Controller) programmable controller. The MAGIC chip contains a protocol processor that allows users to develop their own coherence protocols while providing low-latencies pathways between processing nodes.

The Alewife [15] machine is a large-scale multiprocessor that also provides for both shared memory and message passing. Each node is made of a custom "Sparkle" integer unit, derived from the SPARC processor, and a portion of main memory. Nodes communicate on a mesh interconnect, and the memory coherence protocol is implemented by a custom communication and memory management unit. Alewife’s principle disadvantage is that it is not constructed from commodity parts, and changing the hardware must result in a complete overhaul of the entire system. It is unclear how results derived from such a system translate to today’s processor and network speeds.

APRIL [2] employed a sequential consistency model, but used application-level multithreading combined with fast processor context-switching to mask network latencies. In order to achieve good performance, tasks in programs for April must be written at a fine level of granularity, so that there are several application threads...
available for scheduling when a thread blocks for a network transaction.

The SHRIMP multicomputer [11] is constructed from highly integrated commodity parts, consisting of Pentium PC's with the same routing network used in the Intel Paragon. SHRIMP supports automatic update release consistency [22], which is a hardware assist to shared memory that automatically flushes writes to a home process. Remote processes then retrieve an entire pages of valid data from the home process on an access violation, as is done in Munin. AURC has been shown to provide substantial performance benefits over a software-only coherence scheme, but custom hardware is required.

The CASHMERe [30] prototype consists of 32 processors constructed from eight 4-processor DEC 2100 4/233 multiprocessors interconnected through a Memory Channel network. The Memory Channel plugs into a PCI slot, and allows processors to read and write to memory located on remote machines directly, similar to the mechanism used to implement AURC in the SHRIMP multicomputer. While the Memory Channel provides a mechanism for implementing fine grained sharing in the CASHMERe architecture, the authors in [30] found that overall, performance increased only slightly over that of more coarse-grained systems such as TreadMarks. The authors cite severe limitations in the Memory Channel network hardware as reasons for their results.

SoftFLASH [18] provides an aggressive kernel-level implementation of a virtual shared memory system constructed from a network of SGI multiprocessor clusters using on-cluster hardware coherence mechanisms for local accesses and a software DSM system for remote data accesses. Overall, they concluded that the approach offers promise, but the need for lower latency, higher bandwidth interconnect networks still keeps the performance of such systems below optimum. Our results support this conclusion, but the communication network we employ is a factor of eight slower than that used in the SoftFLASH work, and therefore we observe lower performance than that reported in [18].
Finally, the MGS (multigrain system) system [52] provides for a large-scale multiprocessor that is constructed from clusters of small to medium scale NUMA processors. The authors claim their results apply to machines based on SMP machines as well, but they chose NUMAs as the building blocks for scalability reasons. Coherence within a cluster is maintained at a cache line granularity, while coherence between clusters is maintained using the virtual shared memory protocol introduced by Munin [13], supported by the active message layer. Although similar in concept to Brazos, MGS is currently implemented on top of the Alewife multiprocessor and relies heavily on active message support.

5.3 Runtime Adaptive Protocols

There have been several protocols that adapt at runtime to program behavior. In [13], Carter et al. describe the update with timeout mechanism employed in Munin. In this scheme, updates are passed between processes with copies of the same page, and a timer is set when each update arrives. If the timer goes off before the update is used, a message is sent to inhibit further updates. This mechanism is similar to the dynamic copyset reduction algorithm used in Brazos, except that Brazos counts the number of unused indirect diffs instead of relying on a timing device.

In [3], an adaptive protocol that switched between a single writer protocol and a multiple writer protocol was examined. The benefit of a single writer protocol is mainly that less overhead must be used to keep track of coherence information than in a multiple writers protocol. A multiple writers protocol generally provides better performance when there is a high degree of false or write-write sharing. Brazos does not employ such an adaptive mechanism, but a large reduction in the memory overhead to maintain coherence is achieved in Brazos through the use of multicast.

