High-Performance All-Software Distributed Shared Memory

Kirk L. Johnson
Laboratory for Computer Science
Massachusetts Institute of Technology
545 Technology Square
Cambridge, MA 02139, U.S.A.

MIT Laboratory for Computer Science
Technical Report MIT/LCS/TR-674
18 December 1995

High-Performance All-Software Distributed Shared Memory

by

Kirk Lauritz Johnson

S.B., Massachusetts Institute of Technology (1989)
S.M., Massachusetts Institute of Technology (1989)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1996

© Massachusetts Institute of Technology 1996. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
18 December 1995

Certified by . . . .
Anant Agarwal
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Certified by . . . .
M. Frans Kaashoek
Assistant Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

High-Performance All-Software
Distributed Shared Memory
by
Kirk Lauritz Johnson

Submitted to the Department of Electrical Engineering and Computer Science on 18 December 1995 in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Abstract

The *C Region Library* (CRL) is a new all-software distributed shared memory (DSM) system. CRL requires no special compiler, hardware, or operating system support beyond the ability to send and receive messages between processing nodes. It provides a simple, portable, region-based shared address space programming model that is capable of delivering good performance on a wide range of multiprocessor and distributed system architectures. Each region is an arbitrarily sized, contiguous area of memory. The programmer defines regions and delimits accesses to them using annotations.

CRL implementations have been developed for two platforms: the Thinking Machines CM-5, a commercial multicomputer, and the MIT Alewife machine, an experimental multiprocessor offering efficient hardware support for both message passing and shared memory. Results are presented for up to 128 processors on the CM-5 and up to 32 processors on Alewife.

Using Alewife as a vehicle, this thesis presents results from the first completely controlled comparison of scalable hardware and software DSM systems. These results indicate that CRL is capable of delivering performance that is competitive with hardware DSM systems: CRL achieves speedups within 15% of those provided by Alewife's native hardware-supported shared memory, even for challenging applications (*e.g.,* Barnes-Hut) and small problem sizes.

A second set of experimental results provides insight into the sensitivity of CRL's performance to increased communication costs (both higher latency and lower bandwidth). These results demonstrate that even for relatively challenging applications, CRL should be capable of delivering reasonable performance on current-generation distributed systems.

Taken together, these results indicate the substantial promise of CRL and other all-software approaches to providing shared memory functionality and suggest that in many cases special-purpose hardware support for shared memory may not be necessary.

Thesis Supervisor: Anant Agarwal
Title: Associate Professor of Computer Science and Engineering

Thesis Supervisor: M. Frans Kaashoek
Title: Assistant Professor of Computer Science and Engineering

Acknowledgments

This thesis marks the culmination of my six-and-a-half year career as a graduate student at MIT. During that time, I was extremely fortunate to work in a stimulating, vibrant environment populated with wonderful people. This thesis reflects their influence in many ways, both directly and indirectly.

I am especially indebted to Anant Agarwal, my advisor. Anant's keen insight, boundless enthusiasm, and quick wit helped foster an ideal environment in which to take one's first steps in Science. What I learned from working with Anant will undoubtedly continue to influence my thinking for the rest of my life.

Frans Kaashoek arrived at MIT roughly halfway through my tenure as a graduate student. We began to work together almost immediately, a fact for which I will always be grateful. Frans's supervision, encouragement, and friendship had unmeasurable impact on both this work and my time as a graduate student. His willingness to wade through endless drafts of papers (and this thesis) and provide careful commentary each and every time never ceased to amaze me.

I would also like to thank the third member of my thesis committee, Barbara Liskov. Her careful reading of the thesis document and "outside" perspective on this research helped improve the presentation greatly.

David Chaiken, David Kranz, John Kubiatowicz, Beng-Hong Lim, Anne McCarthy, Ken MacKenzie, Dan Nussbaum, and Donald Yeung were my nearly constant companions and good friends over the past six years. Without their tireless efforts to make Alewife a reality and provide a first-rate research environment, much of this and other interesting research would not have been possible. I feel privileged to have had the opportunity to work with such fine people for such a prolonged period of time; it is with some sadness that I part ways with them.

Debby Wallach, Wilson Hsieh, Sandeep Gupta, and Joey Adler were the first guinea pigs subjected to CRL. Their assistance in porting applications, finding and fixing bugs, and producing a version of CRL that was fit for public consumption was invaluable. CRL also benefited greatly in its early stages from feedback and commentary from John Guttag, Kevin Lew, Margo Seltzer, Dan Scales, Bill Weihl, and Willy Zwaenepoel.

Playing ice hockey with other students and several faculty members once a week (year 'round, Tuesdays from 11pm to midnight) provided a welcome break from tooling and helped keep me sane. People particularly worthy of mention in this context are Mike Noakes, who provided the encouragement and opportunity to start playing, John Buck, who loaned me his equipment until I bought my own, and Ken MacKenzie, my defensive partner for the past several years.

Numerous other residents of the sixth floor provided camaraderie and conversation during my time at MIT, including Rajeev Barua, Robert Bedichek, Ricardo Bianchini, Fred Chong, Stuart Fiske, Matt Frank, Steve Keckler, Kathy Knobe, Rich Lethin, Sramana Mitra, Mike Noakes, and Ravi Soundararajan. For those that have already left MIT, I wish them luck in their future endeavors. For those that have not, I wish them godspeed.

Chris Metcalf shared an office with me for almost my entire time as a graduate student; David Goddeau and Ed Hurley shared the office with us during parts of that time. Chris, David, and Ed deserve a special thanks for putting up with my loud music and sometimes unusual habits.

My first few semesters as a graduate student were supported by Victor Zue and the Spoken Language Systems group at MIT. I was fortunate both as a recipient of said support and for the opportunity it provided me to further the knowledge of speech and signal processing I had developed while working on my Master's Thesis.

Finally, without Heather's constant love, support, and tolerance of my rather unusual work schedule, none of this would have been possible. This thesis is dedicated to her.

A condensed version of parts of this work appears in *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, December 1995 [34].

This research was supported in part by the Advanced Research Projects Agency under contract N00014-94-1-0985, by an NSF National Young Investigator Award, by Project Scout under ARPA contract MDA972-92-J-1032, and by a fellowship from the Computer Measurement Group.

Contents

1 Introduction **13**
1.1 Distributed Shared Memory 14
1.2 The C Region Library 16
1.3 Contributions and Results 18
1.4 Thesis Roadmap 18

2 Distributed Shared Memory **19**
2.1 Implementation Techniques 21
2.1.1 Software DSM 21
2.1.2 Hardware DSM 23
2.2 Mechanisms for DSM 24
2.3 Discussion 25

3 The CRL DSM System **26**
3.1 Goals 26
3.2 Programming Model 27
3.3 Global Synchronization Primitives 31
3.4 Memory/Coherence Model 32
3.5 Discussion 33

4 CRL Internals **35**
4.1 Overview 35
4.2 Region Identifiers 35
4.3 Metadata 36
4.3.1 Common Components 36
4.3.2 Home-side Components 38
4.3.3 Remote-side Components 39
4.3.4 Data Layout 39
4.4 Mapping and Unmapping 40
4.5 Caching 40
4.6 Coherence Protocol 42
4.6.1 Three-Party Optimization 43
4.7 Global Synchronization Primitives 44
4.8 Status 44

4.9 Summary . . . 45

5 Experimental Platforms **46**
5.1 CM-5 . . . 46
5.1.1 Interrupts vs. Polling . . . 46
5.1.2 Whither Polling . . . 47
5.1.3 Communication Performance . . . 53
5.2 Alewife . . . 54
5.2.1 Integrated Support for Shared Memory and Message Passing . . . 54
5.2.2 Communication Performance . . . 55
5.2.3 Status . . . 56

6 Results **57**
6.1 Basic Latencies . . . 58
6.2 Applications . . . 60
6.2.1 Blocked LU . . . 61
6.2.2 Water . . . 61
6.2.3 Barnes-Hut . . . 61
6.2.4 Performance . . . 62
6.3 CRL vs. Shared Memory . . . 67
6.4 Changing Communication Costs . . . 69
6.5 Sensitivity Analysis . . . 71
6.5.1 Modified Alewife CRL Implementation . . . 72
6.5.2 Experimental Results . . . 73
6.6 Summary and Discussion . . . 83

7 Related Work **86**
7.1 Region- and Object-Based Systems . . . 86
7.2 Other Software DSM Systems . . . 88
7.3 Comparative Studies . . . 89

8 Conclusions **90**
8.1 Summary . . . 90
8.2 Goals Revisited . . . 91
8.3 Future Work . . . 92

A Coherence Protocol **94**
A.1 Protocol States and Events . . . 94
A.2 Home-Side State Machine . . . 97
A.3 Remote-Side State Machine . . . 112
A.4 Continuations and Home-Side 'Iip' States . . . 124
A.5 Remote-Side 'Req' States . . . 125
A.6 Protocol Message Format . . . 125
A.7 Atomicity . . . 126
A.8 Naming Regions . . . 128

A.9 Unexpected Messages 129
A.10 Two Kinds of Invalidation Messages 130
A.11 Out-of-Order Message Delivery 131

B Raw Data **135**
B.1 Interrupts vs. Polling 135
B.2 Global Synchronization: Hardware vs. Software 139
B.3 Basic CRL Latencies 140
B.4 Application Characteristics 145
B.5 Alewife CRL Profiling 148
B.6 Sensitivity Analysis 149

List of Figures

2-1 System with memory as single, monolithic resource. 20
2-2 System with distributed memory. 20
2-3 DSM implementation alternatives. 20

3-1 CRL implementation of `cons`. 29
3-2 CRL implementation of `car` and `cdr`. 30
3-3 CRL implementation of mutual-exclusion locks. 31

4-1 Region data layout. 39
4-2 Implementation of the unmapped region cache (URC). 41

5-1 Basic structure of the synthetic workload. 48
5-2 Communication events used in the synthetic workload. 49
5-3 Typical synthetic workload performance data. 51
5-4 Performance of 'poll' synthetic workload case relative to 'intr'. 52
5-5 Basic Alewife architecture. 54

6-1 Absolute running time and speedup for Blocked LU. 64
6-2 Absolute running time and speedup for Water. 65
6-3 Absolute running time and speedup for Barnes-Hut. 66
6-4 Breakdown of normalized running time for Alewife CRL version of Blocked LU. 68
6-5 Breakdown of normalized running time for Alewife CRL version of Water. 69
6-6 Breakdown of normalized running time for Alewife CRL version of Barnes-Hut. 70
6-7 Barnes-Hut performance for larger problem and machine sizes. 72
6-8 Impact of increased message latency on application performance. 75
6-9 Impact of decreased bulk transfer bandwidth on application performance. . 76
6-10 Contour plots indicating combined impact of increased communication costs on Blocked LU. 78
6-11 Contour plots indicating the combined impact of increased communication costs on Water. 79
6-12 Contour plots indicating the combined impact of increased communication costs on Barnes-Hut (4,096 bodies). 80
6-13 Contour plots indicating the combined impact of increased communication costs on Barnes-Hut (16,384 bodies). 81

A-1 HomeExclusive: state transition diagram. 99
A-2 HomeExclusiveRip: state transition diagram. 100
A-3 HomeExclusiveWip: state transition diagram. 101
A-4 HomeShared: state transition diagram. 102
A-5 HomeSharedRip: state transition diagram. 103
A-6 HomeIip: state transition diagram. 104
A-7 HomeIipSpecial: state transition diagram. 105
A-8 HomeInvalid: state transition diagram. 106
A-9 RemoteInvalid: state transition diagram. 113
A-10 RemoteInvalidReq: state transition diagram. 114
A-11 RemoteShared: state transition diagram. 115
A-12 RemoteSharedReq: state transition diagram. 116
A-13 RemoteSharedRip: state transition diagram. 117
A-14 RemoteModified: state transition diagram. 118
A-15 RemoteModifiedRip: state transition diagram. 119
A-16 RemoteModifiedWip: state transition diagram. 120
A-17 Standard (non-data-carrying) protocol message format. 126
A-18 The "flush-invalidation" problem. 128
A-19 The "late invalidation" problem. 131
A-20 The "early invalidation" problem. 132
A-21 The "late release" problem. 133
A-22 The "late invalidation acknowledgement" problem. 133

List of Tables

3-1 Summary of the CRL interface. 27
3-2 Global synchronization functions in CRL. 32

4-1 Metadata elements common to both home and remote copies of regions. . 37
4-2 Metadata elements specific to home copies of regions. 38
4-3 Metadata elements specific to remote copies of regions. 39

6-1 Measured CRL latencies, 16-byte regions. 58
6-2 Measured CRL latencies, 256-byte regions. 59
6-3 Static count of source lines and CRL calls for the three applications. . . . 60
6-4 Approximate number of regions used and typical region sizes for the three applications. 60
6-5 Application running times. 62
6-6 Application characteristics when running under CRL. 63

A-1 CRL home-side protocol states. 95
A-2 CRL remote-side protocol states. 95
A-3 CRL call events. 95
A-4 CRL home-to-remote protocol messages. 96
A-5 CRL remote-to-home protocol messages. 96
A-6 HomeExclusive: protocol events and actions. 107
A-7 HomeExclusiveRip: protocol events and actions. 107
A-8 HomeExclusiveWip: protocol events and actions. 107
A-9 HomeShared: protocol events and actions. 108
A-10 HomeSharedRip: protocol events and actions. 109
A-11 HomeIip: protocol events and actions. 109
A-12 HomeIipSpecial: protocol events and actions. 110
A-13 HomeInvalid: protocol events and actions. 111
A-14 RemoteInvalid: protocol events and actions. 121
A-15 RemoteInvalidReq: protocol events and actions. 121
A-16 RemoteShared: protocol events and actions. 121
A-17 RemoteSharedReq: protocol events and actions. 122
A-18 RemoteSharedRip: protocol events and actions. 122
A-19 RemoteModified: protocol events and actions. 122
A-20 RemoteModifiedRip: protocol events and actions. 123
A-21 RemoteModifiedWip: protocol events and actions. 123

B-1 Synthetic workload performance data, two-message communication events. 136
B-2 Synthetic workload performance data, three-message communication events. 137
B-3 Synthetic workload performance data, four-message communication events. 138
B-4 CM-5 CRL application running times, HW *vs.* SW synchronization. . . . 139
B-5 Dynamic counts of global synchronization primitives. 140
B-6 Events measured by latency microbenchmark. 141
B-7 CM-5 CRL latencies. 142
B-8 Alewife CRL latencies, with CMMU workarounds. 143
B-9 Alewife CRL latencies, without CMMU workarounds. 144
B-10 Call event counts for Blocked LU. 146
B-11 Call event counts for Water. 146
B-12 Call event counts for Barnes-Hut. 147
B-13 Message counts for 32 processors. 147
B-14 Breakdown of Alewife CRL running times. 148
B-15 Running times for Blocked LU on 16 processors using the modified Alewife CRL implementation. 149
B-16 Running times for Blocked LU on 32 processors using the modified Alewife CRL implementation. 150
B-17 Running times for Water on 16 processors using the modified Alewife CRL implementation. 151
B-18 Running times for Water on 32 processors using the modified Alewife CRL implementation. 152
B-19 Running times for Barnes-Hut (4,096 bodies) on 16 processors using the modified Alewife CRL implementation. 153
B-20 Running times for Barnes-Hut (4,096 bodies) on 32 processors using the modified Alewife CRL implementation. 154
B-21 Running times for Barnes-Hut (16,384 bodies) on 16 processors using the modified Alewife CRL implementation. 155
B-22 Running times for Barnes-Hut (16,384 bodies) on 32 processors using the modified Alewife CRL implementation. 156

Chapter 1

Introduction

Distributed systems with communication performance rivaling that of traditional tightly-coupled multicomputer systems are rapidly becoming reality. In order for these “networks of workstations” (NOWs) to be a viable route to cost-effective high-performance computing, good programming environments are necessary; such environments must be both easy to use and capable of delivering high performance to the end user.

The *C Region Library* (CRL) is a new all-software *distributed shared memory* (DSM) system intended for use on message-passing multicomputers and distributed systems. CRL requires no special compiler, hardware, or operating system support beyond the ability to send and receive messages. It provides a simple, portable, region-based shared address space programming model that is capable of delivering good performance on a wide range of multiprocessor and distributed system architectures. Each region is an arbitrarily sized, contiguous area of memory; programmers define regions and delimit accesses to them using annotations.

CRL implementations have been developed for two platforms: the Thinking Machines CM-5, a commercial multicomputer, and the MIT Alewife machine, an experimental multiprocessor offering efficient support for both message passing and shared memory. This thesis presents results for up to 128 processors on the CM-5 and up to 32 processors on Alewife. In a set of controlled experiments, we demonstrate that CRL is the first all-software DSM system capable of delivering performance competitive with hardware DSMs. CRL achieves speedups within 15 percent of those provided by Alewife’s native support for shared memory, even for challenging applications and small problem sizes.

The rest of this chapter defines and provides further motivation for “good” distributed shared memory systems (Section 1.1), briefly describes CRL (Section 1.2), and summarizes the contributions and major results of the thesis as a whole (Section 1.3).

1.1 Distributed Shared Memory

There is a growing consensus that parallel systems should support a *shared address space* or *shared memory* programming model: programmers should not bear the responsibility for orchestrating all interprocessor communication through explicit messages. Support for such distributed shared memory systems can be provided in hardware, software, or some combination of the two. In general, hardware distributed shared memory systems allow programmers to realize excellent performance without sacrificing programmability. Software DSM systems typically provide a similar level of programmability, but trade off somewhat lower performance for reduced hardware complexity and cost—the hardware required to implement a message-passing system (upon which a software DSM is built) is typically less complex and costly than that required to provide aggressive hardware support for shared memory, especially for systems with hundreds or thousands of processors.

The appeal of a shared-address space programming model over a message-passing one arises primarily from the fact that a shared-address space programming model frees the programmer from the onus of orchestrating all communication and synchronization through explicit message-passing. While such coordination can be managed without adversely affecting performance for relatively simple applications (*e.g.,* those that communicate infrequently or have sufficiently static communication patterns), doing so can be far more difficult for large, complex applications (*e.g.,* those in which data is shared at a very fine grain or according to irregular, dynamic communication patterns) [72, 73]. For applications in which the complexity of using a message-passing programming model remains manageable, one can often realize better performance using message-passing instead of shared-memory. However, these gains can be relatively modest [11, 16] and frequently come at the cost of greatly increased programmer effort.

In spite of this fact, message passing environments such as PVM (*Parallel Virtual Machine*) [24] and MPI (*Message Passing Interface*) [55] are often the *de facto* standards for programming multicomputers and networks of workstations. This is primarily due to the fact that these systems are portable. They require no special hardware, compiler, or operating system support, thus enabling them to run entirely at user level on unmodified, "stock" systems. DSM systems that lack this ability are unlikely to gain widespread acceptance as practical vehicles for high-performance computing.

In order to fulfill the promise of parallel and distributed systems as a cost-effective means of delivering high-performance computation, DSM systems must possess four key properties: *simplicity*, *portability*, *efficiency*, *scalability*.

Simplicity: Because DSM systems provide a uniform model for accessing all shared data, whether local or remote, they are often relatively easy to use. Beyond such *ease of use*, however, good DSM systems should exhibit several other forms of simplicity. First, DSM systems should provide *simple interfaces* that allow them to be platform- and language-independent. This form of simplicity has fundamental impact on portability (as discussed below); DSM systems with simple interfaces that are not bound to platform- or language-specific features are more likely to be portable. Second, DSM systems should be amenable to *simple implementations*. This form of simplicity reduces the overhead of designing, building, debugging, and deploying a production system. Of equal or greater importance, however, is the fact that systems with simple implementations are far more likely to afford *extension*, *modification*, and *customization*, whether to improve performance or to explore using them in unforeseen ways.

Portability: Portability across a wide range of platforms and programming environments is particularly important, because it obviates the odious task of having to rewrite large, complex application codes once for each different target platform. In addition to being portable across "space" (the current generation of parallel and distributed systems), however, good DSM systems should also be portable across "time" (able to run future systems). This type of portability is particularly important because it enables *stability*; without stability, it is unlikely that DSM systems will ever be an appealing platform for large, production applications requiring development efforts measured in tens of man-years.

Efficiency: All parallel systems are capable of delivering the potential performance of the underlying hardware for some appropriately-limited domain (*e.g.,* applications that require only trivial interprocessor communication or those for which all communication patterns can be statically identified at compile-time). In order to achieve widespread acceptance, DSM systems should be capable of providing high efficiency over as wide a range of applications as possible (especially challenging applications with irregular and/or unpredictable communication patterns) without requiring Herculean programmer effort.

Scalability: In order to provide an appealing platform for high-performance computing, DSM systems should be able to run efficiently on systems with hundreds (or potentially thousands) of processors. DSM systems that scale well to large systems offer end users yet another form of stability: the comfort of knowing that applications running on small- to medium-scale platforms could run unchanged and still deliver good performance on large-scale platforms (assuming sufficient application parallelism). Even though many production applications may never actually be deployed in large-scale environments, systems lacking artificial limits that would prevent such deployment are likely to be a more appealing platform for application development.

The challenge in building good DSM systems system lies in providing all four of these properties to the greatest extent possible; to not sacrifice too much in one area to excel in another.

1.2 The C Region Library

The fundamental question addressed in this thesis is what hardware support and software tradeoffs are necessary in order to enable good DSM systems that possess these key properties. In order to address this question, the thesis presents the design, implementation, and evaluation of the *C Region Library* (CRL), a new all-software DSM system intended for use with message-passing multicomputers and distributed systems.

CRL is referred to as an *all-software* DSM system because it migrates all shared-memory functionality out of hardware into software; the only functionality CRL requires from the platform (*i.e.,* hardware and operating system) upon which it is built is the ability to send and receive messages. Furthermore, CRL is implemented entirely as a library against which application programs are linked; CRL requires no special compilers, binary rewriting packages, program execution environments, or other potentially complex, non-portable software tools. Because of these features, CRL scores well in the portability department; porting the original (Thinking Machines' CM-5 [48]) CRL implementation to two other platforms (the MIT Alewife machine [1] and a network of Sun workstations communicating with one another using TCP) proved to be straightforward. In addition, by eliminating the need for special-purpose hardware to implement DSM functionality, the implementation effort required to build a system like CRL (or other software-based systems) is greatly reduced.

In terms of simplicity, CRL also does well. Like other DSM systems, CRL provides a uniform model for accessing all shared data, whether local or remote, so it is relatively easy to program. In addition, CRL provides a simple programming model that is system- and language-independent. Although the implementation of CRL used in this research only provides a C language interface, providing the same functionality in other programming languages would require little work. Finally, CRL is amenable to simple implementations: the version of CRL described in this thesis consists of just over 9,200 lines of well-commented C code and supports three platforms with significantly different communication interfaces. As such, CRL should be prove to be relatively easy to extend, modify, and customize as the need arises.

Finally, CRL is efficient and scalable. In a set of controlled experiments (using the MIT Alewife machine), this thesis demonstrates that CRL is capable of delivering application speedups (on 32 processors) within 15 percent of those those provided by systems with aggressive hardware support for shared-memory functionality, even for challenging applications (*e.g.,* Barnes-Hut) and small problem sizes. To achieve these performance levels, however, CRL requires high-performance communication mechanisms with latency and bandwidth characteristics closer to those available as (hardware) primitives to designers of hardware-based DSM systems. Since systems providing this level of communication performance are not widely available, the thesis also evaluates the impact of changing communication costs on delivered application performance. This evaluation is accomplished both through sensitivity analysis of the high-performance implementation and by measuring the performance of CRL when running on a platform (Thinking Machines'

CM-5) with communication performance similar to that available in current-generation distributed systems (networks of workstations). While the use of less-efficient communication mechanisms does lead to lower performance (for the applications and problem sizes used in this study, speedups approximately 30 to 40 percent smaller), experiments indicate that acceptable performance on large-scale systems (*e.g.,* 128 processors) may still be possible for larger, more realistic problem sizes.

The work described in this thesis builds on a large body of research into the construction of software DSM systems, but these four key properties distinguish CRL from other software DSM systems. Proper subsets of these features have appeared in previous systems, but CRL is the first to provide all four in a simple, coherent package. Chapter 7 provides further discussion comparing CRL with other software DSM systems.

In order to achieve this level of simplicity, portability, efficiency, and scalability, however, CRL requires one compromise: a modest deviation from "standard" shared-memory programming models. Parallel applications built on top of CRL share data through *regions*. Each region is an arbitrarily sized, contiguous area of memory named by a unique region identifier. Region identifiers comprise a separate address space that is shared among all processors and is distinct from each processor's private, local address space. In order to access data contained in a region, programmers are responsible for (1) inserting calls to CRL functions that manage translations between region identifiers and the local address space and (2) delimiting accesses to region data with calls to CRL functions that initiate and terminate *operations*.

Annotations of the second sort (delimiting the start and end of accesses to shared data) are similar to those necessary in aggressive hardware and software DSM implementations (*e.g.,* those providing release consistency [25]) when writing to shared data. CRL requires such annotations whether reading or writing to shared data, similar to entry consistency [5]. Experience with the applications described in this thesis indicates that the additional programming overhead of providing these annotations is quite modest.

Annotations of the first sort (related to managing translations from region identifiers to local addresses) are necessary because CRL maintains separate local and global address spaces; these represent a more significant deviation from standard shared-memory models. These annotations could be eliminated entirely (perhaps at a slight performance penalty for some applications) by integrating their functionality into the region access functions, but doing so would not address the more fundamental issue of CRL making an explicit distinction between local and global address spaces. Addressing this issue will likely require leveraging off of virtual memory mechanisms or other efficient address translation techniques; whether this can be done without significantly impacting on simplicity, portability, and efficiency remains a subject of future research.

1.3 Contributions and Results

The primary contributions and results of this thesis are fourfold:

- A detailed description of the design and implementation of CRL, a new all-software distributed shared memory system; CRL is unique in providing the four key properties suggested in Section 1.1 in a simple, coherent package.
- The first completely controlled comparison of scalable hardware and software DSM systems, a comparison in which only the communication interfaces used by the programming systems are changed; all other system components (*e.g.,* compiler, processor, cache, interconnection network) remain fixed.
- A demonstration that when built upon efficient communication mechanisms, an all-software DSM system like CRL can deliver application performance competitive with hardware-supported DSM systems, even for challenging applications and small problem sizes. For domains and applications that can tolerate a modest deviation from "standard" shared memory programming models, these results cast doubt on the value of providing hardware support for shared memory functionality.
- An analysis of how sensitive CRL's performance is to increased communication costs, both higher latency and lower bandwidth. These results indicate that even for current-generation networks-of-workstations technology, systems like CRL should be able to deliver reasonable performance, even for relatively challenging applications.

1.4 Thesis Roadmap

The rest of this thesis is organized as follows. Chapter 2 discusses issues related to DSM systems, including motivating shared-memory programming models, a brief discussion of general implementation techniques and issues, and a framework for classifying DSM systems in terms of three basic mechanisms. Chapter 3 provides an "external" perspective of CRL: in addition to describing the CRL programming model, this chapter discusses the goals and context that motivated CRL's design and implementation. Chapter 4 provides the complementary "internal" perspective by describing in detail the implementation of the prototype CRL implementation. Chapter 5 describes the experimental platforms used in this research. Chapter 6 presents performance results for CRL and compares them with Alewife's native shared memory support, both in terms of low-level features and delivered application performance; the latter sections of this chapter address the impact of increased communication costs on CRL's performance. Chapter 7 provides a brief overview of related work. Finally, Chapter 8 revisits the major points of the thesis, discusses their implications, and identifies some areas for future work.

Chapter 2

Distributed Shared Memory

As discussed in the introduction, one of the goals of this research is to identify what hardware support is necessary to enable good DSM systems. Before addressing this issue, it is important to put it in context, by understanding (in broad terms) what kinds of DSM systems (both hardware and software) have been previously built or proposed and what the general implementation issues, advantages, and disadvantages in such systems are. Thus, this chapter discusses issues related to the implementation of DSM systems. The first section provides a brief overview of general implementation techniques and issues. The second section presents a framework for classifying DSM systems in terms of three basic mechanisms.

Before discussing implementation schemes, however, it is important to clarify what is meant by the term "Distributed Shared Memory":

Distributed denotes a property of the implementation of a programming system. In particular, it implies that the physical memory used to implement the shared address space is not a centralized, monolithic resource (see Figure 2-1). Instead, it is distributed across distinct memory modules, the number of which scales with the number of processors in the system (see Figure 2-2). This research assumes systems in which the numbers of processors and memory modules are always the same, and, in fact, pairs of memory modules and processors are tightly coupled into single processing nodes. Other configurations are possible.

Shared memory denotes a property of the programming system as viewed by the end-user. From the application programmer's point of view, all user computation in such a system takes place in a single, global address space that is shared by all threads; communication between threads is effected by reading and writing locations in that address space.

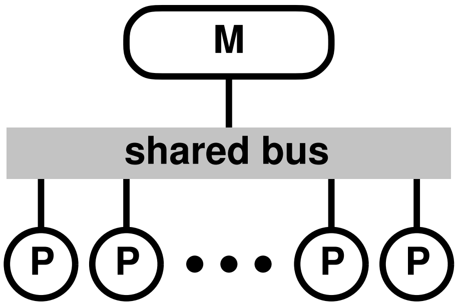


Figure 2-1. System with memory as single, monolithic resource.

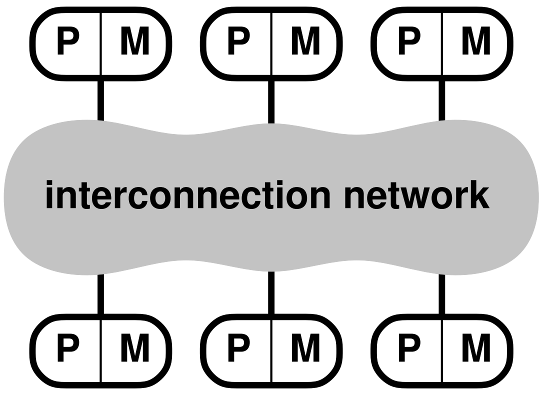


Figure 2-2. System with distributed memory.

Figure 2-3. DSM implementation alternatives.

For the purposes of this thesis, programming systems that implement a shared-address space by directly mapping operations on the shared-address space into message-passing constructs (*e.g.,* Split-C [81], Concurrent Smalltalk [29]) are considered to be message-passing programming systems, not DSM systems.

2.1 Implementation Techniques

This section provides a brief overview of traditional DSM implementation schemes (see Figure 2-3). The first part addresses software schemes intended for message-passing multicomputers or networks of workstations. The second discusses schemes in which DSM functionality is provided by specialized hardware.

2.1.1 Software DSM

A software DSM system is one in which all interprocessor communication is performed through explicit message passing; any shared-address space functionality is synthesized in software by compilers or run-time systems.

Generally speaking, there are two broad classes of software DSMs. In the first class, all message sends and receives are scheduled at compile time (by the compiler). In the second class of software DSMs, communication is scheduled at run time by the run-time or operating system in response to program actions. Accordingly, this proposal uses the terms *static software DSM* and *dynamic software DSM* to refer to members of the first and second class, respectively.

Static Approaches

Static software DSM systems are typified by compilers for FORTRAN-style scientific codes targeting message-passing multicomputers [9, 41, 51, 64, 79, 87]. These are typically *data-parallel* systems with a single thread of control; parallelism can only be expressed in the form of a large number of similar (possibly identical) operations applied in parallel to elements of large, dense arrays according to some user-specified iteration space (parallel loop nest). Inter-iteration dependencies may or may not be respected.

In these systems, storage for each array is partitioned amongst the processing nodes at compile time according to user-supplied or compiler-derived distributions. In the simplest schemes, a single partition of each array is used for the duration of an entire application run; more sophisticated systems allow different partitions for each array within each parallel loop nest.

Parallelism is typically obtained according to some form of the *owner-computes rule*: each operation is executed on the processing node that "owns" the array element modified by that operation, where ownership is determined by whatever array partitions are in effect for the loop nest in question. The illusion of a shared-address space is provided by the compiler: for each operation that reads data resident on a remote node, the compiler schedules appropriate message sends and receives such that the current values of all necessary data are available before the operation is executed. Because of the owner-computes rule, no inter-processor communication is required to write the result of each operation, so the compiler need only worry about scheduling communication for the remote data read by each operation.

Most compiler optimizations in static software DSM systems focus on reducing communication overhead without unnecessarily throttling parallelism. In practice, such optimizations attempt to reduce the total number of messages sends and receives; this is primarily an artifact of the large fixed overhead of message-based communication in many current multiprocessor systems. For applications in which the source and destination of messages can be identified at compile time and communication overhead can be reduced to acceptable levels, static software DSMs have proven to be quite successful. For applications that do not meet these requirements, the extra cost of resorting to *run-time resolution* to determine message endpoints [64] and increased communication overhead can easily overwhelm any potential benefits due to the exploitation of parallelism. Although recent research has yielded some progress on reducing message-based communication overhead [81] and supporting efficient execution of certain kinds of data-dependent communication patterns [50, 67], the applicability of the static software DSM approach appears to remain fairly limited.

Dynamic Approaches

Dynamic software DSM systems typically support a more general programming model than their static counterparts, typically allowing multiple independent threads of control to operate within the shared address space [4, 5, 11, 21, 37, 52, 75]. Given mechanisms for inter-thread synchronization (*e.g.,* semaphores, barriers), a programmer is able to express essentially any form of parallelism.

For the most part, these systems utilize a *data-shipping* paradigm in which threads of computation are relatively immobile and data items (or copies of data items) are brought to the threads that reference them. These systems exploit the *locality of reference* that frequently exists in individual processors' address reference patterns by migrating or replicating *data units* such that most accesses can be satisfied locally without any interprocessor communication or synchronization. They differ primarily in the sizes of data units used (*e.g.,* cache lines, virtual memory pages), the mechanisms used to implement data replication and migration, and the memory/coherence model they provide to the programmer [57] (and thus the details of the protocol used to implement coherence).

Systems utilizing a data-shipping paradigm must address the *cache coherence* problem. When copies of data units are cached close to processors, the system implementation must ensure that the effects of memory references (loads and stores) to a data unit from multiple processors match those specified by the memory/coherence model that is to be supported. For example, in a DSM system that supports a sequentially consistent memory model, a processor that wants to modify (write to) a data unit may need to obtain an exclusive copy (by invalidating any copies of the data unit cached on other processors) before performing the modification.

An alternate approach involves moving computation to the data it references. Systems organized along these lines avoid the overhead of frequent remote communication by migrating computation to the node upon which frequently referenced data resides [10, 65]. Implementations utilizing both computation- and data-migration techniques are also possible [4, 11, 32].

As with static software DSMs, the high fixed overheads of message-based communication in many current generation systems drive dynamic software DSM implementors toward optimizations that reduce the total number of message sends and receives. In addition, because many dynamic software DSM systems assume a relatively low-bandwidth communication substrate (*e.g.,* conventional local area networking technology), these systems also often incorporate optimizations aimed at reducing total communication bandwidth.

Dynamic software DSM systems have proven to be capable of executing a fairly wide range of application classes efficiently, including many irregular, dynamic codes that would be difficult to express in the data-parallel style required by most static software DSM systems, let alone execute efficiently in that style. On the other hand, assuming similar hardware infrastructure to level the playing field, it is reasonable to expect that for a limited class of applications (those for which static software DSM approaches yield extremely efficient code) the application performance delivered by a dynamic software DSM may lag somewhat behind that provided by a static software DSM.

2.1.2 Hardware DSM

A hardware DSM system is one in which all interprocessor communication is effected through loads and stores to locations in a shared global address space. Examples include the NYU Ultracomputer [26], IBM RP3 [62], Stanford DASH [49], and KSR-1 [39]. Other communication mechanisms (*e.g.,* message passing) are synthesized in software using the shared-memory interface. Like dynamic software DSMs, hardware DSM systems support a very general programming model.

Current hardware DSMs typically provide automatic migration and replication of cache-line sized data units (16 to 128 bytes); support for migration and replication is provided by specialized hardware. While the data migration in such systems is inherently dynamic, for applications with completely static communication patterns, sophisticated compilers can apply prefetching techniques to approximate the behavior of a static DSM system [58].

If only a modest number of processors are to be supported (perhaps up to a few dozen), the complexity of a hardware DSM implementation can be reduced substantially through the use of a bus-based organization in which all processor-memory communication traverses a bus shared by all processing nodes (see Figure 2-1). By *snooping* all bus transactions and modifying cache line states appropriately, the caches in each processing node can kept coherent [22]. While such systems are well-understood and relatively simple to build, they are not scalable beyond a modest number of processors. Because this thesis focuses on scalable DSM systems, it assumes that the complexity of implementing hardware DSM systems is on the order of that required for systems based on scalable, general-purpose interconnection networks.

As was the case for dynamic software DSM systems, hardware DSMs have proven to be capable of executing a wide range of application classes efficiently. Compared to dynamic software DSM systems, their primary advantage appears to by the ability to support extremely frequent, fine-grained interprocessor communication and synchronization [21].

2.2 Mechanisms for DSM

This section presents a framework for classifying and comparing dynamic software and hardware DSM systems. The framework identifies three basic mechanisms required to implement dynamic DSM functionality; systems are classified according to whether those mechanisms are implemented in hardware or software. Although this classification scheme is primarily intended for use with DSM systems that employ a data shipping model, it could likely be generalized for use with other kinds of dynamic software and hardware DSM systems.

The three basic mechanisms required to implement dynamic DSM functionality are as follows:

Hit/miss check (processor-side): Decide whether a particular reference can be satisfied locally (*e.g.,* whether or not it hits in the cache).

Request send (processor-side): React to the case where a reference cannot be satisfied locally (*e.g.,* send a message to another processor requesting a copy of the relevant data item and wait for the eventual reply).

Memory-side: Receive a request from another processor, perform any necessary coherence actions, and send a response.

Observing whether these mechanisms are implemented in hardware or software yields the following breakdown of the spectrum of dynamic DSM systems and implementation techniques that have been discussed in the literature.

All-Hardware In all-hardware DSM systems, all three of these mechanisms are implemented in specialized hardware; the Stanford DASH multiprocessor [49] and KSR-1 [39] are typical all-hardware systems.

Mostly Hardware As discussed in Section 5.2, the MIT Alewife machine implements a mostly hardware DSM system—processor-side mechanisms are always implemented in hardware, but memory-side support is handled in software when widespread sharing is detected [13]. Dir_1SW and its variations [27, 85] are also mostly hardware schemes.

The Stanford FLASH multiprocessor [46] and Wisconsin Typhoon architecture [63] represent a different kind of mostly hardware DSM system. Both of these systems implement the request send and memory-side functionality in software, but that software runs on a specialized coprocessor associated with every processor/memory pair in the system; only "memory system" code is expected to be run on the coprocessor.

Mostly Software Many software DSM systems are actually mostly software systems in which the hit/miss check functionality is implemented in hardware (*e.g.,* by leveraging off of virtual memory protection mechanisms to provide access control). Typical examples of mostly software systems include Ivy [52], Munin [11], and TreadMarks [38]; coherence units in these systems are the size of virtual memory pages.

Blizzard [70] implements a similar scheme on the CM-5 at the granularity of individual cache lines. By manipulating the error correcting code bits associated with every memory block, Blizzard can control access on a cache-line by cache-line basis.

All-Software In an all-software DSM system, all three of the mechanisms identified above are implemented entirely in software (*e.g.,* Orca [3]). Several researchers have recently reported on experiences with all-software DSM systems obtained by modifying mostly software DSM systems such that the "hit/miss check" functionality is provided in software [70, 86].

2.3 Discussion

Generally speaking, for applications where static software techniques cannot be effectively employed, increased use of software to provide shared-memory functionality tends to decrease application performance because processor cycles spent implementing memory system functionality might otherwise have been spent in application code. This thesis demonstrates, however, that given a carefully-designed shared memory interface and high-performance communication mechanisms, it is possible to implement all shared memory functionality entirely in software and still provide performance with hardware-based systems on challenging shared-memory applications.

Chapter 3

The CRL DSM System

This chapter provides an "external" perspective of the C Region Library (CRL), describing both the goals and context that motivated CRL's design and implementation and the CRL programming model. This chapter describes CRL in terms of the C language bindings provided by our current implementation; it would be straightforward to provide similar bindings in other imperative languages.

In terms of the classification presented in Section 2.2, CRL is an all-software DSM system. Furthermore, CRL is implemented as a library against which user programs are linked; no special hardware, compiler, or operating system support is required.

3.1 Goals

Three major goals guided the design and implementation of CRL; these goals can be thought of as operational interpretations of the key properties (simplicity, portability, efficiency, and scalability) suggested in the introduction.

- First and foremost, we strove to preserve the essential "feel" of the shared memory programming model without requiring undue limitations on language features or, worse, an entirely new language. In particular, we were interested in preserving the uniform access model for shared data (whether local or remote) that most DSM systems have in common.

- Second, we were interested in a system that could be implemented efficiently in an all-software context and thus minimized the functionality required from the underlying hardware and operating system. Systems that take advantage of more complex hardware or operating system functionality (*e.g.,* page-based mostly software DSM systems) can suffer a performance penalty because of inefficient interfaces for accessing such features [86].

Function	Effect	Argument
`rgn_create`	Create a new region	Size of region to create
`rgn_delete`	Delete an existing region	Region identifier
`rgn_map`	Map a region into the local address space	Region identifier
`rgn_unmap`	Unmap a mapped region	Pointer returned by `rgn_map`
`rgn_rid`	Returns region identifier of a region	Pointer returned by `rgn_map`
`rgn_size`	Returns size (in bytes) of a region	Pointer returned by `rgn_map`
`rgn_start_read`	Initiate a read operation on a region	Pointer returned by `rgn_map`
`rgn_end_read`	Terminate a read operation on a region	Pointer returned by `rgn_map`
`rgn_start_write`	Initiate a write operation on a region	Pointer returned by `rgn_map`
`rgn_end_write`	Terminate a write operation on a region	Pointer returned by `rgn_map`
`rgn_flush`	Flush the local copy of a region	Pointer returned by `rgn_map`

Table 3-1. Summary of the CRL interface.

- Finally, we wanted a system that would be amenable to simple and lean implementations in which only a small amount of software overhead sits between applications and the message-passing infrastructure used for communication.

In light of the our experience with CRL and the results presented in Chapter 6, Section 8.2 discusses the extent to which CRL meets these goals.

3.2 Programming Model

Table 3-1 summarizes the interface provided by CRL. In the CRL programming model, communication is effected through operations on *regions*. Each region is an arbitrarily sized, contiguous area of memory named by a unique region identifier. The memory areas representing distinct regions are non-overlapping. New regions can be created dynamically by calling `rgn_create` with one argument, the size of the region to create (in bytes); `rgn_create` returns a region identifier for the newly created region. Thus `rgn_create` can be thought of as the CRL analogue to `malloc`. (There is no CRL analogue to `realloc`, however; once created, regions cannot be dynamically resized.)

A region identifier is a portable and stable name for a region (other systems use the term "global pointer" for this concept). Region identifiers comprise a separate address space that is shared among all processors and is distinct from each processor's private, local address space. Region identifiers are of abstract type `rid_t`. In order to ensure region identifiers can be manipulated (*e.g.,* used as arguments in a procedure call) without undue overhead, implementations of CRL are expected to employ a compact (scalar) representation for items of type `rid_t`.

Before accessing a region, a processor must *map* it into the local address space using the `rgn_map` function. `rgn_map` takes one argument, a region identifier, and returns a pointer to the base of the region's data area. A complementary `rgn_unmap` function allows the processor to indicate that it is done accessing the region, at least for the time being. Any number of regions can be mapped simultaneously on a single node, subject to the limitation that each mapping requires at least as much memory as the size of the mapped region, and the total memory usage per node is ultimately limited by the physical resources available. The address at which a particular region is mapped into the local address space may not be the same on all processors. Furthermore, while the mapping is fixed between any `rgn_map` and the corresponding `rgn_unmap`, successive mappings on the same processor may place the region at different locations in the local address space.

Because CRL makes no guarantees about the addresses regions get mapped to, applications that need to store a "pointer" to shared data (*e.g.,* in another region as part of a distributed, shared data structure) must store the corresponding region's unique identifier (as returned by `rgn_create`), *not* the address at which the region is currently mapped. Subsequent references to the data referenced by the region identifier must be preceded by calls to `rgn_map` (to obtain the address at which the region is mapped) and followed by calls to `rgn_unmap` (to clean up the mapping). This is illustrated in Figures 3-1 and 3-2, which show a simple CRL implementation of `cons`, `car`, and `cdr` that could be used to build shared data structures.

After a region has been mapped into the local address space, its data area can be accessed in the same manner as a region of memory referenced by any other pointer: no additional overhead is introduced on a per-reference basis. CRL does require, however, that programmers group accesses to a region's data area into *operations* and annotate programs with calls to CRL library functions to delimit them. Two types of operations are available: *read* operations, during which a program is only allowed to read the data area of the region in question, and *write* operations, during which both loads and stores to the data area are allowed. Operations are initiated by calling either `rgn_start_read` or `rgn_start_write`, as appropriate; `rgn_end_read` and `rgn_end_write` are the complementary functions for terminating operations. These functions all take a single argument, the pointer to the base of the region's data area that was returned by `rgn_map` for the region in question. An operation is considered to be *in progress* from the time the initiating `rgn_start_`*op* returns until the corresponding `rgn_end_`*op* is called. CRL places no restrictions on the number of operations a single processor may have in progress at any one time. The effect of loads from a region's data area when no operation is in progress on that region is undefined; similarly for stores to a region's data area when no write operation is in progress. Figures 3-1, 3-2, and 3-3 provide examples of how these functions might be used in practice.