The effects of false sharing in page-based DSM systems was examined in [4]. The authors conclude that while false sharing contributes significantly to the number of extra bytes that must be sent to maintain coherence, the number of messages resulting
from false sharing is few for many applications. A protocol that aggregates pages together to capture the benefits of prefetching shows some performance improvement without a substantial increase in the false sharing rates. Brazos achieves the same effect through the use of the early update mechanism for data that exhibits a specific sharing pattern. False sharing in Brazos is further addressed by the use of the scope consistency coherence protocol.
Chapter 6

Conclusions and Future Work

The goal of the Brazos project was to show that parallel scientific computing can be efficiently carried out on a network of low-cost, high performance personal computers. We achieved this through the development of a suite of protocols that makes use of multithreading to allow the efficient use of available SMP computers, selective multicast to reduce the amount of necessary coherence traffic, adaptive runtime mechanisms to ameliorate some adverse effects of multicast, software scope consistency to reduce the amount of false sharing, and a unique history mechanism that retains performance information at the program variable granularity across program executions. By utilizing features available for the first time in a Windows-based operating system, we have shown that Brazos outperforms current state-of-the-art DSM systems.

Through detailed analysis of twelve scientific applications, we show that providing a multithreaded DSM environment can improve performance of software DSMs that run on SMP computers. When compared to TreadMarks, the current state-of-the-art in software DSM systems, Brazos achieved an average performance improvement of 83%. The principal contributions to this superior performance were the use of multicast communication and application-level multithreading.

Unsurprisingly, application-level multithreading provided the largest overall performance improvement across most applications studied. Because the use of multithreading reduces the number of processes that must use network primitives in half for the system we studied, a corresponding reduction in the number of messages sent was observed for the majority of the application suite.

We have demonstrated that the use of selective multicast in software DSM systems
can reduce the required network communication. We both delay the creation of diffs until required, and provide processes with data in advance of when it is needed, thus combining the advantages of lazy DSM systems with those of more eager systems. The use of multicast in Brazos reduced the network communication rate by an average of 38% over a conventional point-to-point DSM system based on multithreaded lazy release consistency.

The Brazos design and performance validates our initial assumption that a software DSM system based on multicast network primitives can be more scalable than a comparable software DSM system based on point-to-point messages, despite the widely held belief that multicast would limit the scalability of software DSM systems. Because of the high cost of the communication primitives used in software DSM systems, especially those systems based on networks of workstations or PC’s, these systems are not scalable to begin with. The use of multicast, instead of reducing the overall scalability, actually improves scalability over a point-to-point system through a reduction in the number of DSM messages that must be sent to execute a shared-memory parallel application.

In some situations, side-effects from the use of multicast can limit performance improvements. unless steps are taken to reduce excessive multicast traffic. We identified two of these effects, unused multicast diffs and multicast conflicts, as performance problems in several of our applications. To address these problems, we developed and implemented two adaptive runtime protocols: dynamic copyset reduction and early update. Dynamic copyset reduction, while effective in reducing the amount of multicast traffic, had little impact on overall performance due to the large mismatch between network communication and processing speeds present in our system. Early update was effective at reducing the number of multicast conflicts. This adaptive protocol lead to a 17% average performance increase for the three applications that exhibited multicast conflicts. A third adaptive protocol was also implemented that allows pages to be handled in either a home-based protocol, similar to [14], or a dis-
tributed page based protocol similar to [27]. This protocol was marginally successful, but again its performance-benefit was limited due to the computation/communication mismatch of 200 MHz processors with a 100 Mbps network.

Brazos uses scope consistency to reduce the amount of false sharing in programs that make use of frequent, short critical sections. One of the four applications that makes use of locks showed significant performance improvement with the use of software scope consistency. Our implementation represents the first all-software scope consistency protocol used in distributed page-based systems. Previous scope consistency protocols have been for use in hardware-assisted DSM systems, or home-based software DSMs.

We believe that scope consistency provides a more intuitive model of how programmers view the relationship between synchronization and shared data than that provided by other relaxed consistency models. This is due to the implicit association that scope consistency makes between when shared data is modified (e.g., in a global or local sense) and when it is guaranteed to be coherent. Although we found limited benefits from the use of scope consistency in current shared memory applications, we believe that shared memory programs constructed “from scratch” can take advantage of the benefits of scope consistency to better improve performance.

Finally, Brazos introduced a unique history mechanism that maintains information about the performance of our adaptive protocols across program executions to allow faster adaptation the next time a program is run. Information is kept at a program variable level instead of on a page-by-page basis to make information applicable to a specific program even when the program problem size or system configuration changes. Runtime information is maintained on a page basis during execution, and is converted to variable-based information after the completion of the program. The process is reversed when the program is next run, with variable information being mapped onto the pages that contain the variables after the DSM system has been started.