In addition to providing *data access* information (indicating where programs are allowed to issue loads and stores to a region's data area), operations also serve as a primitive *synchronization* mechanism in CRL. In particular, write operations are serialized with respect to all other operations on the same region, including those on other processors. Read operations to the same region are allowed to proceed concurrently, independent of the processor on which they are executed. If a newly initiated operation conflicts with those already in progress on the region in question, the invocation of `rgn_start_`*op* responsible for initiating the operation spins until it can proceed without conflict. As such,

```
typedef struct
{
  rid_t car;
  rid_t cdr;
} CRLpair;

rid_t CRLpair_cons(rid_t car, rid_t cdr)
{
  rid_t    rslt;
  CRLpair *pair;

  /* create a region for the new pair, map it, and initiate a
   * write operation so we can fill in the car and cdr fields
   */
  rslt = rgn_create(sizeof(CRLpair));
  pair = (CRLpair *) rgn_map(rslt);
  rgn_start_write(pair);

  pair->car = car;
  pair->cdr = cdr;

  /* terminate the write operation and unmap the region
   */
  rgn_end_write(pair);
  rgn_unmap(pair);

  return rslt;
}
```

Figure 3-1. CRL implementation of `cons`.

```
typedef struct
{
  rid_t car;
  rid_t cdr;
} CRLpair;

rid_t CRLpair_car(rid_t pair_rid)
{
  rid_t    rslt;
  CRLpair *pair;

  /* map the pair region and initiate a read operation
   * so we can read the value of the car field
   */
  pair = (CRLpair *) rgn_map(pair_rid);
  rgn_start_read(pair);

  rslt = pair->car;

  /* terminate the read operation and unmap the region
   */
  rgn_end_read(pair);
  rgn_unmap(pair);

  return rslt;
}

rid_t CRLpair_cdr(rid_t pair_rid)
{
  rid_t    rslt;
  CRLpair *pair;

  /* map the pair region and initiate a read operation
   * so we can read the value of the cdr field
   */
  pair = (CRLpair *) rgn_map(pair_rid);
  rgn_start_read(pair);

  rslt = pair->cdr;

  /* terminate the read operation and unmap the region
   */
  rgn_end_read(pair);
  rgn_unmap(pair);

  return rslt;
}
```

Figure 3-2. CRL implementation of `car` and `cdr`.

```
/* map the region named by rid and acquire
 * mutually-exclusive access to it
 */
void *mutex_acquire(rid_t rid)
{
  void *rslt;

  rslt = rgn_map(rid);
  rgn_start_write(rslt);

  return rslt;
}

/* release mutually-exclusive access to rgn
 * and unmap it
 */
void mutex_release(void *rgn)
{
  rgn_end_write(rgn);
  rgn_unmap(rgn);
}
```

Figure 3-3. CRL implementation of mutual-exclusion locks.

traditional shared-memory synchronization primitives like mutual-exclusion or reader-writer locks can be implemented in a straightforward manner using CRL operations (see Figure 3-3).

In addition to functions for mapping, unmapping, starting operations, and ending operations, CRL provides a handful of other functions relating to regions. First, CRL provides a *flush* call that causes the local copy of a region to be flushed back to whichever node holds the master copy of the region (this node is referred to as the *home node* for the region; it is discussed further in Chapter 4). By selectively flushing regions, it may be possible to reduce future coherence traffic (*e.g.,* invalidations) related to the flushed regions. Flushing a region is analogous to flushing a cache line in hardware DSM systems. Second, CRL provides two simple functions that can be used to determine the region identifier and size (in bytes) of a mapped region (`rgn_rid` and `rgn_size`, respectively).

3.3 Global Synchronization Primitives

In addition to the basic region functionality described in the previous section, CRL provides a modest selection of primitives for effecting global synchronization and communication. These primitives are summarized in Table 3-2.

Function	Effect
rgn_barrier	Participate in a global barrier
rgn_bcast_send	Initiate a global broadcast
rgn_bcast_recv	Receive a global broadcast
rgn_reduce_dadd	Participate in a global reduction (sum)
rgn_reduce_dmin	Participate in a global reduction (minimum)
rgn_reduce_dmax	Participate in a global reduction (maximum)

Table 3-2. Global synchronization functions in CRL.

`rgn_barrier` can be used to effect a global synchronization point. `rgn_barrier` takes no arguments; it does not return on any node until it has been called on all nodes.

`rgn_bcast_send` and `rgn_bcast_recv` provide a means for one (sending) processor to broadcast information to all other (receiving) processors. The sending processor calls `rgn_bcast_send` with two arguments (the number of bytes to broadcast and a pointer to a buffer containing the data to be sent). All other processors must call `rgn_bcast_recv` with a size argument matching that provided on the sending processor and a pointer to an appropriately-sized receive buffer; calls to `rgn_bcast_recv` return after all broadcast data has been received locally.

`rgn_reduce_dadd`, `rgn_reduce_dmin`, and `rgn_reduce_dmax` provide global reduction functionality. In general, the global reduction functions operate as follows: Each processor calls a reduction function passing an argument value; no calls return until the reduction function has been called on all processors. Upon return, the reduction function provides an "accumulation" of the argument values supplied by each processor according to some associative, binary operator. The reduction functions currently provided by CRL allow users to compute global sums, minima, and maxima of double-precision floating-point values.

Extending the set of global synchronization primitives to make it more complete (*e.g.,* reductions for other data types) would be straightforward.

3.4 Memory/Coherence Model

The simplest explanation of the coherence model provided by CRL considers entire operations on regions as indivisible units. From this perspective, CRL provides sequential consistency for read and write operations in the same sense that a sequentially consistent hardware-based DSM does for individual loads and stores.

In terms of individual loads and stores, CRL provides a memory/coherence model similar to entry [5] or release consistency [25]. Loads and stores to global data are allowed only within properly synchronized sections (operations), and modifications to a region are only made visible to other processors after the appropriate release operation (a call to `rgn_end_write`). The principal difference between typical implementations of these models and CRL, however, is that synchronization objects (and any association of data with particular synchronization objects that might be necessary) are not provided explicitly by the programmer. Instead, they are implicit in the semantics of the CRL interface: every region has an associated synchronization object (what amounts to a reader-writer lock) which is "acquired" and "released" using calls to `rgn_start_`*op* and `rgn_end_`*op*.

3.5 Discussion

CRL shares many of the advantages and disadvantages of other software DSM systems when compared to hardware DSMs. In particular, the latencies of many communication operations may be significantly higher than similar operations in a hardware-based system. Four properties of CRL allow it to offset some of this disadvantage. First, CRL is able to use part of main memory as a large secondary cache instead of relying only on hardware caches, which are typically small because of the cost of the resources required to implement them. Second, if regions are chosen to correspond to user-defined data structures, coherence actions transfer exactly the data required by the application. Third, CRL can exploit efficient bulk data transport mechanisms when transferring large regions. Finally, because CRL is implemented entirely in software at user level, it is easily modified or extended (*e.g.,* for instrumentation purposes, in order to experiment with different coherence protocols, *etc.*).

The programming model provided by CRL is not exactly the same as any "standard" shared memory programming model (*i.e.,* that provided by a sequentially-consistent all-hardware DSM system). The principal differences in the CRL programming model are twofold:

- CRL requires programmers to explicitly manage translations between the shared address space (region identifiers) and the local address space in order to allow access using standard language mechanisms.
- CRL requires programmers to insert annotations (calls to `rgn_start_`*op* and `rgn_end_`*op*) delimiting accesses to shared data.

Annotations of the second sort (delimiting accesses to shared data) are similar to those necessary in aggressive hardware and software DSM implementations (*e.g.,* those providing release consistency [25]) when writing to shared data. CRL requires such annotations whether reading or writing to shared data, similar to entry consistency [5]. As discussed in Section 6.2, experience with the applications described in this thesis indicates that the additional programming overhead of providing these annotations is quite modest. Furthermore, with this modest change to the programming model, CRL implementations are able to amortize the cost of providing the mechanisms described in Section 2.2 entirely in software over entire operations (typically multiple loads and stores) instead of paying that cost for every reference to potentially shared data.

Annotations of the first sort (related to managing translations from region identifiers to local addresses) are necessary because CRL maintains separate local and global address spaces; these represent a more significant deviation from standard shared-memory models. These annotations could be eliminated entirely (perhaps at a slight performance penalty for some applications) by integrating their functionality into the region access functions, but doing so would not address the more fundamental issue of CRL making an explicit distinction between local and global address spaces. Addressing this issue will likely require leveraging off of virtual memory mechanisms or other efficient address translation techniques; whether this can be done without adverse impact on simplicity, portability, and efficiency remains a subject of future research.

CRL places no restrictions on the number of operations a node may have in progress at any one time or the order in which those operations must be initiated. As such, programmers are faced with the same potential opportunities for introducing deadlock as they would be when using traditional DSM synchronization mechanisms (*e.g.,* mutual-exclusion or reader-writer locks) in an unstructured manner. It is possible that deadlock problems could be addressed with some combination of (1) compile-time analysis and (2) run-time support (*e.g.,* a "debugging" version of the CRL library) that is able to dynamically detect deadlock when it occurs, but neither of these approaches are employed in the current CRL implementation.

Finally, it is worth noting that CRL's integration of data access and synchronization into a single mechanism is not unlike that provided by monitors, a linguistic mechanism suggested by Hoare [28] and Brinch Hansen [8], or other linguistic mechanisms that integrate synchronization and data access (*e.g.,* mutexes in Argus [53], mutex operations in COOL [15], *etc.*).

Chapter 4

CRL Internals

This chapter describes the general structure of the prototype CRL implementation used in this thesis. Platform-specific implementation details are discussed in Chapter 5.

4.1 Overview

The prototype CRL implementation supports single-threaded applications in which a single user thread or process runs on each processor in the system. Interprocessor synchronization can be effected through region operations, barriers, broadcasts, and reductions. Many shared memory applications (*e.g.,* the SPLASH application suites [74, 84]) are written in this style. Although an experimental version of CRL that supports multiple user threads per processor and migration of threads between processors is operational [31], all results reported in this thesis were obtained using the single-threaded version.

CRL is implemented as a library against which user programs are linked; it is written entirely in C. Both CM-5 and Alewife versions can be compiled from a single set of sources with conditionally compiled sections to handle machine-specific details (*e.g.,* different message-passing interfaces). In both the CM-5 and Alewife versions, all communication is effected using active messages [81]. Message delivery is assumed to be reliable but in-order delivery is not required.

4.2 Region Identifiers

The prototype CRL implementation represents region identifiers using 32-bit unsigned integers. Each region identifier encodes a 24-bit sequence number and an eight-bit home node number using a simple, fixed encoding scheme. The home node number indicates which processor is responsible for coordinating coherence actions for a region. In the current implementation, the home node for a region is the node it was created on. The sequence numbers are unique on each home node (*i.e.,* distinct regions with the same home node have distinct sequence numbers); sequence numbers are assigned (in order) at region creation time. Because the prototype CRL implementation never reuses sequence numbers, the size of the sequence number field imposes a limit on the number of regions that can be created on a single node: the use of 24-bit sequence numbers means a node can create over 16 million regions before exhausting the local sequence number space. Similarly, the use of an eight-bit home node number limits the maximum number of processors to 256.

With a fixed encoding scheme, the maximum number of processors can only be increased at the cost of reducing the size of the sequence number space by the same factor, and vice versa. These problems could be addressed (to some extent) by either (1) allowing sequence numbers that are no longer in use to be reused (this would require some means of determining or remembering which region identifiers are no longer in use) or (2) using a flexible encoding scheme (in which segments of region identifier space are assigned to processors dynamically, on demand). However, if the real problem is that 32 bits of region identifier space is too small, it may make more sense to simply use larger region identifiers (*e.g.,* 64 bits).

4.3 Metadata

CRL allocates a fixed-size metadata area at the front of each region (or copy of a region) to hold various coherence and implementation information. Because no effort has been made to optimize for space, each region's metadata area is relatively large (104 bytes). Only a very small amount of the metadata associated with a copy of a region is ever included in protocol messages regarding the region (four or eight bytes; see Figure A-17), thus the only significant impact of relatively large metadata areas is in terms of per-region memory overhead. Tables 4-1, 4-2, and 4-3 provide a breakdown of the metadata area into individual components.

4.3.1 Common Components

Table 4-1 shows those elements that appear in the metadata area for both home and remote (non-home) copies of a region. Elements marked with a '*' in the 'CM-5 only?' column are only necessary in the CM-5 implementation of CRL.

The first element (region state) contains a pointer to the `State` data structure that indicates the current coherence protocol state for the region (see Section 4.6). The second element (region identifier) contains the region identifier for the region, as discussed in Section 3.2. The third element (version number) is used to handle problems caused by out-of-order message delivery (see Section A.11). The fourth element (region size) indicates the size of the associated region, in bytes.

Offset (bytes)	CM-5 Only?	Description	Data Type
0		Region state	Pointer to `State`
4		Region identifier	32-bit unsigned integer
8		Version number	32-bit unsigned integer
12		Region size (bytes)	32-bit unsigned integer
16	*	Recv in progress flag	32-bit unsigned integer
20	*	Recv type	32-bit unsigned integer
24	*	Recv src	32-bit unsigned integer
28	*	Recv vers	32-bit unsigned integer
32		Transaction done flag	32-bit unsigned integer
36		Continuation func	Pointer to function
40		Continuation arg1	32-bit unsigned integer
44		Continuation arg2	32-bit unsigned integer
48		Continuation arg3	32-bit unsigned integer
52		Read count	32-bit unsigned integer
56		Map count	32-bit unsigned integer
60	*	Send count	32-bit unsigned integer
64		Rtable link	Pointer to `Region`
68	*	Queue link	Pointer to `Region`

Table 4-1. Metadata elements common to both home and remote copies of regions.

The next four elements (recv in progress flag, recv type, recv src, and recv vers) are only needed on the CM-5. These fields are used to buffer the scalar components (message type, source node, and version number) of data-carrying protocol messages (see Section A.6) until the entire message has arrived; this is necessary because the bulk data transfer mechanisms used on the CM-5 deliver the scalar and bulk data components of data-carrying messages at different times.

The first of the next five fields (transaction done flag) is used to indicate the completion of a two-phase set of protocol actions. The remaining four fields (continuation func, continuation arg1, continuation arg2, and continuation arg3) are used to store the "continuation" (function pointer and arguments) that implements the second phase of a two-phase set of protocol actions. Further details can be found in Section A.4.

The next three fields (read count, map count, and send count) count the number of read operations in progress locally, the number of times a region has been mapped locally, and the number of protocol messages containing a copy of the corresponding region's data are either currently in progress or pending. The send count field is only used on the CM-5 (to determine if all pending sends for a region have completed), where it is kept up to date by incrementing it before initiating a send and having a handler initiated after a send is completed decrement it. Similar (but slightly more conservative) functionality is achieved in the Alewife CRL implementation by directly querying the outgoing bulk data transfer engine if all pending messages have been sent.

The first of the final two fields (rtable link) is used form singly-linked lists of regions in each bucked of the region table (see Section 4.4). The remaining field (queue link)

Offset (bytes)	Field	Type
72	Number of pointers	32-bit unsigned integer
76	1st ptr: Node number	32-bit unsigned integer
80	1st ptr: Metadata address	Pointer to `Region`
84	1st ptr: Version number	32-bit unsigned integer
88	1st ptr: Next pointer	Pointer to `Pointer`
92	Blocked msg queue: head	Pointer to `ProtMsg`
96	Blocked msg queue: tail	Pointer to `ProtMsg`
100	(padding)	

Table 4-2. Metadata elements specific to home copies of regions.

is only necessary on the CM-5; it is used to construct a FIFO queue of regions that are waiting to receive bulk data transfer messages (the bulk data transfer mechanism provided by the CM-5 only allows each node to be in the process of receiving a limited number of bulk data transfer messages at any given time).

Clearly, even a modest focus on reducing memory overhead could eliminate many of these fields without any significant performance penalty (*e.g.,* by replacing groups of related fields with pointers to dynamically-allocated data structures that are only instantiated when necessary).

4.3.2 Home-side Components

Table 4-2 shows those metadata elements that are specific to home copies of regions. The first five elements (number of pointers, node number, metadata address, version number, and next pointer) implement the *directory* maintained by the home copy of each region; the directory maintains information about all copies of a region that are cached on other nodes. The first of these fields (number of pointers) indicates the number of directory entries; the other four fields (node number, metadata address, version number, and next pointer) provide storage for the first directory entry; subsequent directory entries are maintained in a singly-linked list using the ‘next pointer’ fields.

The next two elements (head and tail) are used to implement a FIFO queue of “blocked” protocol messages, which is discussed further in Section A.9.

The final element (four bytes of padding) is included to ensure that the total size of the metadata area is double-word aligned, thus ensuring that the start of user data areas are double-word aligned (user data areas are allocated immediately following metadata areas; see Section 4.3.4).

Once again, some amount of metadata memory overhead could easily be eliminated (*e.g.,* by not allocating space for the first directory element in the metadata area) if desired.

Offset (bytes)	Field	Type
72	Home node number	32-bit unsigned integer
76	Metadata address	Pointer to `Region`
80	Rcvd invalidate flag	32-bit unsigned integer
84	(padding)	
88	(padding)	
92	(padding)	
96	(padding)	
100	(padding)	

Table 4-3. Metadata elements specific to remote copies of regions.

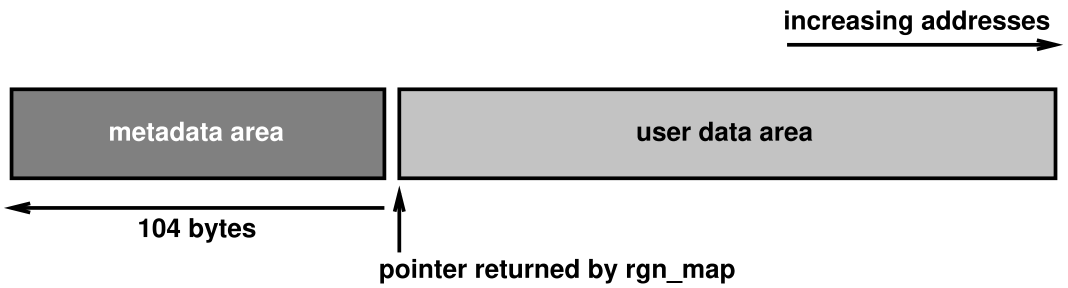


Figure 4-1. Region data layout.

4.3.3 Remote-side Components

Table 4-3 shows those metadata elements that are specific to remote copies of regions. The first element (home node number) indicates the home node number for a region. The second element (metadata address) records the address of the region metadata on the home node (see Section A.8). The third element (rcvd invalidate flag) is used to "queue" invalidate messages that cannot be processed immediately (see Section A.9).

The remaining five elements (20 bytes of padding) are included to pad out the size of the metadata for remote nodes to match that for home nodes. Strictly speaking, this is not necessary; it is done in the prototype CRL implementation for the sake of simplicity and uniformity.

4.3.4 Data Layout

The metadata and user data areas for each copy of a region (whether home or remote) are allocated adjacent to one another. The user data area starts at the end of the metadata area; the base of the user data area is the "handle" returned by `rgn_map` when a region is mapped (see Figure 4-1). The use of such a simple data layout ensures that given the address of a region's user data area, the corresponding metadata can be located quickly using a single constant-offset address calculation (which can often by combined with any offset calculations necessary for accessing particular elements of the metadata).

4.4 Mapping and Unmapping

The information necessary to implement the mapping and unmapping of regions is maintained in a *region table* kept on each node. Each region table is a hash table containing some number of buckets. Regions are hashed into buckets by applying a simple hash function to their region identifiers; buckets containing multiple regions do so using a singly-linked-list organization.

At any given time, a node's region table contains all regions that are mapped or cached on that node. In addition, because CRL currently employs a fixed-home coherence protocol (see Section 4.6), a node's region table also contains all regions that were created on that node.

Given the region table data structure, implementing `rgn_map` is straightforward. `rgn_map` examines the local region table to see if a copy of the region in question is already present. If the desired region is found, minor bookkeeping actions are performed (*e.g.,* incrementing the count of how many times the region is mapped on the local node), and a pointer to the region's user data is returned. If the desired region is not found, the home node is queried to determine the size of the region (in addition to a small amount of other auxiliary information), an appropriately-sized area of memory is allocated, the metadata area initialized, and a pointer to the user data area of the newly allocated region is returned.

The implementation of `rgn_unmap` is even simpler: aside from minor bookkeeping (*e.g.,* decrementing the mapping count), `rgn_unmap` does nothing beyond calling into the code that manages caching of unmapped regions, if necessary (see Section 4.5).

The prototype CRL implementation uses fixed-size region tables with 8,192 buckets. Since each bucket is represented with a single (four-byte) pointer to a region (the linked-list organization is maintained using a field in each region's metadata area), region tables require 32 kilobytes of memory per node.

4.5 Caching

CRL caches both mappings and data aggressively in an attempt to avoid communication whenever possible. This section describes how caching is implemented.

Whenever a region is unmapped and no other mappings of the region are in progress locally, it is inserted into a software table called the *unmapped region cache* (URC); the

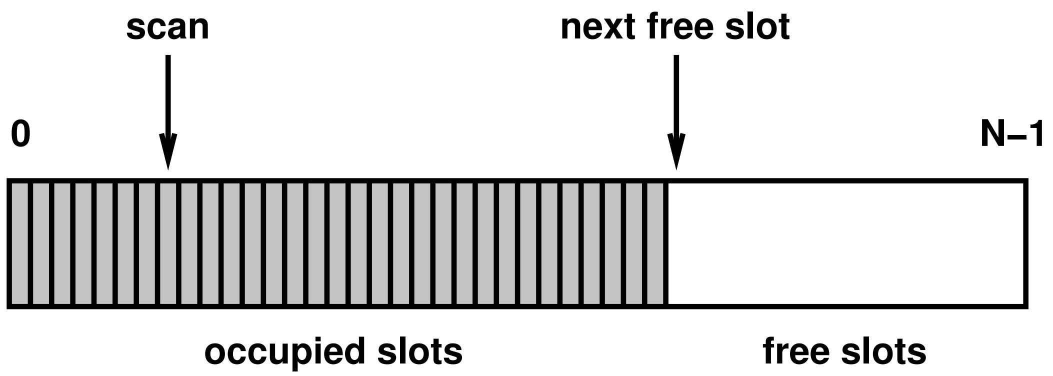


Figure 4-2. Implementation of the unmapped region cache (URC).

state of the region's data (*e.g.,* invalid, clean, dirty) is left unchanged. Inserting a region into the URC may require evicting an existing entry. This is accomplished in three steps. First, the region to be evicted (chosen using a simple round-robin scheme) is flushed (by calling `rgn_flush`). Flushing the region to be evicted ensures that if it has a valid copy of that's region data, the home node is informed that the local copy of the data has been dropped and, if necessary, causes any changes to that data to be written back. Second, the region is removed from the region table. Third, any memory resources that had been allocated for the evicted region are freed.

Unmapped regions cached in the URC are not removed from the region table, so attempts to map such regions can be satisfied efficiently without complicating the implementation of `rgn_map` described above. However, since the URC is only used to hold unmapped regions, calls to `rgn_map` that are satisfied from the URC also cause the region in question to be removed from the URC.

The URC serves two purposes. First, it allows the caching of data between subsequent `rgn_map`/`rgn_unmap` pairs on the same region. If a region with a valid copy of the associated data is placed in the URC and the data is not invalidated before the next time the region is mapped, it may be possible to satisfy subsequent calls to `rgn_start_`*op* locally, without requiring communication with the home node. Second, it enables the caching of mappings. Even if the data associated with a region is invalidated while the region sits in the URC (or perhaps was already invalid when the region was inserted into the URC), caching the mapping allows later attempts to map the same region to be satisfied more quickly than they might be otherwise. Calls to `rgn_map` that cannot be satisfied locally require sending a MsgRgnInfoReq message to the region's home node requesting information (*e.g.,* the size and current version number), waiting for the reply (a MsgRgnInfoAck message), allocating a local copy for the region, and initializing the protocol metadata appropriately.

Each node has its own URC. As shown in Figure 4-2, each URC is implemented as a simple linear table of (four-byte) pointers to regions plus two words of bookkeeping data (*scan*, which is used to implement the round-robin replacement strategy, and *next free* *slot*, which points to the next available URC slot). CRL currently employs a fixed-size URC with 1024 entries (4 kilobytes of memory per node).

Pointers to regions residing in the URC are kept in a contiguous section of the URC starting at index 0. When a region is inserted into the URC, the index at which it was inserted is remembered (in the "Continuation arg3" metadata slot, which is otherwise never in use for regions residing in the URC); doing so ensures that a region can be located and deleted from the URC with a minimum of overhead when necessary (*e.g.,* when a call to `rgn_map` is satisfied from the URC). Because any region can reside in any slot of the URC, it is effectively fully-associative.

Finally, in addition to caching mappings and data in the URC, CRL caches the data contained in mapped regions. When an application keeps a region mapped on a particular processor through a sequence of operations, CRL caches the data associated with the region between operations. Naturally, the local copy might be invalidated because of other processors initiating operations on the same region. As in hardware DSM systems, whether or not such invalidation actually happens is effectively invisible to the end user (except in terms of any performance penalty it may cause).

Cached region data, whether associated with mapped or unmapped regions, is kept in the user data area. Coherence of cached data is maintained using the protocol described in the following section. Because region data is cached in this manner, CRL gains the usual benefits associated with caching in main memory. The high density and low cost of main memory technology (DRAM) allow effective cache sizes to be quite large, particularly in contrast to SRAM-based caches that are typically found in hardware DSM systems.

4.6 Coherence Protocol

This section provides an overview of the protocol used by CRL to maintain the coherence of cached data required by the memory/coherence model described in Section 3.4. A detailed description of the coherence protocol can be found in Appendix A.

The prototype CRL implementation employs a fixed-home, directory-based invalidate protocol similar to that used in many hardware (*e.g.,* Alewife [1], DASH [49]) and software (*e.g.,* Ivy [52], Munin [11]) DSM systems. In this protocol, coherence actions for a each region are coordinated by a particular *home node* (in the current CRL implementation, the home node is always the node where the region was created).

Roughly speaking, any copy of a region, whether on the home node for that region or some other remote node, can be in one of three states: EXCLUSIVE, SHARED, or INVALID. For the copy of a region residing on its home node, these states are interpreted as follows:

EXCLUSIVE: This node (the home node) has the only valid copy of the region data.

SHARED: Both this node (the home node) and some number of other (remote) nodes have valid copies of the region data.

INVALID: Some other (remote) node has the only valid copy of the region data (thus this node does not have a valid copy).

For a copy of a region residing on a remote node, the interpretations of EXCLUSIVE and SHARED remain the same, but the meaning of INVALID changes slightly:

INVALID: This (remote) node does not have a valid copy of the region data (the state of the home node indicates where valid copies can be found).

In order to initiate a read operation on a region, the local copy of the region must be in either the EXCLUSIVE or SHARED state. If the local copy is in the INVALID state, a message is sent to the home node requesting a shared copy. Upon receiving such a request, if the copy of the region on the home node is in either the EXCLUSIVE or SHARED state, a reply containing a (shared) copy of the region data can immediately be sent to the requesting node (leaving the home node in the SHARED state).

If the copy of the region on the home node is in the INVALID state when a request for a shared copy arrives, the directory in the home region's metadata (see Section 4.3.2) is used to identify the remote node that is holding an exclusive copy of the data, and an invalidate message for the region is sent to that node. Upon receiving the invalidate message, the remote node changes the state for the local copy of the region in question to INVALID and returns an invalidate acknowledgement to the home node. When the acknowledgement for the invalidate message is received to the home node (possibly including a new copy of the region data if it had been modified on the remote node), a reply containing a (shared) copy of the region data can be sent to the original requesting node (again leaving the home node in the SHARED state).

In order to initiate a write operation on a region, the local copy of the region must be in the EXCLUSIVE state. If the local copy is in either the SHARED or INVALID state, a message is sent to the home node requesting an exclusive copy. Once again, in a manner similar to that described above for read operations, the directory information maintained on the home node is used to invalidate any outstanding copies of the region in question (if necessary), then a reply containing a (exclusive) copy of the region data is sent back to the requesting node (leaving the home node in the INVALID state).

4.6.1 Three-Party Optimization

In the coherence protocol used by the prototype CRL implementation, responses to invalidate messages are always sent back to a region's home node, which is responsible for collecting them and responding appropriately after all invalidate messages have been acknowledged. Using such a scheme, the critical path of any protocol actions involving three (or more) nodes involves four messages (request, invalidate, acknowledgement, reply). In many situations, a more aggressive (and somewhat more complicated) coherence protocol could reduce the critical path in such cases to three messages (request, invalidate, acknowledge) by having invalidate acknowledgements sent directly to the original requesting node (as is done in DASH). Eliminating long latency events (*i.e.,* messages) from the critical path of three-party protocol actions in this manner would likely yield at least a small performance improvement for many applications.

4.7 Global Synchronization Primitives

On Alewife, the global synchronization primitives described in Section 3.3 are implemented entirely in software using message-passing. Broadcasts are implemented using a binary broadcast tree rooted at each node. Barriers and reductions are implemented in a *scan* [6] style: For an n processor system, messages are sent between nodes according to a butterfly network pattern requiring $\log_2 n$ stages of n messages each.

On the CM-5, the baseline CRL implementation takes advantage of the CM-5's hardware support for global synchronization and communication (the *control network*) to implement the global synchronization primitives. The performance of the baseline CM-5 CRL implementation has been compared with that of a modified implementation in which global synchronization primitives are implemented in software (using essentially the same implementation techniques that are used in the Alewife implementation). The results of this comparison (shown in Section B.2) indicate that for the applications discussed in this thesis, use of a software-based implementation typically changes running time by no more than a few percent (sometimes a slight increase, more often a slight decrease).

4.8 Status

The prototype CRL implementation has been operational since early 1995. It has been used to run a handful of shared-memory-style applications, including two from the SPLASH-2 suite [84], on a 32-node Alewife system and CM-5 systems with up to 128 processors. A "null" implementation that provides null or identity macros for all CRL functions except `rgn_create` (which is a simple wrapper around malloc) is also available to obtain sequential timings on Alewife, the CM-5, or uniprocessor systems (*e.g.,* desktop workstations).

The `rgn_delete` function shown in Table 3-1 is a no-op in our current CRL implementation. We plan to implement the `rgn_delete` functionality eventually; the implementation should be straightforward, but there has not been any pressing need to do so for the applications we have implemented to date.

A "CRL 1.0" distribution containing user documentation, the current CRL implementation, and CRL versions of several applications are available on the World Wide Web [33].

In addition to the platforms employed in this thesis (CM-5 and Alewife), the CRL 1.0 distribution can be compiled for use with PVM [24] on a network of Sun workstations communicating with one another using TCP.

4.9 Summary

The preceding sections described the general structure of a prototype CRL implementation that provides the features and programming model described in Chapter 3. The prototype implementation is relatively simple, consisting of just over 9,200 lines of well-commented C code that can be compiled for use on three platforms with significantly different communication interfaces (Alewife, CM-5, and TCP/UNIX). Aside from the coherence protocol, the most important components of the prototype implementation are the *region table* and *unmapped region cache* data structures, which together provide a means of resolving region references (*i.e.,* mapping) and caching shared (region) data.

Chapter 5

Experimental Platforms

This chapter describes the experimental platforms used in the thesis research: Thinking Machines' CM-5 family of multiprocessors and the MIT Alewife machine.

5.1 CM-5

The CM-5 [48] is a commercially-available message-passing multicomputer with relatively efficient support for low-overhead, fine-grained message passing. Each CM-5 node contains a SPARC v7 processor (running at 32 MHz) and 32 Mbytes of physical memory.

The experiments described in this thesis were run on a 128-node CM-5 system running version 7.4 Final of the CMOST operating system and version 3.3 of the CMMD message-passing library. All application and (CRL) library source code was compiled using `gcc` version 2.6.3 with `-O2` optimization. All measurements were performed while the system was running in dedicated mode.

5.1.1 Interrupts vs. Polling

Application codes on the CM-5 can be run with interrupts either enabled or disabled. If interrupts are enabled, active messages arriving at a node are handled immediately by interrupting whatever computation was running on that node; the overhead of receiving active messages in this manner is relatively high. This overhead can be reduced significantly by running with interrupts disabled, in which case incoming active messages simply block until the code running on the node in question explicitly polls the network (or tries to send a message, which implicitly causes the network to be polled). Running with interrupts disabled is not a panacea for systems like CRL, however. With interrupt-driven message delivery, the programmer is not aware of when CRL protocol messages are processed by the local node. In contrast, if polling is used, the programmer needs to be aware of when protocol messages might need to be processed and ensure that the network is polled frequently enough to allow them to be serviced promptly. Placing this additional burden on programmers could have serious negative impact on the ease of use promised by the shared memory programming model.

The CRL implementation for the CM-5 works correctly whether interrupts are enabled or disabled. If it is used with interrupts disabled, users are responsible for ensuring the network is polled frequently enough, as is always the case when programming with interrupts disabled. For the communication workloads induced by the applications and problem sizes used in this thesis, however, there is little or no benefit to using polling instead of interrupt-driven message delivery. This has been verified both through adding different amounts of polling (by hand) to the most communication-intensive application (Barnes-Hut) and through the use of a simple synthetic workload that allows message reception mechanism (interrupts or polling), communication rate, and polling frequency (when polling-based message delivery is used) to be controlled independently. (Details about the synthetic workload and results obtained with it are presented in the following section.) Thus, unless stated otherwise, all CM-5 results presented in this these were obtained by running with interrupts enabled.

5.1.2 Whither Polling

As described in the previous section, active message delivery on the CM-5 can either be done in an interrupt-driven or polling-based manner. Generally speaking, the tradeoff between the two styles of message reception can be summarized as follows: Polling-based message delivery can be more efficient than interrupt-driven message delivery, but it requires that each node poll the network sufficiently frequently whenever other nodes might be sending messages to it. In a system like CRL, where the user is often unable to identify exactly when communication occurs, applications using polling-based message delivery must resort to simply polling the network at some constant, hopefully near-optimal frequency.

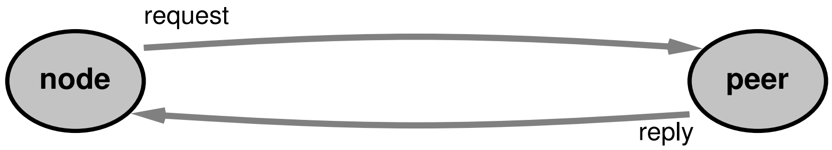
What remains unclear, however, even to veteran CM-5 programmers, is what the optimal polling frequency is, how that frequency depends on the communication workload induced by applications, and how sensitive delivered application performance is to hitting the optimal polling rate exactly. Furthermore, it is clear that at the extreme ends of the communication-rate spectrum, having applications explicitly poll the network may not be the best strategy, even when done at exactly the optimal rate. For applications that communicate frequently, the optimal strategy may be to use a polling-based message delivery model, but rely exclusively on the implicit polls associated with sending messages on the CM-5 to handle any incoming messages. If messages are being sent frequently enough, the overhead of any additional explicit polling of the network may do more harm than good. For applications that communicate very infrequently but at unpredictable times, the optimal strategy may be use interrupt-driven message delivery—the overhead of polling often enough (many times without receiving any messages) to ensure prompt service of any incoming active messages may outweigh any potential savings of receiving

```
int i;
int count;
int limit;

count = 0;
for (i=0; i<NumIters; i++)
{
  /* do "useful work" for a while
   */
  limit = count + [value];
  while (count < limit)
  {
    if ((count & [mask]) == 0)
    {
      /* if explicit polling of the network is being used, also poll
       * for incoming messages after null_proc returns
       */
      null_proc();
    }
    count += 1;
  }

  /* initiate a communication event with one or more randomly
   * selected peers and wait for it to complete
   */
  [... ... ...]
}
```

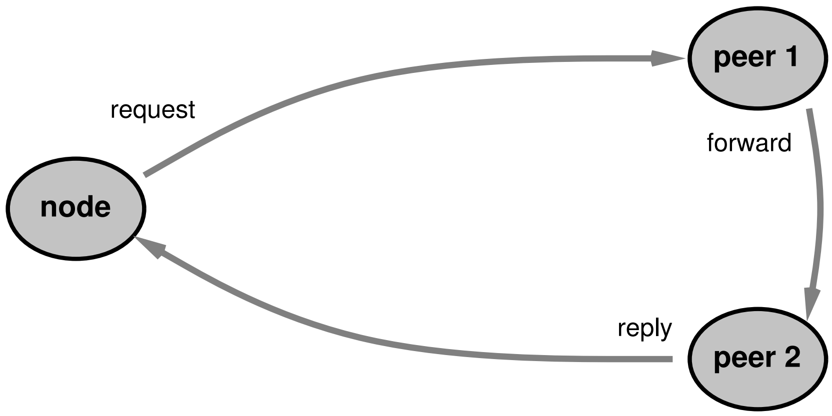
Figure 5-1. Basic structure of the synthetic workload.

messages using polling instead of interrupts. Indeed, the situation with respect to polling is murky; basic questions such as how frequently to poll or whether polling should even be used remain unanswered. The rest of this section describes and presents results form a simple synthetic workload that attempts to shed light on the subject.

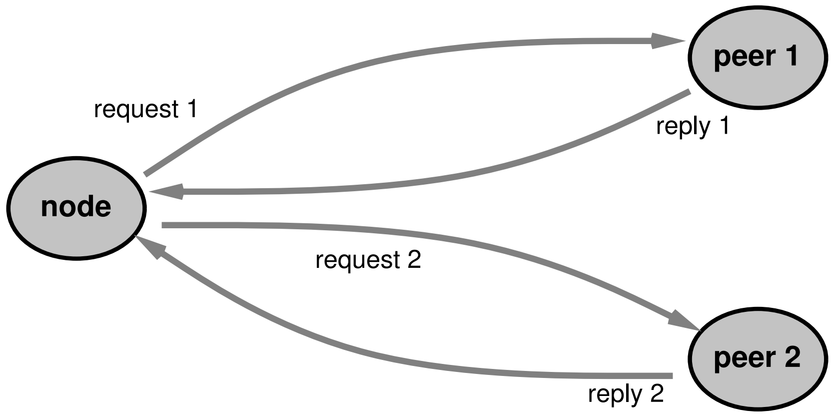
The synthetic workload is designed to cause communication patterns similar to those that might be induced by a simple DSM system. Specifically, the communication patterns caused by the synthetic workload are unpredictable, in both time (*when* communication happens) and space (*who* communication happens with). In addition, basic communication events in the workload are always of a request/reply form (*i.e.,* synchronous), sometimes involving multiple peer nodes.



(a) Two-message event: one peer, request/reply.



(b) Three-message event: two peers sequentially.



(c) Four-message event: two peers concurrently.

Figure 5-2. Communication events used in the synthetic workload.

Figure 5-1 illustrates the basic structure of the synthetic workload. All processors execute a large outer loop, each iteration of which consists of computation phase (a simple inner loop that models some amount of useful work) followed by a communication phase (one of three possible "communication events" described below).

The *computation-to-communication ratio* of the workload (*i.e.,* the amount of useful work between communication events) is determined by how *[value]* is selected on each iteration of the outer loop. For the results presented in this section, each *[value]* is a sample an exponentially-distributed random variable with a mean selected to yield the desired amount of useful work between communication events. To eliminate complex calculations from the middle of the workload loop, *[value]*s are obtained from a precomputed table (with `NumIters` entries).

A running `count` is tested an updated on each iteration of the inner loop. By using values of the form $2^n - 1$ for the *[mask]* against which `count` is tested, one can ensure that `null_proc` gets called every 2^n-th pass through the inner loop. When interrupts are disabled and the network is explicitly polled to receive messages, a call to the polling function is inserted after `null_proc` returns. Thus, the frequency with which the network is polled can be controlled by varying the value used for *[mask]*.

Figure 5-2 illustrates the three basic types of communication events used in the synthetic workload. Each event is named by the number of messages involved. Two-message events involve a pair of messages between the invoking node and a single peer in a request/reply pattern. Three-message events involve three messages between the invoking node and a pair of peers in a request/forward/reply pattern. Finally, four-message events involve four messages between the invoking node and a pair of peers where both peers are simultaneously contacted in a request/reply pattern. While far from exhaustive, these communication events are intended to be representative of the low-level communication events that occur in DSM systems.

In all cases, peers are selected at random (again, using a precomputed table). Each communication event consists of sending the initial message or messages and polling until the appropriate reply or replies are received. All messages sent between nodes are implemented with a single `CMAML_rpc` active message. When interrupt-driven message delivery is used, each communication event is preceded by a call to `CMAML_disable_interrupts` (to ensure that interrupts are disabled when `CMAML_rpc` is called, as is required for correct operation) and followed by a call to `CMAML_enable_interrupts` (to reenable interrupt-driven message delivery).[1] Only a single type of communication event is used in each workload run.

For each combination of computation-to-communication ratio, polling frequency, and communication event type, the average time per iteration of the outer workload loop shown in Figure 5-1 (measured over a large number of iterations) is measured for four variations: a 'null' (baseline) case that includes only the inner "useful work" loop (no

[1]The combination of `CMAML_rpc` and (when interrupt-driven message delivery is used) disabling/reenabling interrupts is used to match the communication behavior of CRL; see Section 5.1.3.

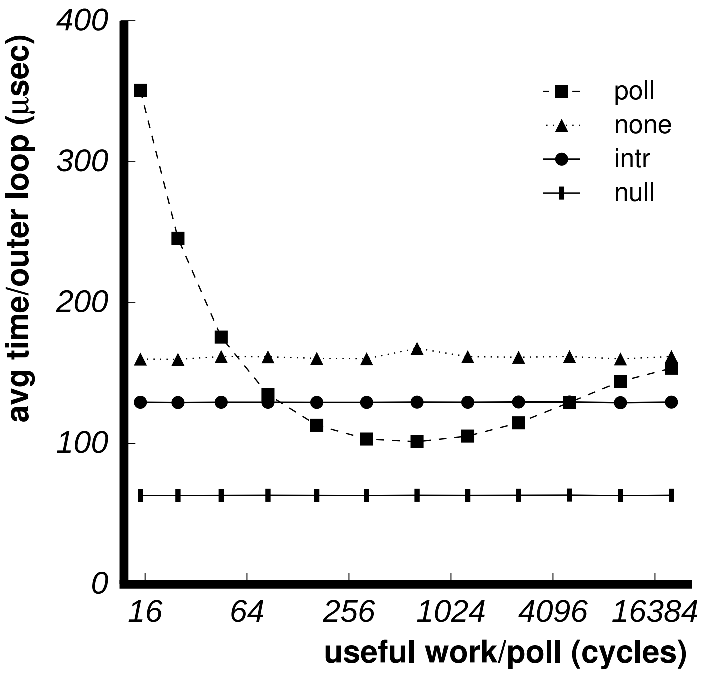


Figure 5-3. Typical synthetic workload performance data (2000 cycles of useful work per four-message communication events).

messages are sent or received); an 'intr' case in which interrupt-driven message delivery is employed; a 'none' case in which interrupts are disabled, but messages are only received via the implicit polling that occurs when messages are sent; and, finally, a 'poll' case in which interrupts are disabled and the network is explicitly polled as well. Since the first three of these cases ('null', 'intr', and 'none') do not employ explicit polling for message reception, the only effect that changing the "polling frequency" (varying the value used for *[mask]*) has on them is changing the frequency with which the inner loop test succeeds and `null_proc` is called. Thus, other than measurement variations due to experimental noise, only the behavior of the 'poll' case is affected by changing the value used for *[mask]*.

The synthetic workload was run on a 32-node CM-5 partition running in dedicated mode for a wide range of parameters (cycles of useful work per communication event ranging from 125 to 32,000; cycles of useful work per poll ranging from 15 to 20,485). Figure 5-3 shows a typical sampling of the data thus obtained. As is often the case, polling too frequently (the left side of the figure) or not frequently enough (the right side of the figure) leads to poor performance. However, for this communication workload, polling at the optimal rate (roughly 700 cycles of useful work per poll) leads to performance 22 percent better than relying on interrupt-driven message delivery and 37 percent better than relying on implicit polling (when messages are sent). Complete results for all parameter combinations can be found in Section B.1.

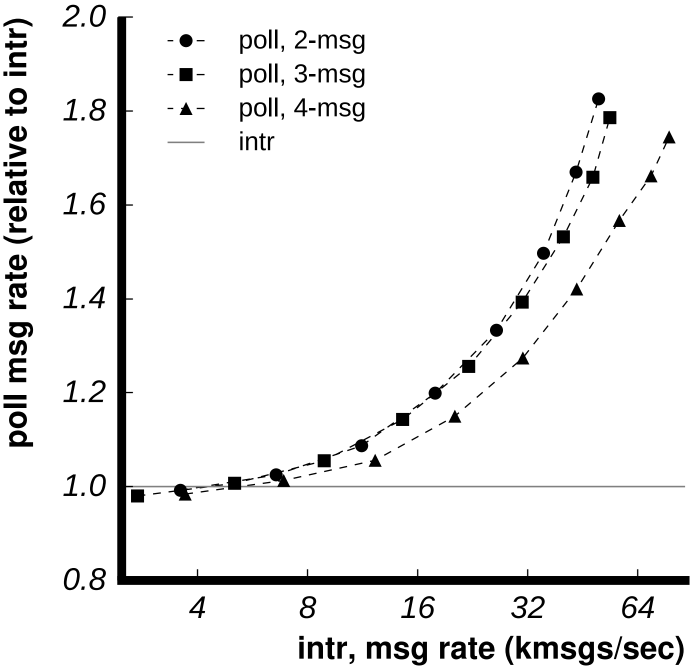


Figure 5-4. Performance of 'poll' synthetic workload case relative to 'intr' (see text for explanation).

In terms of sensitivity of performance to polling at exactly the optimal rate, the CM-5 appears to be fairly robust. Over the entire data set collected with the synthetic workload, the largest performance decrease due to polling at either half or twice the observed optimal rate was approximately five percent.