We have shown that the improving price/performance ratio of networks of per-
sonal computers has finally reached the point where the scientific community should recognize their potential as high-end multiprocessor systems. By making use of available multicast primitives, operating system support for multithreading, and advanced runtime mechanisms, high-performance software distributed shared memory systems can be implemented in this new and exciting computing environment.

**Future Work**

Although software DSM systems have gained in research popularity in the last decade, there remains little use or development of these systems in the commercial marketplace. We see three main areas for work in DSM systems in order to make their use more widespread:

- **Fault tolerance.** Commercial enterprises often place more emphasis on data security and integrity than on raw parallel performance. A principle challenge in the development of new distributed shared memory systems will be how well fault tolerance and check-pointing techniques can be integrated into the performance-oriented aspect that DSM researchers have focused on. In particular, research into developing database applications for use in DSM systems must be accompanied by concurrent research in the area of DSM fault tolerance.

- **Multiprogramming.** We believe that multiprogramming can be used as effectively in the context of DSM systems as it is used in modern operating systems. In Brazos, the vast majority of access faults, and thus communication, occurs immediately after a synchronization point. Figure 6.1 shows the time from the last synchronization point to each access fault in ILINK. Because our communication patterns are so clustered, as indicated in Figure 6.1, the network is utilized very little during large portions of many applications’ execution. By simultaneously running several applications, we believe we can fill these “communication gaps” with useful communication for other DSM programs. We are
currently developing such a DSM system that can accommodate multiple independent user applications simultaneously, allowing a greater overlap between computation and long communication latencies.

- Improved communication primitives. As this thesis has shown, the commonly available communication primitives carry significant overhead associated with their use. Some work, such as the Fast Sockets [40] project at the University of California, has shown that by stripping out all but the bare essentials from the network communication protocol stack, significant reductions in the overhead associated with communication primitives can be obtained.

Plans are in place to pursue research in each of these three areas using the Brazos DSM system presented here. By incorporating these and other improvements into the existing DSM framework, we believe that software DSM systems in general, and Brazos in particular, will begin to make inroads into the commercial and business marketplaces.
Bibliography


FIGURE A.2.2. Intracellular calcium responses of BAEC to venous (1.3 dynes/cm$^2$) and arterial (13 dynes/cm$^2$) levels of shear stress. Data point represent the average ± SEM calculated from 62 cells from 3 different experiments for venous and 56 cells from 3 different experiments for arterial shear stress levels.
FIGURE A.2.3. Fura-2 *in situ* calibration in HUVEC.
FIGURE A.2.4. Intracellular calcium responses of HUVEC to venous (1.3 dynes/cm²) and arterial (13 dynes/cm²) levels of shear stress. Data points represent the average ± SEM calculated from 40 cells from 2 different experiments for venous and 39 cells from 2 different experiments for arterial shear stress levels.
FIGURE A.2.5. Fura-2 *in situ* calibration in HASMC.
FIGURE A.2.6. Intracellular calcium responses of HASMC to venous (1.3 dynes/cm²) and arterial (13 dynes/cm²) levels of shear stress. Data point represent the average ± SEM calculated from 47 cells from 6 different experiments for venous and 37 cells from 6 different experiments for arterial shear stress levels.
Similar observations for endothelial cells (Dull and Davies, 1991; James et al., 1995; Mo et al., 1991; Nollert et al., 1991; Nollert and McIntire, 1992; Patrick and McIntire, 1995c; Schilling et al., 1992) as well as for smooth muscle cells (Geiger et al., 1992) have been published before. Note that in chapter 5 we demonstrated by 3D fluorescence microscopy that there is a flow-induced nuclear calcium increase in endothelial cells, which is masked by blurring effects in 2D studies.