Figure 5-4 plots the performance of the 'poll' case relative to 'intr'. The horizontal axis plots average per-node message injection rate for the 'intr' case; the vertical axis plots the relative performance for the 'poll' case (assuming the optimal polling rate). Each curve represents a set of measurements for which the type of communication event is kept fixed; each symbol on a curve represents a single experiment. For example, in the data shown in Figure 5-3, using interrupt-driven message delivery yields an average per-node message injection rate of approximately 31,000 messages per second. At the optimal polling rate, using polling-based message delivery results in a message rate of roughly 39,500 messages per second; 1.27 times the performance in the interrupt-driven case. Thus, a symbol is placed on the "poll, 4-msg" curve at (31,000, 1.27).

Although somewhat complex, presenting the data in this way is particularly useful for the task at hand: for an existing application that uses interrupt-driven message delivery, deciding what the potential performance benefit of switching to a polling-based model would be. For example, for applications using interrupt-driven message delivery with per-node message injection rates of no more than approximately 5,000 messages per second, changing to a polling-based message delivery model would probably yield little or no performance improvement. As can be seen from the data presented in Section 6.2.4, the applications used in this thesis easily satisfy this criterion, so it is not surprising that polling-based versions of the applications performed no better (or worse) than the baseline interrupt-driven versions. For more communication-intensive workloads, however, it is clear from this data that use of polling-based message delivery can lead to significant performance improvements (up to 80 percent).

5.1.3 Communication Performance

In a simple ping-pong test, the round-trip time for four-argument active messages (the size CRL uses for non-data carrying protocol messages) on the CM-5 is approximately 34 microseconds (1088 cycles). This includes the cost of disabling interrupts on the requesting side[2], sending the request, polling until the reply message is received, and then reenabling interrupts. On the replying side, message delivery is interrupt-driven, and the handler for the incoming request message does nothing beyond immediately sending a reply back to the requesting node.

For large regions, data-carrying protocol messages use the CMMD's `scopy` functionality to effect data transfer between nodes. `scopy` achieves a transfer rate of 7 to 8 Mbytes/second for large transfers, but because it requires prenegotiation of a special data structure on the receiving node before data transfer can be initiated, performance on small transfers can suffer. To address this problem, CRL employs a special mechanism for data transfers smaller than 256 bytes (the crossover point between the two mechanisms). This mechanism packs three payload words and a destination base address into each four-argument active message; specialized message handlers are used to encode offsets from the destination base address at which the payload words should be stored in the message handler. While this approach cuts the effective transfer bandwidth roughly in half, it provides significantly reduced latencies for small transfers by avoiding the need for prenegotiation with the receiving node.

Networks of workstations with interprocessor communication performance rivaling that of the CM-5 are rapidly becoming reality [7, 56, 77, 80]. For example, Thekkath *et al.* [78] describe the implementation of a specialized data-transfer mechanism implemented on a pair of 25 MHz DECstations connected with a first-generation FORE ATM network. They report round-trip times of 45 microseconds (1125 cycles) to read 40 bytes of data from a remote processor and bulk data transfer bandwidths of roughly 4.4 Mbytes/second. Since these parameters are relatively close to those for the CM-5, we expect that the performance of CRL on the CM-5 is indicative of what should be possible for implementations targeting networks of workstations using current- or next-generation technology.

[2]Disabling interrupts is required when using `CMAML_rpc` to send an active message; `CMAML_rpc` must be used because CRL's coherence protocol does not fit into the simple request/reply network model that is supported somewhat more efficiently on the CM-5.

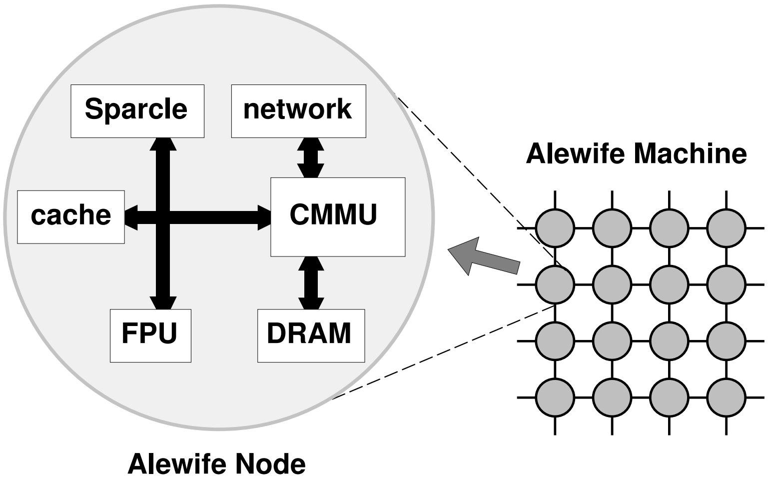


Figure 5-5. Basic Alewife architecture.

5.2 Alewife

Alewife is an experimental distributed memory multiprocessor. The basic Alewife architecture consists of processor/memory nodes communicating over a packet-switched interconnection network organized as a two-dimensional mesh (see Figure 5-5). Each processor/memory node consists of a Sparcle processor [2], an off-the-shelf floating-point unit (FPU), a 64-kilobyte unified instruction/data cache (direct mapped, 16-byte lines), eight megabytes of DRAM, the local portion of the interconnection network (a Caltech "Elko" series Mesh Routing Chip [71]), and a Communications and Memory Management Unit (CMMU). Because Sparcle was derived from a SPARC v7 processor not unlike that used in the CM-5 nodes, basic processor issues (instruction set, timings, etc.) are quite similar on the two machines.

The experiments described in this thesis were run on a 32-node Alewife machine running a locally-developed minimal, single-user operating system. All application and library source code was compiled using the Alewife C compiler, which uses a modified version of ORBIT [44] for code generation. The Alewife C compiler delivers approximately 90 percent of the performance of `gcc -O2` for integer code. Since the Alewife C compiler does not attempt to schedule floating-point operations optimally, code quality is somewhat worse with floating-point code.

5.2.1 Integrated Support for Shared Memory and Message Passing

Alewife provides efficient support for both coherent shared-memory and message-passing communication styles. Shared memory support is provided through an implementation of the LimitLESS cache coherence scheme [13]: limited sharing of memory blocks (up to five remote readers) is supported in hardware; higher-degree sharing is handled by trapping the processor on the home memory node and extending the small hardware directory in software. This organization is motivated by studies indicating that small-scale sharing of data is the common case [12, 61, 83]; data shared more widely is relatively uncommon. In general, Alewife's shared memory system performs quite well, enabling speedups comparable to or better than other scalable hardware-based DSM systems [1, 49].

In addition to providing support for coherent shared memory, Alewife provides the processor with direct access to the interconnection network for sending and receiving messages [45]. Efficient mechanisms are provided for sending and receiving both short (register-to-register) messages and long (memory-to-memory, bulk data transfer) messages. In addition, messages combining both types of data can be sent: some elements of a message can be register-to-register, scalar values, while the rest of the message consists of bulk data that is transferred directly out of the memory of the sending node into the network and, upon message reception, directly from the network into the memory of the receiving node. Using Alewife's message-passing mechanisms, a processor can send a message with just a few user-level instructions. A processor receiving such a message will trap and respond either by rapidly executing a message handler or by queuing the message for later consideration when an appropriate message handler gets scheduled. Scheduling and queuing decisions are made entirely in software.

Two non-fatal bugs in the first-run CMMU silicon warrant mention here. First, because of a timing conflict between the CMMU and the FPU, codes that make significant use of the FPU are limited to running at 20 MHz instead of the target clock rate of 33 MHz. Because of this, all Alewife performance results presented in this thesis assume a 20 MHz clock. Second, in order to ensure data integrity when using the bulk data transfer mechanism, it is necessary to flush message buffers from the memory system before sending or initiating storeback on the receiving processor. This overhead cuts the effective peak bandwidth of the bulk data transfer mechanism from approximately 2.2 bytes/cycle (44 Mbytes/second) to roughly 0.9 bytes/cycle (18 Mbytes/second). Aside from the clock speed limitation, neither bug has any impact on the performance of Alewife's hardware-supported shared memory mechanisms. Both bugs will be fixed in second-run parts resulting from a CMMU respin effort.

5.2.2 Communication Performance

For the same simple ping-pong test used on the CM-5, the round-trip time for four-word active messages on Alewife (using interrupt-driven message delivery on both ends) is approximately 14 microseconds (280 cycles). Even without correcting for the differences in clock speed, this is more than a factor of two faster than the CM-5. In the Alewife CRL implementation, active message latencies are somewhat higher, however, because all protocol message handlers are effectively transitioned into full-fledged threads that can be interrupted by incoming messages. This transition prevents long-running handlers from blocking further message delivery and causing network congestion. Currently, this transition adds approximately 12.4 microseconds (248 cycles) to the round-trip time, but minor functionality extensions planned for the CMMU respin will make it possible to reduce this overhead by at least an order of magnitude.

In the Alewife CRL implementation, data-carrying protocol messages are implemented using messages that consist of a header containing the control portion of the protocol message (passed as register-to-register, scalar values using the same layout as non-data carrying protocol messages; see Figure A-17) followed by the data portion of the protocol message (passed using Alewife's memory-to-memory bulk data transfer mechanism). As discussed in the previous section, the effective peak performance delivered by the bulk data transfer mechanism (including the overhead required to flush message buffers on both sender and receiver) is approximately 18 Mbytes/second.

5.2.3 Status

A sixteen-node Alewife machine has been operational since June, 1994; this system was expanded to 32 nodes in November, 1994. A CMMU respin effort is currently underway; once the second-run CMMU parts are available, plans call for the construction of a 128-node system.

Chapter 6

Results

This chapter returns to one of the core issues of this thesis, the question of how much hardware support is necessary to enable good DSM systems capable of delivering performance competitive with aggressive hardware-based implementations.

This question is addressed in two ways. First, to investigate the importance of providing hardware support for the three basic mechanisms described in Section 2.2, the performance of CRL is compared to that delivered by Alewife's native, hardware-supported shared memory. The applications used in this comparison are described in Section 6.2. Section 6.3 presents the performance of the Alewife version of CRL on these applications and compares it with that delivered by Alewife's native shared memory support. These results indicate that when built upon aggressive, high-performance communication mechanisms, CRL is capable of delivering performance within 15 percent of Alewife's hardware-supported shared memory, even for challenging applications (*e.g.,* Barnes-Hut) and small problem sizes.

Second, to investigate the importance of aggressive hardware support in the form of high-performance communication mechanisms (both at the processor-network interface and in the network fabric proper), Sections 6.4 and 6.5 evaluate the sensitivity of CRL performance to increased communication costs. Two approaches are used to perform this evaluation. First, by comparing the performance of the three applications described in Section 6.2 running under CRL on both Alewife and the CM-5, the Section 6.4 indicates the sensitivity of CRL to large changes in the cost of interprocessor communication. Second, by measuring the impact of inserting additional overhead (in the form of no-op instructions) into the code paths used for sending and receiving messages, Section 6.5 provides insight into the "small-scale" sensitivity to changes in communication performance (both latency and bandwidth).

Section 6.1 sets the context for the application-level results by presenting results from a simple microbenchmark that measures the latencies of various basic CRL events and compares them to those provided by Alewife's native shared memory system.

Event		CM-5 cycles	CM-5 μsec	Alewife cycles	Alewife μsec	Alewife (native) cycles	Alewife (native) μsec
Start read	hit	79	2.5	47	2.3	—	—
End read		99	3.1	51	2.6	—	—
Start read	miss, no invalidations	1925	60.2	1030	51.5	39	1.9
Start write	miss, one invalidation	3620	113.1	1760	88.0	67	3.3
Start write	miss, six invalidations	4663	145.7	3288	164.4	769	38.4

Table 6-1. Measured CRL latencies for 16-byte regions (in both cycles and microseconds). Measurements for Alewife's native shared memory system are provided for comparison.

Unless stated otherwise, all Alewife CRL results presented in this chapter include the overhead of flushing message buffers and transitioning message handlers into threads as discussed in Section 5.2.2. Because this overhead comprises 36 to 49 percent of measured Alewife CRL latencies, CRL performance on both microbenchmarks and applications should improve somewhat after the CMMU respin (as discussed further in Section 6.5).

6.1 Basic Latencies

The following simple microbenchmark is used to measure the cost of various CRL events. 64 regions are allocated on a selected home node. Situations corresponding to desired events (*e.g.,* a start write on a remote node that requires other remote read copies to be invalidated) are constructed mechanically for some subset of the regions; the time it takes for yet another processor to execute a simple loop calling the relevant CRL function for each of these regions is then measured. The time for the event in question is then computed by repeating this process for all numbers of regions between one and 64 and then computing the linear regression of the number of regions against measured times; the slope thus obtained is taken to be the time per event.

Invocations of `rgn_map` that can be satisfied locally (*e.g.,* because the call was made on the home node for the region in question, the region is already mapped, or the region is present in the URC) are termed "hits." On both Alewife and the CM-5, invocations of `rgn_map` that are hits cost between 80 and 140 cycles, depending on whether or not the region in question had to be removed from the unmapped region cache. Calls to `rgn_map` that cannot be satisfied locally ("misses") are more expensive (roughly 830 cycles on Alewife and 2,200 cycles on the CM-5). This increase reflects the cost of sending a message to the region's home node, waiting for a reply, allocating a local copy for the region, and initializing the protocol metadata appropriately. Invocations of `rgn_unmap` take between 30 and 80 cycles; the longer times correspond to cases in which the region being unmapped needs to be inserted into the unmapped region cache.

Table 6-1 shows the measured latencies for a number of typical CRL events, assuming 16-byte regions. The first two lines ("start read, hit" and "end read") represent events

Event		CM-5 cycles	CM-5 μsec	Alewife cycles	Alewife μsec
Start read	miss, no invalidations	3964	123.9	1174	58.7
Start write	miss, one invalidation	5644	176.4	1914	95.7
Start write	miss, six invalidations	6647	207.7	3419	171.0

Table 6-2. Measured CRL latencies for 256-byte regions (in both cycles and microseconds).

that can be satisfied entirely locally. The other lines in the table show miss latencies for three situations: "start read, miss, no invalidations" represents a simple read miss to a remote location requiring no other protocol actions; "start write, miss, one invalidation" represents a write miss to a remote location that also requires a read copy of the data on a third node to be invalidated; "start write, miss, six invalidations" represents a similar situation in which read copies on six other nodes must be invalidated.

Latencies for Alewife's native shared memory system are provided for comparison. The first two cases shown here (read miss, no invalidations, and write miss, one invalidation) are situations in which the miss is satisfied entirely in hardware. The third case (write miss, six invalidations) is one in which LimitLESS software must be invoked, because Alewife only provides hardware support for up to five outstanding copies of a cache line. For 16-byte regions (the same size as the cache lines used in Alewife), the CRL latencies are roughly a factor of 15 larger than those for a request handled entirely in hardware; this factor is entirely due to time spent executing CRL code and the overhead of active message delivery.

Table 6-2 shows how the miss latencies given in Table 6-1 change when the region size is increased to 256 bytes. For Alewife, these latencies are only 130 to 160 cycles larger than those for 16-byte regions; roughly three quarters of this time is due to the overhead of flushing larger message buffers (which will be unnecessary after the CMMU respin). Even so, the fact that the differences are so small testifies to the efficiency of Alewife's block transfer mechanism.

Interestingly, these latencies indicate that with regions of a few hundred bytes in size, Alewife CRL achieves a remote data access bandwidth similar to that provided by hardware-supported shared memory. With a miss latency of 1.9 microseconds for a 16-byte cache line, Alewife's native shared memory provides a remote data access bandwidth of approximately 8.4 Mbytes/second. For regions the size of cache lines, Alewife CRL lags far behind. For 256-byte regions, however, Alewife CRL delivers 4.4 Mbytes/second (256 bytes @ 58.7 microseconds); discounting the overhead of flushing message buffers and transitioning message handlers into threads increases this to 7.9 Mbytes/second (256 bytes @ 32.4 microseconds). While such a simple calculation ignores numerous important issues, it does provide a rough indication of the data granularity that CRL should be able to support efficiently when built on top of fast message-passing mechanisms. Since the CM-5 provides less efficient mechanisms for bulk data transfer, much larger regions are

	Blocked LU	Water	Barnes-Hut
Source lines	1,732	2,971	3,825
`rgn_map`	27	5	31
`rgn_unmap`	30	0	29
`rgn_start_read`	19	11	17
`rgn_end_read`	19	11	15
`rgn_start_write`	11	20	22
`rgn_end_write`	11	20	27
`rgn_flush`	0	0	0

Table 6-3. Static count of source lines and CRL calls for the three applications.

	Blocked LU	Water	Barnes-Hut
Number of regions used	2,500	500	16,000
Typical region size (bytes)	800	672	100

Table 6-4. Approximate number of regions used and typical region sizes for the three applications (assuming default problem sizes).

required under CM-5 CRL to achieve remote data access bandwidth approaching that delivered by Alewife's hardware-supported shared memory.

6.2 Applications

While comparisons of the performance of low-level mechanisms can be revealing, end-to-end performance comparisons of real applications are far more important. Three applications (Blocked LU, Water, Barnes-Hut) were used to evaluate the performance delivered by the two different versions of CRL and compare it with that provided by Alewife's native support for shared memory. All three applications were originally written for use on hardware-based DSM systems. In each case, the CRL version was obtained by porting the original shared-memory code directly—regions were created to correspond to the existing shared data structures (*e.g.,* structures, array blocks) in the applications, and the basic control flow was left unchanged. Judicious use of conditional compilation allows a single set of sources for each application to be compiled to use either CRL (on Alewife or the CM-5) or shared memory (Alewife only) to effect interprocessor communication. Table 6-3 shows total source line counts (including comments and preprocessor directives) and static counts of CRL calls for the three applications. Table 6-4 shows the approximate number of regions used by each application and the typical sizes of said regions.

The shared-memory versions of applications use the hardware-supported shared memory directly without any software overhead (calls to the CRL functions described in Section 3.2 are compiled out). For the sake of brevity, the rest of the thesis uses the term "Alewife SM" to refer to this case. None of the applications employ any prefetching.

6.2.1 Blocked LU

Blocked LU implements LU factorization of a dense matrix; the version used in this study is based on one described by Rothberg *et al.* [66]. Unless stated otherwise, the results for Blocked LU presented in this thesis were obtained with a 500x500 matrix using 10x10 blocks.

In the CRL version of the code, a region is created for each block of the matrix to be factored; thus the size of each region—the data granularity of the application—is 800 bytes (100 double-precision floating point values). Blocked LU also exhibits a fairly large computation granularity, performing an average of approximately 11,000 cycles of useful work per CRL operation. (This figure is obtained by dividing the sequential running time by the number of operations executed by the CRL version of the application running on a single processor; see Tables 6-5 and 6-6.)

6.2.2 Water

The Water application used in this study is the "n-squared" version from the SPLASH-2 application suite; it is a molecular dynamics application that evaluates forces and potentials in a system of water molecules in the liquid state. Applications like Water are typically run for tens or hundreds of iterations (time steps), so the time per iteration in the "steady state" dominates any startup effects. Therefore, running time is determined by running the application for three iterations and taking the average of the second and third iteration times (thus eliminating timing variations due to startup transients that occur during the first iteration). Unless stated otherwise, the results for Water presented in this thesis are for a problem size of 512 molecules.

In the CRL version of the code, a region is created for each molecule data structure; the size of each such region is 672 bytes. Three small regions (8, 24, and 24 bytes) are also created to hold several running sums that are updated every iteration (via a write operation) by each processor. Although the data granularity of Water is still relatively large, its computation granularity is over a factor of seven smaller than that of Blocked LU—an average of approximately 1,540 cycles per CRL operation.

6.2.3 Barnes-Hut

Barnes-Hut is also taken from the SPLASH-2 application suite; it employs hierarchical n-body techniques to simulate the evolution of a system of bodies under the influence of gravitational forces. As was the case with Water, applications like Barnes-Hut are often run for a large number of iterations, so the steady-state time per iteration is an appropriate measure of running time. Since the startup transients in Barnes-Hut persist through the first two iterations, running time is determined by running the application for four iterations and taking the average of the third and fourth iteration times. Unless stated otherwise,

	Blocked LU			Water			Barnes-Hut		
	CM-5	Alewife		CM-5	Alewife		CM-5	Alewife	
	CRL	CRL	SM	CRL	CRL	SM	CRL	CRL	SM
sequential	24.73	53.49	53.49	11.74	22.80	22.80	12.82	22.84	22.84
1 proc	25.31	54.67	53.57	13.75	24.16	22.82	24.30	34.80	22.99
2 procs	13.96	28.42	28.48	7.36	12.84	12.36	15.03	19.05	11.74
4 procs	7.74	14.83	14.69	4.01	6.93	6.69	8.20	10.02	6.16
8 procs	4.53	7.89	7.78	2.23	3.68	3.38	4.68	5.42	3.45
16 procs	2.57	4.25	4.20	1.57	2.00	1.91	2.53	2.85	2.17
32 procs	1.79	2.40	2.71	1.13	1.18	1.02	1.49	1.58	1.41

Table 6-5. Application running times (in seconds). All values are averages computed over three consecutive runs.

the results for Barnes-Hut presented in this thesis are for a problem size of 4,096 bodies (one-quarter of the suggested base problem size). Other application parameters (Δt and θ) are scaled appropriately for the smaller problem size [74].

In the CRL version of the code, a region is created for each of the octree data structure elements in the original code: bodies (108 bytes), tree cells (88 bytes), and tree leaves (100 bytes). In addition, all versions of the code were modified to use the efficient reduction primitives for computing global sums, minima, and maxima (the CRL versions of Barnes-Hut use the reduction primitives provided by CRL; the shared memory version uses similarly scalable primitives implemented using shared memory mechanisms).

Barnes-Hut represents a challenging communication workload. First, communication is relatively fine-grained, both in terms of data granularity (roughly 100 bytes) and computation granularity—approximately 436 cycles of useful work per CRL operation, a factor of roughly 3.5 and 25 smaller than Water and Blocked LU, respectively. Second, although Barnes-Hut exhibits a reasonable amount of temporal locality, access patterns are quite irregular due to large amounts of "pointer chasing" through the octree data structure around which Barnes-Hut is built. In fact, Barnes-Hut and related hierarchical n-body methods present a challenging enough communication workload that they have been used by some authors as the basis of an argument in favor of aggressive hardware support for cache-coherent shared memory [72, 73].

6.2.4 Performance

Table 6-5 summarizes the running times for the sequential, CRL, and shared memory (SM) versions of the three applications. Sequential running time is obtained by linking each application against the null CRL implementation described in Section 4.8 and running on a single node of the architecture in question; this time is used as the basepoint for computing application speedup. The running times for the CRL versions of applications running on one processor are larger than the sequential running times. This difference rep-

	Events	Blocked LU		Water		Barnes-Hut	
		CM-5	Alewife	CM-5	Alewife	CM-5	Alewife
1 proc	map count (in 1000s)	84.58	84.58	—	—	983.60	983.60
	operation count (in 1000s)	84.63	84.63	269.32	269.32	992.22	992.22
32 procs	map count (in 1000s)	2.81	2.81	—	—	30.76	30.76
	(miss rate, %)	15.3	15.3	—	—	1.1	1.1
	operation count (in 1000s)	2.78	2.78	8.68	8.68	31.34	31.22
	(miss rate, %)	14.3	14.3	7.7	9.3	4.6	4.6
	msg count (in 1000s)	1.65	1.65	2.53	3.03	5.69	5.69

Table 6-6. Application characteristics when running under CRL (see Section 6.2.4 for description). All values are averages computed over three consecutive runs.

resents the overhead of calls to CRL functions—even CRL calls that “hit” incur overhead, unlike hardware systems where hits (*e.g.,* in a hardware cache) incur no overhead.

Table 6-6 presents event counts obtained by compiling each application against an instrumented version of the CRL library and running the resulting binary. The instrumented version of the CRL library collected many more statistics than those shown here (see Section B.4); applications linked against it run approximately 10 percent slower than when linked against the unmodified library. Table 6-6 shows counts for three different events: “map count” indicates the number of times regions were mapped (because calls to `rgn_map` and `rgn_unmap` are always paired, this number also represents the number of times regions were unmapped); “operation count” indicates the total number of CRL operations executed (paired calls to `rgn_start_`*op* and `rgn_end_`*op*); and “msg count” shows the number of protocol messages sent and received. For the 32 processor results, miss rates are also shown; these rates indicate the fraction of calls to `rgn_map` and `rgn_start_`*op* that could not be satisfied locally (without requiring interprocessor communication). All counts are average figures expressed on a per-processor basis.

Map counts and miss rates for Water are shown as ‘—’ because the application’s entire data set is kept mapped on all nodes at all times; regions are mapped once at program start time and never unmapped. While this may not be a good idea in general, it is reasonable for Water because the data set is relatively small (a few hundred kilobytes) and is likely to remain manageable even for larger problem sizes.

Figure 6-1 shows the performance of the three different versions of Blocked LU (CM-5 CRL, Alewife CRL, Alewife SM) on up to 32 processors. The top plot shows absolute running time, without correcting for differences in clock speed between the CM-5 (32 MHz) and Alewife (20 MHz). The bottom plot shows speedup; the basepoints for the speedup calculations are the sequential running times shown in Table 6-5 (thus both Alewife curves are normalized to the same basepoint, but the CM-5 speedup curve uses a different basepoint). Figures 6-2 and 6-3 provide the same information for Water and Barnes-Hut, respectively.

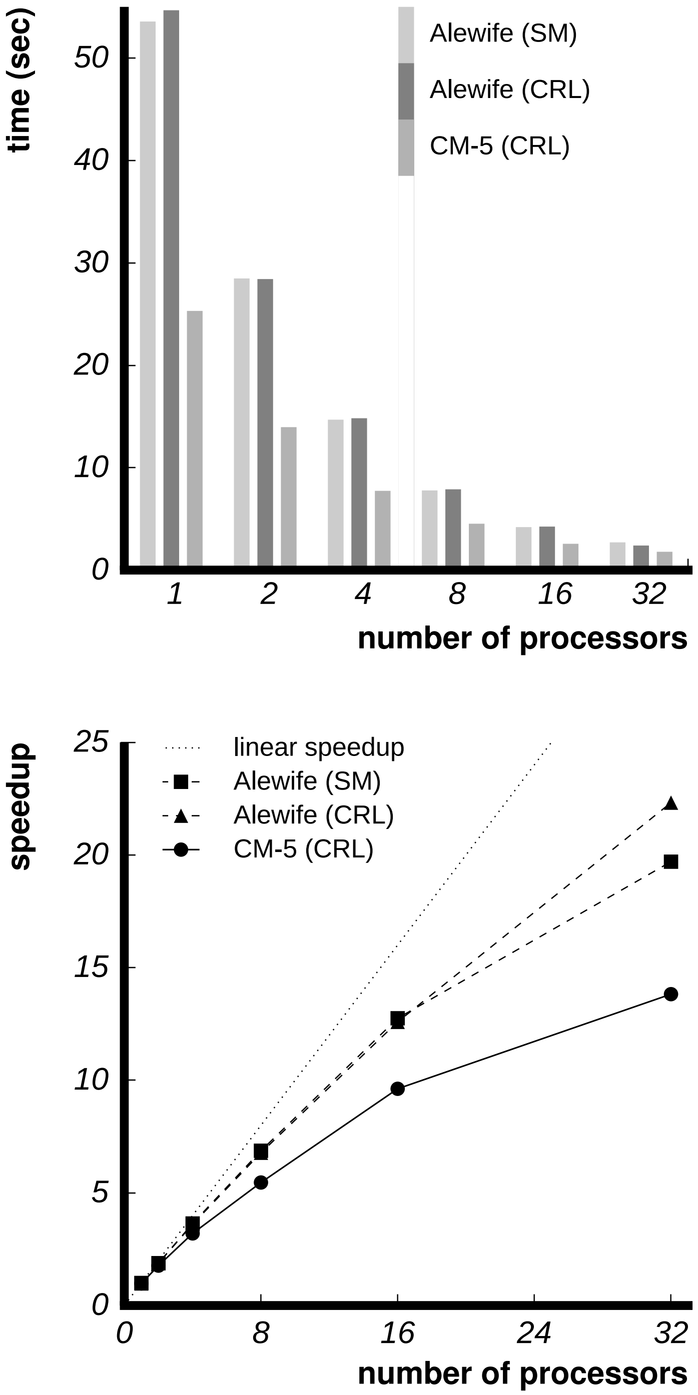


Figure 6-1. Absolute running time (top) and speedup (bottom) for Blocked LU (500x500 matrix, 10x10 blocks).

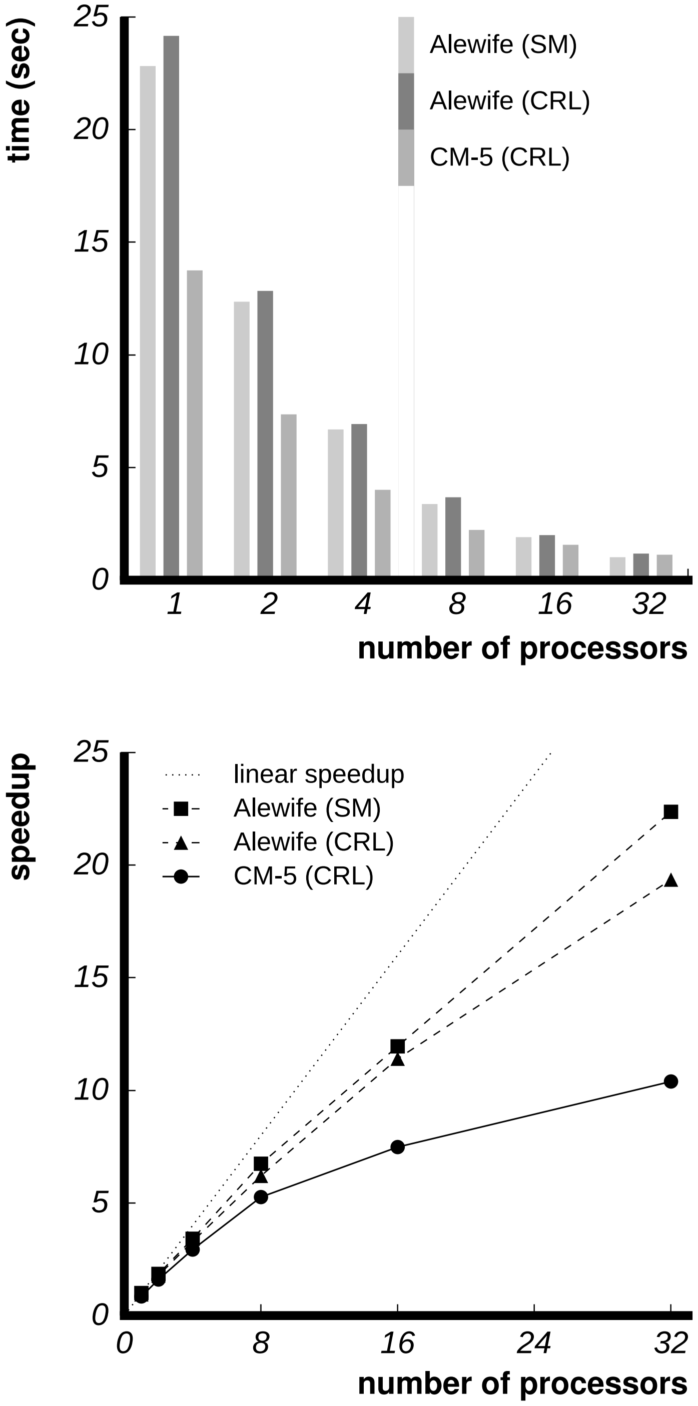


Figure 6-2. Absolute running time (top) and speedup (bottom) for Water (512 molecules).

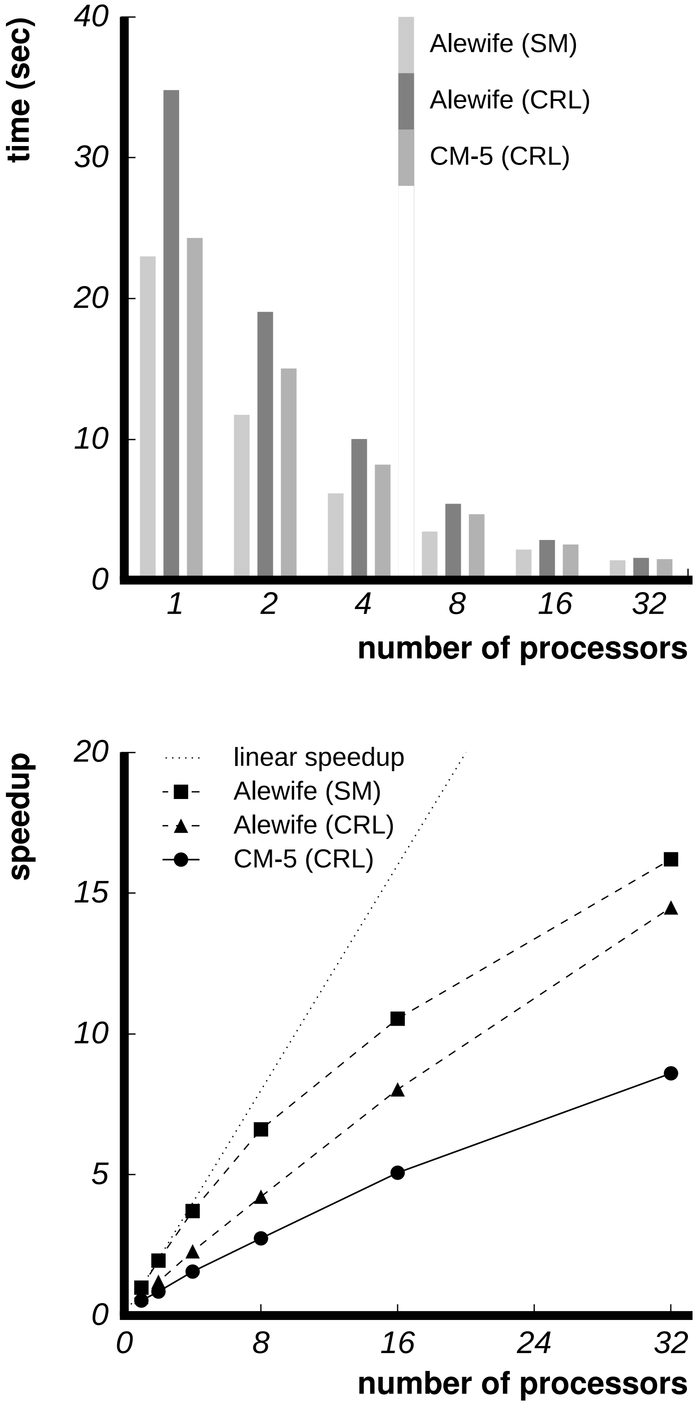


Figure 6-3. Absolute running time (top) and speedup (bottom) for Barnes-Hut (4,096 bodies).

6.3 CRL vs. Shared Memory

In order to address the question of whether a CRL implementation built on top of high-performance communication mechanisms is capable of delivering performance competitive with that provided by hardware DSM implementations, we compare the performance of the Alewife CRL and Alewife SM versions of the three applications.

As can be seen in Figure 6-1, both Alewife CRL and Alewife SM perform well for Blocked LU (speedups of 22.3 and 19.7 on 32 processors, respectively). This is not particularly surprising; since Blocked LU exhibits large computation and data granularities, it does not present a particularly challenging communication workload.

Somewhat surprising, however, is the fact that Alewife CRL outperforms Alewife SM by almost 15 percent on 32 processors. This occurs because of LimitLESS software overhead. On 16 processors, only a small portion of the LU data set is shared more widely than the five-way sharing supported in hardware, so LimitLESS software is only invoked infrequently. On 32 processors, this is no longer true: over half of the data set is shared by more than five processors at some point during program execution. The overhead incurred by servicing some portion of these requests in software causes the performance of Alewife SM to lag behind that of Alewife CRL.

For Water, a somewhat more challenging application, both versions of the application again perform quite well; this time, Alewife SM delivers roughly 15 percent better performance than Alewife CRL (speedups of 22.4 and 19.3 on 32 processors, respectively).

This performance difference is primarily due to the fact that the Alewife CRL version uses three small regions to compute global sums once per iteration; each small region must "ping-pong" amongst all processors before the sum is completed. Given Alewife CRL's relatively large base communication latencies, this communication pattern can limit performance significantly as the number of processors is increased. Modifying the source code such that these global sums are computed using CRL's reduction primitives (as was already the case for Barnes-Hut) confirms this; doing so yields an Alewife CRL version of Water that delivers the same speedup at 32 processors as the Alewife SM version of the code. Because the base communication latencies for Alewife's native shared memory are significantly lower than for Alewife CRL, little or no benefit is obtained by applying the same modification to the Alewife SM version of the code (in which the same global sums were originally computed into small regions of shared memory protected by spin locks). One might expect this to change when Alewife systems with more than 32 processors become available.

For Barnes-Hut, the most challenging application used in this study, Alewife SM once again delivers the best performance—a speedup of 16.2 on 32 processors—but Alewife CRL is not far behind with a speedup of 14.5. Thus, while Alewife's aggressive hardware support for coherent shared memory does provide some performance benefit, the performance improvement over Alewife CRL's all-software approach is somewhat less

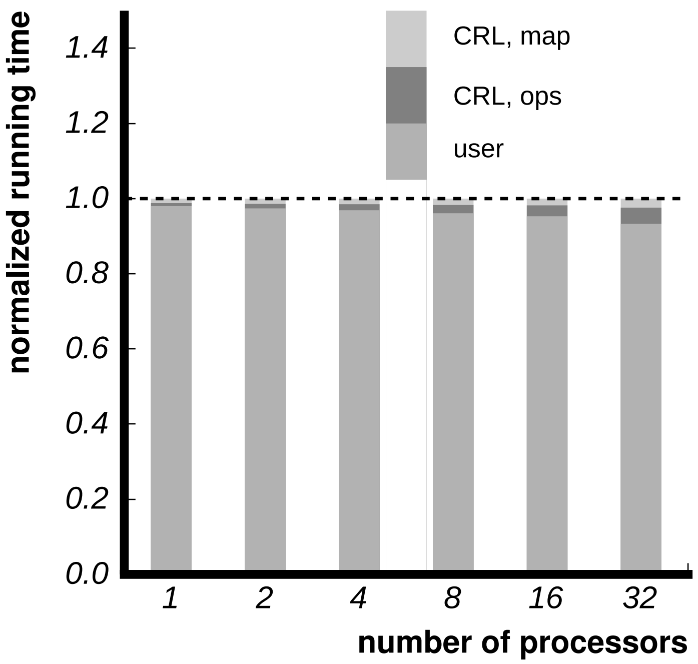


Figure 6-4. Breakdown of normalized running time for Alewife CRL version of Blocked LU (500x500 matrix, 10x10 blocks).

than one might expect (roughly 12 percent; experiments indicate that this gap decreases slightly for larger problem sizes).

Finally, in order to understand how different components of CRL contribute to overall running time, a profiled version of the CRL library was developed. Figures 6-4 through 6-6 show breakdowns of running time for the Alewife CRL versions of each application that were obtained using the profiled library. Normalized running time (for each bar, 1.0 corresponds to the absolute running time for Alewife CRL on that number of processors) is divided into three categories: time spent in CRL executing map/unmap code ("CRL, map"), time spent in CRL starting and ending operations ("CRL, ops"), and time spent running application code ("user"). "CRL, map" and "CRL, ops" include any "spin time" spent waiting for communication events (*i.e.,* those related to calls to `rgn_map` or `rgn_start_`*op* that miss) to complete.

As can be seen in Figure 6-4 the Alewife CRL version of Blocked LU spends very little time executing CRL code—even on 32 processors, only 4.2 and 2.5 percent of the total running time is spent in the CRL library executing operation and mapping code, respectively. Since Blocked LU is a fairly coarsed grained application that achieves good speedups, this is not surprising.

Figure 6-5 shows the profiling information for Water. As was discussed in Section 6.2.4 above, Water's entire data set is kept mapped on all nodes at all times, so none of the

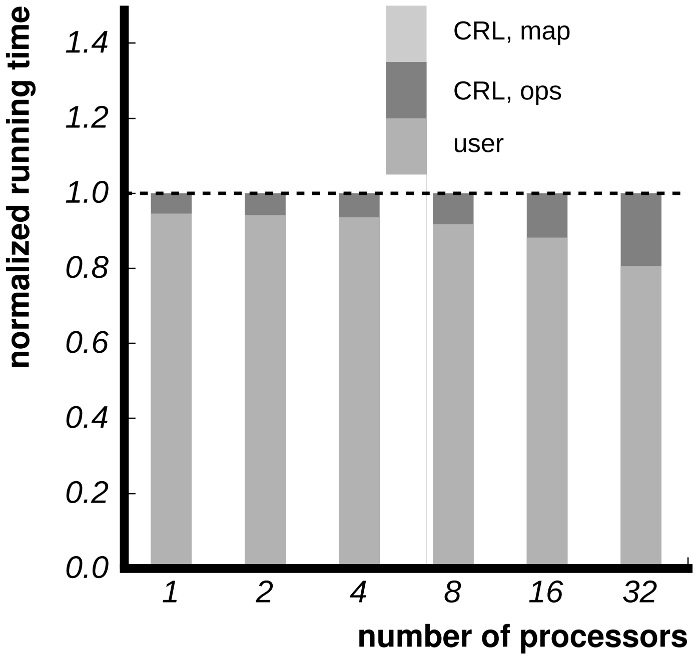


Figure 6-5. Breakdown of normalized running time for Alewife CRL version of Water (512 molecules).

application running time is spent in the CRL library executing mapping code. Time spent executing CRL operation code ranges from roughly 5.3 to 19.5 percent on one and 32 processors, respectively. As discussed above, this increase is primarily due to the use of small regions to compute several global sums and can be addressed effectively by using scalable reduction primitives instead.

Figure 6-6 shows the profiling information for Barnes-Hut. When running on a single processor, approximately one third of the total running time is spent executing CRL code; slightly more than half of this time is spent mapping and unmapping. Not surprisingly, CRL overhead increases as number of processors is increased: at 32 processors, almost half of the total running time is spent in CRL, but now slightly less than half of the overhead is spent mapping and unmapping.

6.4 Changing Communication Costs

The results shown in the previous section demonstrate that when built upon high-performance communication substrates, CRL is capable of delivering performance close to that provided by hardware-supported shared memory, even for challenging applications and small problem sizes. Unfortunately, many interesting platforms for parallel and dis-

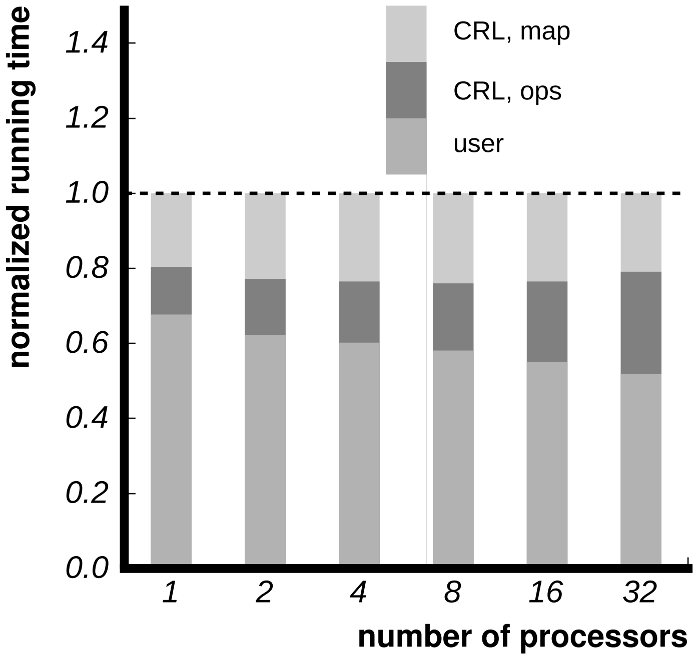


Figure 6-6. Breakdown of normalized running time for Alewife CRL version of Barnes-Hut (4,096 bodies).

tributed computing (*e.g.,* networks of workstations) provide communication performance significantly worse than that found in Alewife.

To gauge the sensitivity of CRL's performance to increased communication costs, we compare the behavior of applications running under Alewife CRL and CM-5 CRL. Although the CM-5 is a tightly-coupled multiprocessor, current-generation network-of-workstations technology is capable of providing similar communication performance [78], so the results for CM-5 CRL are indicative of what should be possible for implementations targeting networks of workstations using current- or next-generation technology.

For Blocked LU, CM-5 CRL delivers respectable performance (a speedup of 13.8 on 32 processors; see Figure 6-1), lagging roughly 30 percent behind the speedup achieved with Alewife CRL. Because Blocked LU uses relatively large regions, this difference can be attributed not only to the higher communication latencies on the CM-5 (1088 cycles for a simple round trip *vs.* 528 cycles for Alewife) but also to the lower bulk-data transfer performance (approximately 8 Mbytes/second *vs.* 18 Mbytes/second for Alewife).

For Water, the performance gap widens, with CM-5 CRL delivering a speedup of 10.4 on 32 processors (46 percent less than Alewife CRL; see Figure 6-2). As was the case for Water under Alewife CRL, however, this figure can be improved upon by using reductions to compute the global sums in Water; doing so increases the speedup on 32 processors to 14.1 (37 percent less than the speedup on 32 processors for the same code running under

Alewife CRL). The remaining performance gap between Alewife CRL and CM-5 CRL can be attributed to the smaller computation granularity of Water (approximately 1,540 cycles of useful work per CRL operation). Even given a relatively low miss rate, this granularity is small enough that the larger miss latencies for CM-5 CRL begin to contribute a significant portion of the total running time, thus limiting the possible speedup.

In spite of this performance gap, CM-5 CRL performs comparably with existing mostly software DSM systems. The CM-5 CRL speedup (5.3 on eight processors) for Water (without reductions) is slightly better than that reported for TreadMarks [38], a second-generation page-based mostly software DSM system (a speedup of 4.0 on an ATM network of DECstation 5000/240 workstations, the largest configuration that results have been reported for)[1].