The BCECF calibration curves for BAEC and HUVEC are shown in figures A.2.7 and A.2.9, correspondingly (see Figure 4.1 for HASMC). \( \Delta pHi \) has been defined in section 4.2.5. \( pHi \) responses to venous (1.3 dynes/cm\(^2\)) and arterial (13 dynes/cm\(^2\)) levels of shear stress of BAEC, HUVEC, and HASMC are shown in figures A.2.8, A.2.10, and 4.2, correspondingly. Shear-induced intracellular acidification has been reported before in rat (Ziegelstein et al., 1992) and bovine (Patrick and McIntire, 1995a) endothelial cells. Our results are in agreement with the results of these authors for the fact that venous shear stress levels induce acidification followed by a recovery phase, whereas arterial shear stress levels induce "stable" acidification. The flow-induced endothelial acidification has been attributed to activation of the Cl\(^-\)/HCO\(_3^-\) exchanger on the membrane of endothelial cells possibly due to convective mass transfer-induced changes of the local HCO\(_3^-\) concentrations (Patrick and McIntire, 1995a; Ziegelstein et al., 1992). In chapter 4 we reported a different response of the smooth muscle cells, namely a shear-induced alkalinization [see also (Stamatas et al., 1997)]. Furthermore, we demonstrated that the response was due to activation of the Na\(^+\)/H\(^+\) exchanger by flow.
FIGURE A.2.7. BCECF in situ calibration in BAEC.
FIGURE A.2.8. Intracellular pH responses of BAEC to venous (1.3 dynes/cm²) and arterial (13 dynes/cm²) levels of shear stress. Data point represent the average ± SEM calculated from 20 cells from 3 different experiments for venous and 60 cells from 4 different experiments for arterial shear stress levels.
FIGURE A.2.9. BCECF \textit{in situ} calibration in HUVEC.
FIGURE A.2.10. Intracellular pH responses of HUVEC to venous (1.3 dynes/cm$^2$) and arterial (13 dynes/cm$^2$) levels of shear stress. Data points represent the average ± SEM calculated from 48 cells from 3 different experiments for venous and 36 cells from 2 different experiments for arterial shear stress levels.
Table A.2.1 summarizes the flow-induced responses in the vascular cells presented here and the corresponding exchangers involved. The significance of these observations is the existence of different mechanotransduction pathways in different cell types.

<table>
<thead>
<tr>
<th></th>
<th>([\text{Ca}^{2+}]_i)</th>
<th>(\text{pH}_i)</th>
<th>Ion Exchanger Activated</th>
</tr>
</thead>
<tbody>
<tr>
<td>EC (BAEC, HUVEC)</td>
<td>[No]*</td>
<td>acidification ↓</td>
<td>Cl(^{-})/HCO(_3)^{-}</td>
</tr>
<tr>
<td>SMC (HASMC)</td>
<td>No</td>
<td>alkalinization ↑</td>
<td>Na(^{+})/H(^{+})</td>
</tr>
</tbody>
</table>

Table A.2.1. Summary of the intracellular ionic shear-induced responses in vascular endothelial and smooth muscle cells as observed by 2D fluorescence microscopy.

* In chapter 5 we have demonstrated by 3D fluorescence microscopy and deconvolution that nuclear calcium increases in BAEC in response to shears stress.
A.3. Optical Properties of the Microscope: Lateral Resolution and Depth of Field

The lateral resolution of an epifluorescence microscope, MR (nm), is given by (Patrick and McIntire, 1995c):

\[ MR = \frac{0.61 \cdot \lambda}{\text{NA}} \]  \hspace{1cm} (A.3.1)

where \( \lambda \) is the wavelength of the emitted fluorescence (nm) and NA is the numerical aperture of the objective lens. For the Fura-2 experiments NA = 0.95 and \( \lambda = 510 \) nm, therefore MR = 327 nm = 0.33 \( \mu \)m. In the case of the BCECF experiments we have NA = 0.95 and \( \lambda = 535 \) nm, therefore MR = 343 nm = 0.34 \( \mu \)m.

The depth of field of a microscope, MD (nm), is the distance along the focal axis throughout which a single point object can be moved without changing the observed image (Patrick and McIntire, 1995c) and it is given by:

\[ MD = \frac{\lambda}{4 \cdot n \cdot \sin^2\left(\frac{\theta}{2}\right)} \]  \hspace{1cm} (A.3.2)

where \( n \) is the refractive index of the objective's immersion medium (for dry objective \( n=1.0 \)), and \( \theta \) is the half-angle (rad) of the cone of light that is captured by the objective given by:
\[ \theta = \sin^{-1} \left( \frac{\text{NA}}{n} \right) \]  

(A.3.3)

For our objective NA = 0.95 and \( \theta = 1.25 \) rad. For Fura-2, \( \lambda = 510 \) nm and MD = 370 nm = 0.37 \( \mu \)m, whereas for BCECF, \( \lambda = 535 \) nm and MD = 389 nm = 0.39 \( \mu \)m.
REFERENCES


Determine Shear Stress-Induced Nitric Oxide Production in Native Endothelial Cells. *Circulation Research* 78, 750-758.


Endothelial Cells. Ph.D. Thesis: Department of Chemical Engineering, Rice University, Houston, TX.