For Barnes-Hut, CM-5 CRL performance for Barnes-Hut lags roughly 41 percent behind that provided by Alewife CRL (speedups of 8.6 and 14.5 at 32 processors, respectively; see Figure 6-3). As was the case with Water, this is primarily due to small computation granularity; small enough that even given particularly low map and operation miss rates (1.2 and 4.7 percent, respectively), the larger miss latencies of CM-5 CRL cause significant performance degradation.

As was pointed out in Section 6.2.3, the problem size used to obtain the Barnes-Hut results (4,096 bodies) is one-quarter of the suggested problem size (16,384 bodies). Furthermore, even the suggested problem size is fairly modest; it is not unreasonable for production users of such codes (*e.g.,* astrophysicists) to be interested in problems with several hundred thousands bodies or more. Because larger problem sizes lead to decreased miss rates for Barnes-Hut, performance problems due to less efficient communication mechanisms on the CM-5 tend to decrease with larger problem sizes. Figure 6-7 demonstrates this fact by plotting the performance of the CM-5 CRL version of Barnes-Hut on up to 128 processors for both the 4,096 body problem size discussed above and the suggested problem size of 16,384 bodies. For the larger machine sizes (64, 96, and 128 processors), the increased problem size enables speedups 40 to 70 percent better than those for 4,096 bodies. Such results indicate that for realistic problem sizes, even the CM-5 CRL version of Barnes-Hut may be capable of delivering at least acceptable performance.

6.5 Sensitivity Analysis

This section presents results from a set of experiments intended to provide a more detailed understanding of how sensitive CRL performance is to increased communication costs. These experiments utilize a modified Alewife CRL implementation that allows communication performance to be artificially decreased (*i.e.,* higher latency, lower bandwidth) in a "tunable" fashion. Using the modified Alewife CRL implementation, application

[1] The SPLASH-2 version of Water used in this thesis incorporates the "M-Water" modifications suggested by Cox *et al.* [18].

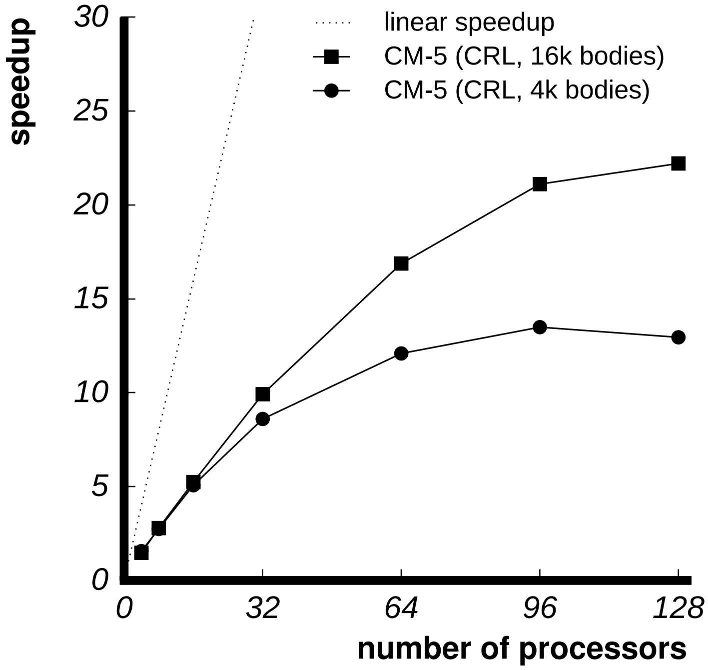


Figure 6-7. Barnes-Hut performance for larger problem (16,384 bodies) and machine sizes (128-node CM-5).

performance can be measured in various reduced-communication-performance scenarios and compared to that obtained with the baseline Alewife CRL implementation. In this manner, one can obtain a much better understanding of CRL's sensitivity to increased communication costs than was possible with the comparison of Alewife CRL and CM-5 CRL.

6.5.1 Modified Alewife CRL Implementation

The modified Alewife CRL implementation allows communication performance to be degraded in two ways: by increasing active message latency or decreasing the effective bandwidth of the bulk data transfer mechanism; both features can be controlled independently. Increases in latency affect *all* active messages; decreased bulk data transfer performance only impacts those that carry bulk transfer data.

Active message latency is increased by inserting a five-instruction delay loop (three loop instructions, two no-ops) before every active message send and at the top of every active message handler. By varying the number of iterations executed each time a delay loop is encountered (a compile-time constant), the effective latency of all active messages can be increased by essentially arbitrary amounts. Since delay loops are inserted on both the sending and receiving side, cycles spent in the delay loops are split equally by senders and receivers.

Bulk data transfer performance is decreased by inserting some number (a compile-time constant) of no-op instructions into the loops that flush send and receive buffers before sending and receiving messages that include bulk transfer data. Since each loop iteration flushes a single 16-byte cache line, each no-op inserted into the flush loops increases the per-byte bulk transfer cost by 0.125 cycles (1 extra cycle per 16 bytes incurred on both sending and receiving nodes).

Careful use of conditional compilation ensures that in the zero extra overhead cases (for both latency and per-byte bulk transfer cost), the resulting code is the same as in the baseline Alewife CRL implementation.

Artificially increasing message latency and decreasing bulk transfer bandwidth by increasing the software overheads incurred by senders and receivers effectively simulates systems in which network interfaces are less and less closely coupled with the processor core (*e.g.,* on the L2 cache bus, memory bus, or an I/O bus) but the network fabric proper retains the relatively favorable latency and bandwidth characteristics of Alewife's EMRC-based network [71]. It seems likely, however, that systems in which software overhead constitutes a smaller portion of end-to-end latency (*e.g.,* because of relatively efficient network interfaces coupled with a less aggressive network fabric) will yield better application performance than systems with the same end-to-end communication performance but latencies are dominated by software overhead: roughly speaking, fewer processor cycles spent in message-delivery overhead means more cycles spent in useful work. Therefore, one expects that the impact on application performance measured with the modified Alewife CRL implementation for a particular combination of message latency and bulk transfer performance is probably somewhat pessimistic for systems that deliver the same communication performance through a combination of lower software overheads and less aggressive networking technology.

6.5.2 Experimental Results

The three applications described in Section 6.2 were linked against modified CRL implementation and run with all combinations of nine message latencies (0, 50, 100, 150, 200, 250, 300, 350, and 400 delay cycles in addition to the base one-way latency of 264 cycles) and nine per-byte bulk transfer costs (0.000, 0.125, 0.375, 0.625, 0.875, 1.125, 1.375, 1.625, and 1.875 cycles/byte in addition to the base performance of 1.110 cycles/byte). Average running times over three consecutive runs were measured on 16- and 32-processor Alewife configurations. For Barnes-Hut, results were obtained for both the default (4,096 body) and suggested (16,384 body) problem sizes. In each case, the figure of merit is not the measured running time, but how much running time increased over the baseline (zero extra overhead) case, expressed as a percentage of the baseline running time.

Increased Latency

Figure 6-8 shows the effect of increasing the one-way message latency from 264 cycles to 664 cycles while keeping the bulk transfer performance fixed at the baseline value (1.110 cycles/byte); the top and bottom plots provide results for 16 and 32 processors, respectively. For the most part, the qualitative observations that can be made about this data serve to confirm what might be intuitively expected; some of the quantitative aspects are rather interesting, however.

First, of these applications, the smaller problem instance of Barnes-Hut is the most sensitive to increases in message latency. Since the smaller problem instance of Barnes-Hut is also the application that induces the heaviest communication workload in terms of per-node message rate (see Table 6-6), this is not surprising. On 32 processors, each additional 100 cycles of message latency increases running time by roughly 4.4 percent of the baseline time.

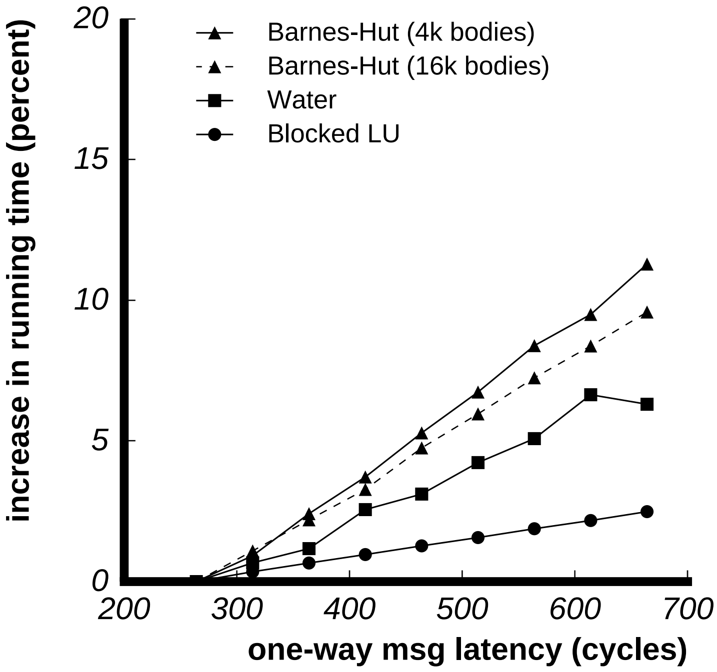
Second, since the larger problem instance of Barnes-Hut induces a lighter communication workload than the smaller problem instance (as is discussed in Section 6.4), the larger problem instance is less sensitive to increases in message latency. On 32 processors, each additional 100 cycles of message latency only increases running time by roughly 2.7 percent of the baseline time. Similarly, the smaller problem instance running on a smaller system (16 processors) is also less sensitive to increases in message latency.

Third, of these applications, Blocked LU is the least sensitive to increases in message latency—approximately 4.7 and 2.9 times less sensitive than the smaller Barnes-Hut problem instance on 16 and 32 processors, respectively. Given that Blocked LU is the least challenging of these applications in terms of the communication workload it induces, this is not surprising.

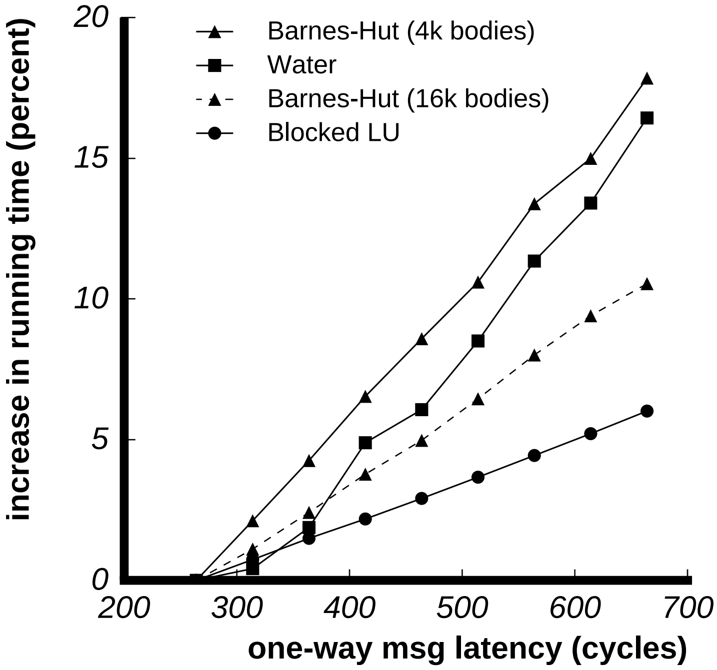
Finally, perhaps the most interesting observation that can be made from the data shown in these plots has to do with the impact of *decreasing* message latency instead of increasing it. As discussed in Section 5.2.2, protocol message handlers on Alewife are effectively transitioned into full-fledged threads before executing. With the first-run CMMU silicon, this adds approximately 6.2 microseconds (124 cycles) to the one-way message latency, but planned functionality extensions in the second-run CMMU parts will make it possible to effect this transition much more rapidly. Assuming it is reasonable to extrapolate the curves shown in Figure 6-8 beyond the range of the measurements, eliminating this overhead entirely (and thus nearly halving the message latency) would only reduce the running time of the *most* latency sensitive application (the smaller Barnes-Hut problem instance running on 32 processors) by five to six percent.

Decreased Bandwidth

Figure 6-9 shows the effect of increasing the bulk transfer costs from 1.110 to 2.975 cycles/byte while keeping the message latency fixed at the baseline value (264 cycles).

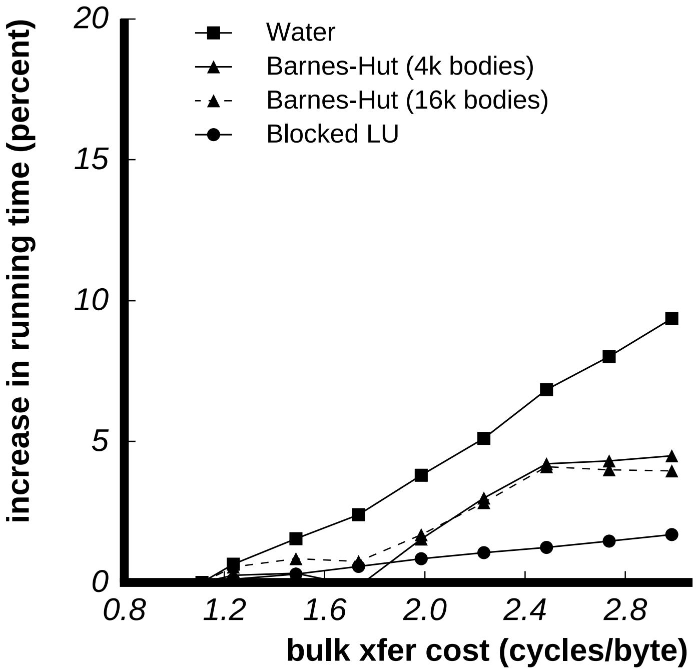


(a) 16 processors

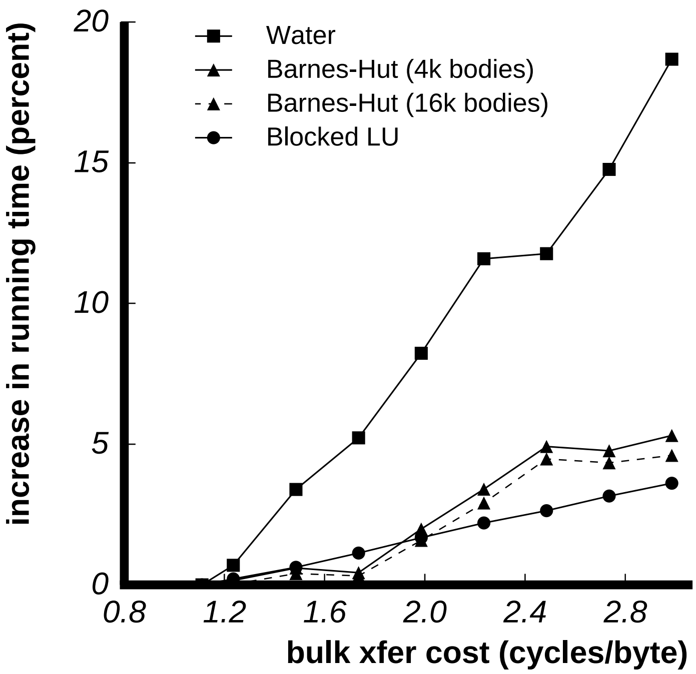


(b) 32 processors

Figure 6-8. Impact of increased message latency on application performance.



(a) 16 processors



(b) 32 processors

Figure 6-9. Impact of decreased bulk transfer bandwidth on application performance.

As in Figure 6-8, results are shown both for 16 (top) and 32 processors (bottom). The intuitive expectations about the effect of changing problem size and number of processors on latency sensitivity discussed above (roughly, that larger problem sizes and smaller machine sizes result in lower sensitivity) continue to hold with respect to sensitivity to changes in bandwidth.

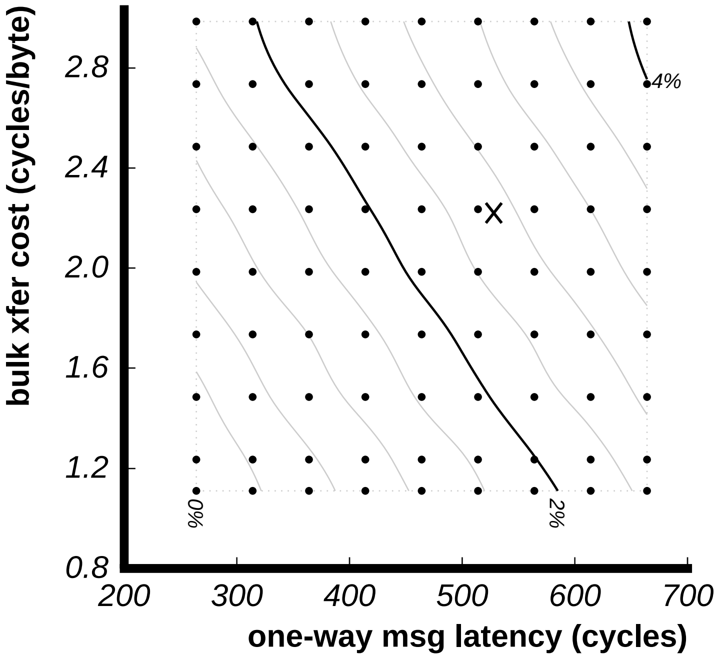
Perhaps the most interesting observation that can be made from this data is that only Water is particularly sensitive to increased bulk transfer costs: On 32 processors, increasing the per-byte cost (and thus decreasing the bandwidth) by a factor of 2.7 causes the running time for Water to increase by nearly 19 percent. This increase occurs because (1) most of the regions used in Water are relatively large (672 bytes), so data-transfer time constitutes a significant fraction of the latency for data-carrying protocol messages and (2) miss rates on those regions are high enough that Water induces a relatively high per-node message rate. For the other applications, the same increase in per-byte bulk-transfer costs has a much smaller impact on running time (between 3.6 and 5.3 percent).

As was discussed in Section 5.2.1, the flushing of message buffers required with the first-run CMMU parts reduces the effective peak bandwidth of Alewife's bulk data transfer mechanism from approximately 2.2 to 0.9 bytes/cycle. Once again, a simple extrapolation can be used to estimate the performance improvement that should be possible when second-run CMMU parts are available and message buffers need not be flushed. For the most bandwidth sensitive application (Water, running on 32 processors), the estimated performance improvement is between six and seven percent.

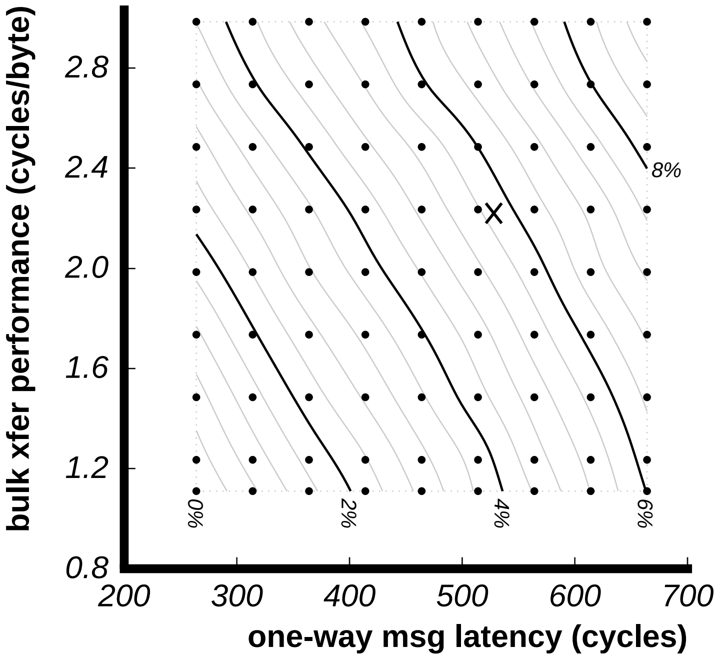
Combined Effects

Although interesting, Figures 6-8 and 6-9 are limited by the fact that they keep one parameter (latency or bandwidth) fixed at the baseline value while varying the other. In contrast, Figures 6-10 through 6-13 show the combined effects of simultaneously increasing message latency and bulk transfer costs. As with the previous figures, each figure shows results for both 16 (top) and 32 processors (bottom). The horizontal and vertical axes of each plot indicate one-way message latency and per-byte bulk transfer cost, respectively. The body of each plot is a "contour diagram" showing lines of constant performance impact (measured in percent increase in running time over the baseline case); minor and major contour intervals of 0.4 and 2 percent are used. In each plot, small filled circles indicate measured data points; the contour surface is derived using a simple polynomial interpolation. In addition, an 'X' in each plot indicates the point where communication performance is half as good as the baseline Alewife CRL implementation (twice the latency, half the bulk transfer bandwidth).

Several observations can be made about this data. The relatively wide spacing of contour lines in Figure 6-10 confirm the previous observations that Blocked LU is relatively insensitive to increasing communication costs. In contrast, the close spacing of contour lines in Figure 6-11 indicate that of the applications used in this study, Water is perhaps the most sensitive to increasing communication costs. In part, this is probably due to Water's

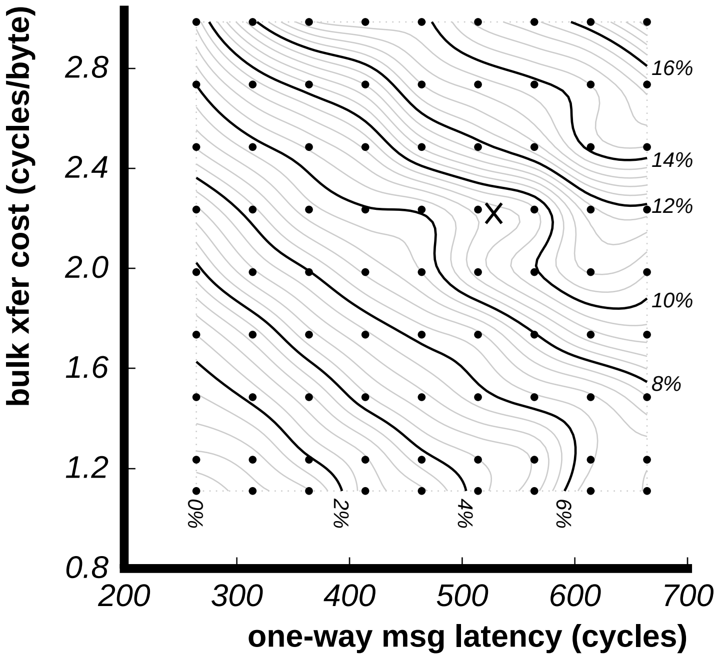


(a) 16 processors

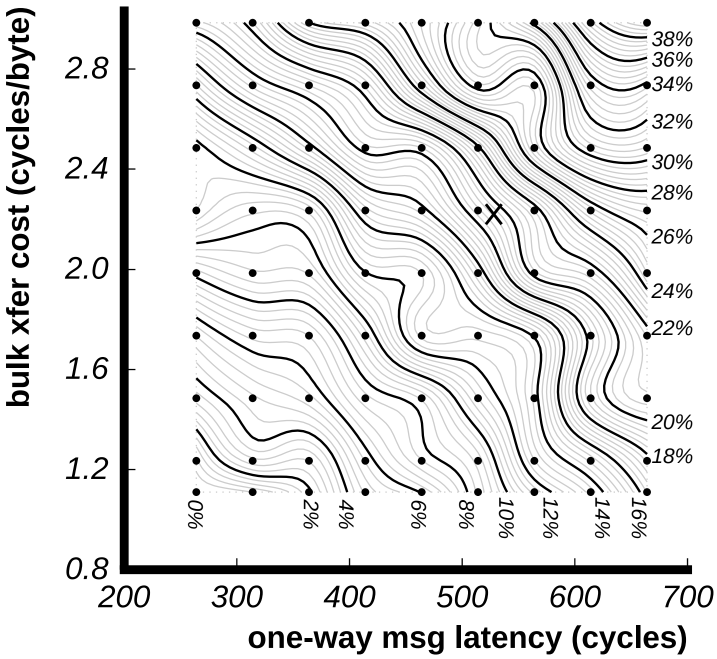


(b) 32 processors

Figure 6-10. Contour plots indicating combined impact of increased communication costs on Blocked LU (500x500 matrix, 10x10 blocks).

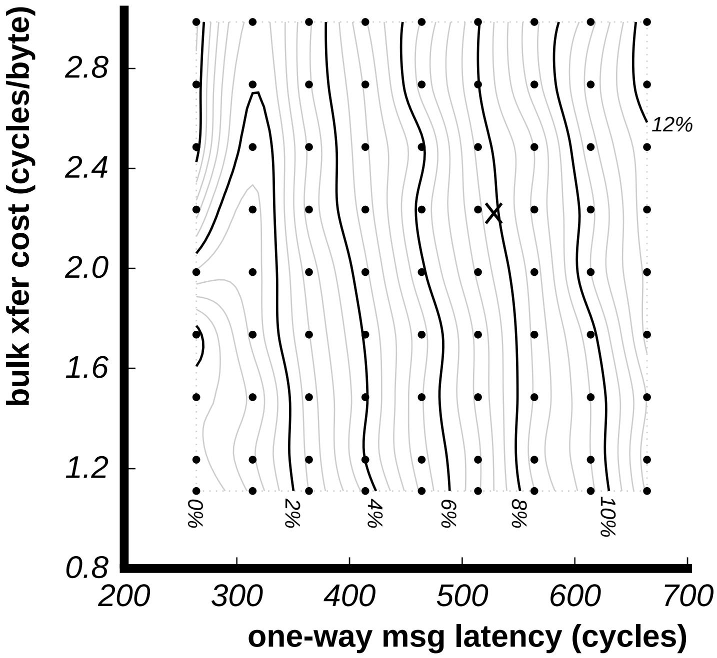


(a) 16 processors

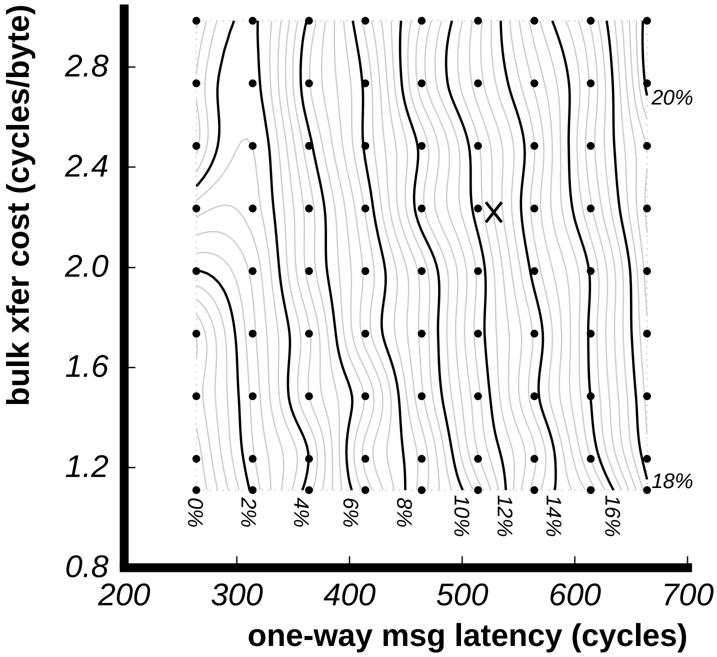


(b) 32 processors

Figure 6-11. Contour plots indicating the combined impact of increased communication costs on Water (512 molecules).

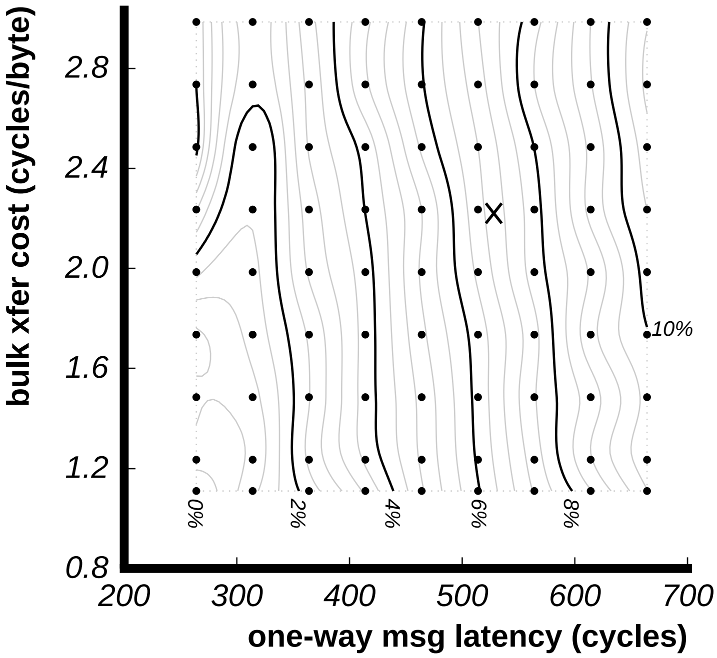


(a) 16 processors

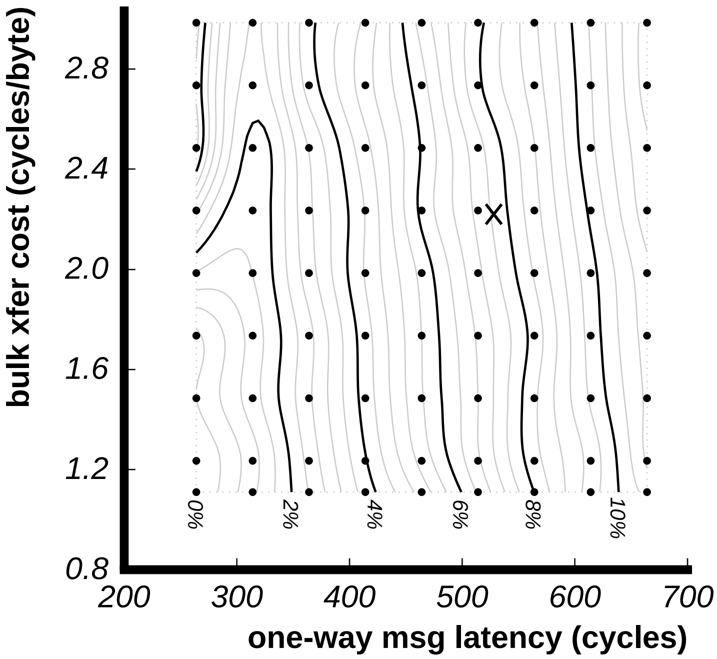


(b) 32 processors

Figure 6-12. Contour plots indicating the combined impact of increased communication costs on Barnes-Hut (4,096 bodies).



(a) 16 processors



(b) 32 processors

Figure 6-13. Contour plots indicating the combined impact of increased communication costs on Barnes-Hut (16,384 bodies).

use of three small regions to compute global sums, as discussed in Section 6.3; it seems likely that the modified version of Water where global sums are computed using CRL's reduction primitives would exhibit lower sensitivity to increases in message latency.

The extent to which the contour lines in Figure 6-11 deviate from smooth diagonal lines (as in Figure 6-10) can be attributed to experimental noise and the use of a contouring interpolation algorithm that guarantees the resulting contour surface passes through all measured data points. For example, Table B-18 (in Appendix B) shows the raw data for Water running on 32 processors used to produce Figure 6-11(a). As can be seen in the table, the standard deviations of the various running times (computed over three consecutive application runs) range from around one hundred thousand to one million cycles with a mean value of around 440,000 cycles, approximately 1.9 percent of the baseline running time of 23.68 million cycles.

To provide insight into the extent to which these data points deviate from a single plane (which would result in smooth diagonal contour lines), a bilinear regression is used to obtain a best-fit plane for the same data. Regression errors (the difference between the value predicted by the regression and the actual measured value) are then computed for each of the 81 data points. For Water running on 32 processors, the standard deviation of these regression errors is around 380,000 cycles, approximately 1.6 percent of the baseline running time. Since this value is somewhat smaller than the average standard deviation of running times (computed over three measurements) discussed in the previous paragraph, it seems likely that the kinks and bumps in the contour lines of Figure 6-11 would smooth out if a larger number of measurements were made at each data point.

A second anomaly occurs along the left edge of the contour plots for Barnes-Hut (Figures 6-12 and 6-13). In these figures, for bulk transfer costs of two or more cycles per byte, the initial increase of one-way message latency from the baseline value consistently results in performance *improvements* of as much as two percent. This anomaly appears to be a result of small changes in the cache alignment of key data structures or loops caused by the insertion of the delay loop code before and after every message send and receive; such cache alignment effects are a well-known problem with unified, direct-mapped caches such as those used in Alewife.

This hypothesis was explored by comparing Barnes-Hut running times obtained with zero extra latency to those obtained with the extra one-way message latency set to the smallest non-zero amount possible (10 cycles) without changing the alignment of the code used in the modified CRL implementation. Given the trends observed for the rest of the Barnes-Hut data, such a small increase in one-way message latency (10 cycles over the baseline 264 cycles) should result in an increase in running time of approximately half a percent. Instead, even this small *increase* in one-way message latency consistently resulted in as much as a 3.5 percent *decrease* in running time. Since this anomaly has only been observed between data points measured for the baseline one-way message latency (where delay loops *are not* present in the compiled code) and data points with the minimum amount of additional latency (where delay loops *are* present in the compiled code), it seems likely performance differences can be attributed to the small cache alignment changes caused by adding the delay loops to the compiled code.

In Figures 6-10 and 6-11, contours generally trace a diagonal path from upper left to lower right, indicating that the applications (Blocked LU and Water) are roughly equally sensitive to increases in latency and bandwidth. In contrast, the contours shown in Figures 6-12 and 6-13 (corresponding the smaller and larger problem instances of Barnes-Hut, respectively) are mostly vertical, confirming the earlier observations that Barnes-Hut is fairly sensitive to increases in message latency, but relatively insensitive to decreases in bulk transfer performance.

Finally, it is worth noting that for a system with twice the message latency and half the bulk transfer bandwidth of the baseline Alewife CRL implementation (which in turn provides message latency and bandwidth approximately twice and half that of what should be possible on Alewife with second-run CMMU parts), the largest increase in running time over the baseline system is only around 20 percent (Water on 32 processors). Given that all of the problem sizes used in these experiments are relatively modest (even the larger Barnes-Hut problem instance) and larger problem sizes frequently result in decreased sensitivity to increased communication costs, this bodes well for the likely usefulness of systems like CRL on systems with faster processors and less aggressive networking technology.

6.6 Summary and Discussion

Two major conclusions can be drawn from the results presented in this chapter. First, the CRL implementation on Alewife demonstrates that given efficient communication mechanisms, an all-software DSM system can achieve application performance competitive with hardware-supported DSM systems, even for challenging applications. For example, for Barnes-Hut with 4,096 particles, Alewife CRL delivers a speedup on 32 processors within 12 percent of that for Alewife SM. As discussed above, this gap should narrow somewhat after the Alewife CMMU respin.

Second, these results indicate the extent to which this level of performance depends on high-performance communication mechanisms. In particular, they indicate that even on a system with communication performance half as good as that used to obtain the baseline Alewife CRL results (and thus roughly one-quarter as good as what should be possible after the CMMU respin), CRL should still be able to provide performance within around 30 percent of that provided by Alewife's hardware supported shared memory. Further, although communication performance of the CM-5 is not sufficient for CRL to be competitive with hardware-supported DSMs, it does appear to be sufficient to allow reasonable speedups on realistic problem sizes.

Prefetching

One issue that has not been fully investigated is the extent to which the CRL functions used to initiate operations provide a performance advantage by fetching an entire region's worth of data in a single action. In contrast, the shared memory versions of applications (in which no prefetching is used) fetch shared data on demand, one cache line at a time. When only a small fraction of a shared data structure is referenced in a single operation (or related group of shared memory references by a single thread), fetching data on a cache-line by cache-line basis may be better than spending the (primarily network) resources necessary to fetch an entire region or shared data structure. In contrast, when a large fraction of a region or shared data structure is accessed at once, fetching the entire region is clearly a better solution. In either case, it is not unreasonable to expect that some applications may realize improved performance through careful application of prefetching techniques. For the applications used in this study, however, it is unclear whether or not employing prefetching in the shared memory versions would yield significant performance benefits (*e.g.,* Blocked LU is already almost entirely compute-bound; Barnes-Hut consists primarily of "pointer chasing" code that can be difficult to prefetch effectively).

A simple experiment that might shed some light on this issue would involve modifying the shared memory versions of applications by inserting prefetch instructions every place the CRL version of the same application initiates an operation, in each case mimicking the effect of the `rgn_start_`*op* call by obliviously prefetching the entire data structure being referenced. Unfortunately, obliviously fetching potentially large amounts of data in this manner without consuming it on the fly interacts poorly with the prefetch mechanisms provided by Alewife's shared memory system[2], so it has not been possible to perform this experiment.

Migratory Data

The performance improvement realized by modifying the CRL version of Water to compute global sums using CRL's reduction primitives instead of small regions that get ping-ponged amongst nodes point to one weakness of the CRL system: Because the latencies of basic communication events in CRL (region operations) are quite large, applications that share small data objects in a migratory fashion (*e.g.,* in a read-modify-write pattern, like the global sums in Water) may perform poorly, especially when compared to the same applications running on hardware-supported DSM systems. In the case of Water, this migratory sharing pattern was easily identified and regular enough that replacing it with an efficient primitive (a global reduction implemented using message passing) with far more favorable scalability properties required little work. Although there has been some work related to optimizing coherence protocols to improve performance on migratory data in the context of hardware-based DSM systems [20, 76], it remains unclear whether migratory data sharing will dominate for applications other than those studied in this thesis,

[2] John D. Kubiatowicz, personal communication, August 1995.

and, if so, how frequently can be replaced with carefully selected, efficient message-based primitives. The extent to which any migratory data problem can be addressed using other techniques such as function shipping [11] and computation migration [31, 32] probably also warrants further investigation.

Chapter 7

Related Work

CRL builds on a large body of research into the construction of distributed shared memory systems. However, as discussed in Section 1.1, four key properties distinguish CRL from other DSM systems: simplicity, portability, efficiency, and scalability. Proper subsets of these features have appeared in previous systems, but CRL is unique in providing all four in a simple, coherent package.

This chapter provides a brief overview of research related to CRL; it is divided into three sections. The first section discusses region- and object-based software DSM systems. The second section describes other software DSM work, including page-based systems. Finally, the third section discusses other research that provides comparative results (*e.g.,* all-software *vs.* mostly software, message passing *vs.* shared memory).

7.1 Region- and Object-Based Systems

Except for the notion of mapping and unmapping regions, the programming interface CRL presents to the end user is similar to that provided by Shared Regions [68]; the same basic notion of synchronized access ("operations") to regions ("objects") also exists in other programming systems for hardware-based DSM systems (*e.g.,* COOL [15]). The Shared Regions work arrived at this interface from a different set of constraints, however: their goal was to provide software coherence mechanisms on machines that support non-cache-coherent shared memory in hardware. CRL could be provided on such systems using similar implementation techniques and defining `rgn_map` and `rgn_unmap` to be null macros.

In [47], Lee describes Concord, a software DSM system that employs a region-like approach similar to that used in Shared Regions and CRL. Concord is implemented in the context of a new language (an extension of C called "High C"); applications written in this new language are compiled to C++ with calls to a runtime library that provides appropriate communication, coherence, and synchronization facilities. Although it is possible that such a system organization may be useful from the perspective of portability, the paper does not address this issue. In fact, in addition to not discussing any issues related to portability, the paper makes no mention of what system primitives are required from potential target platforms. (Although it is not specifically addressed, the fact that the paper specifically mentions virtual addresses in several places raises the question of whether or not Concord requires virtual memory hardware.) By including support for non-contiguous and dynamically changing coherence units, Concord also presents a somewhat more complicated interface than CRL. Since no low-level performance measurements of Concord mechanisms are provided, it is difficult to ascertain what impact this complexity has on efficiency (*i.e.,* the amount of software overhead above and beyond that required by the basic communication mechanisms). Finally, since Concord performance results are only presented for a relatively outdated platform (a 32-processor Intel iPSC/2), it is unclear how these results compare to similar results measured on other hardware- and software-based DSM systems.

Chandra *et al.* [14] propose a hybrid DSM protocol in which region-like annotations are used to demark access to regions of shared data. Coherence for regions thus annotated is provided using software DSM techniques analogous to those used by CRL; hardware DSM mechanisms are used for coherence on all other memory references. All synchronization must be effected through hardware DSM mechanisms. In contrast, CRL is an all-software DSM system in which *all* communication and synchronization is implemented using software DSM techniques.

Of all other software DSM systems, Cid [60] is perhaps closest in spirit to CRL. Like CRL, Cid is an all-software DSM system in which coherence is effected on regions ("global objects") according to source code annotations provided by the programmer. Cid differs from the current CRL implementation in its potentially richer support for multithreading, automatic data placement, and load balancing. To date, Cid has only been implemented and used on a small cluster of workstations connected by FDDI[1]. CRL runs on two large-scale platforms and has been shown to deliver performance competitive with hardware DSM systems.

Several all-software DSM systems that employ an object-based approach have been developed (*e.g.,* Amber [17], Concert [35], Orca [3]). Like these systems, CRL effects coherence at the level of application-defined regions of memory ("objects"). Any necessary synchronization, data replication, or thread migration functionality is provided automatically at the entry and exit of methods on shared objects. Existing systems of this type either require the use of an entirely new object-oriented language [3, 35] or only allow the use of a subset of an existing one [17]. In contrast, CRL is not language specific; the basic CRL interface could easily be provided in any imperative programming language.

Scales and Lam [69] have described SAM, a shared object system for distributed memory machines. SAM is based on a new set of primitives that are motivated by optimizations commonly used on distributed memory machines. Like CRL, SAM is

[1] Rishiyur S. Nikhil, personal communication, March 1995.

implemented as a portable C library. Informal discussion with Scales indicates that SAM delivers approximately 35 percent better speedup than CRL for Barnes-Hut when running on the CM-5 with a problem size of 16,384 bodies. This advantage is enabled by additional information about communication patterns provided through SAM's new communication primitives. SAM's performance edge comes at a cost, however: Because the primitives SAM offers are significantly different than "standard" shared memory models, converting existing shared-memory applications to use SAM is likely to be more difficult than converting them to use CRL.

Interestingly, the implementation techniques used in CRL (in particular, the coherence protocol) is quite similar to *callback locking* [23], an algorithm for maintaining the coherence of cached data in distributed database systems. Both the CRL protocol and callback locking allow remote nodes to cache shared or exclusive copies of data items, which are invalidated ("called back") by the home node (server responsible for managing a data item) as necessary to satisfy other requests. Furthermore, similar schemes have also been used to maintain cache consistency in distributed file systems such as Andrew [30] and Sprite [59].

7.2 Other Software DSM Systems

TreadMarks [38] is a second-generation page-based (mostly software) DSM system that implements a release consistent memory model. Unlike many page-based systems, TreadMarks is implemented entirely in user space; virtual memory protection mechanisms are manipulated through library wrappers around system calls into the kernel. Since these virtual memory mechanisms and associated operating system interfaces are relatively standard in current commodity workstations, TreadMarks is fairly portable. There are interesting platforms (*e.g.,* CM-5, Alewife) that lack the support required to implement TreadMarks, however; we believe that this will continue to be the case. In addition, the software overhead of systems like this (*e.g.,* from manipulating virtual memory mechanisms and computing diffs) can be large enough to significantly impact delivered application performance [21].

Midway is a software DSM system based on entry consistency [5]. As discussed in Section 3.4, CRL's programming model is similar to that provided by Midway. An important difference, however, is that Midway requires a compiler that can cull user-provided annotations that relate data and synchronization objects from the source code and provide these to the Midway run-time system. By bundling an implicit synchronization object with every region, CRL obviates the need for special compiler support of this sort. Both mostly software and all-software versions of Midway have been implemented [86]. To the best of our knowledge, Midway has only been implemented on a small cluster of workstations connected with an ATM network.

A number of other approaches to providing coherence in software on top of non-cache-coherent shared-memory hardware have also been explored [19, 42]. Like the

Shared Regions work, these research efforts differ from that described in this thesis both in the type of hardware platform targeted (non-cache-coherent shared memory *vs.* message passing) and the use of simulation to obtain controlled comparisons with cache-coherent hardware DSM (when such a comparison is provided).

7.3 Comparative Studies

Several researchers have reported results comparing the performance of systems at adjacent levels of the classification presented in Section 2.2 (*e.g.,* all-hardware *vs.* mostly hardware [13, 27, 85], mostly software *vs.* all-software [70, 86]), but to our knowledge, only Cox *et al.* [18] have published results from a relatively controlled comparison of hardware and software DSM systems. While their experiments kept many factors fixed (*e.g.,* processor, caches, compiler), they were unable to keep the communication substrate fixed: they compare a bus-based, all-hardware DSM system with a mostly software DSM system running on a network of workstations connected through an ATM switch. Furthermore, their results for systems with more than eight processors were acquired through simulation. In contrast, the results presented in this thesis were obtained through controlled experiments in which only the communication interfaces used by the programming systems were changed. Experimental results comparing hardware and software DSM performance are shown for up to 32 processors (Alewife); software DSM results are shown for up to 128 processors (CM-5).

Klaiber and Levy [40] describe a set of experiments in which data-parallel (C*) applications are compiled such that all interprocessor communication is provided through a very simple library interface. They employ a simulation-based approach to study the message traffic induced by the applications given implementations of this library for three broad classes of multiprocessors: message passing, non-coherent shared memory, and coherent shared memory. In contrast, this thesis shows results comparing the absolute performance of implementations of CRL for two message-passing platforms and compares the delivered application performance to that achieved by a hardware-supported DSM.

In terms of comparing message passing and shared memory, most other previous work has either compared the performance of applications written and tuned specifically for each programming model [11, 16] or looked at the performance gains made possible by augmenting a hardware DSM system with message passing primitives [43]. Such research addresses a different set of issues than those discussed in this thesis, which takes a distributed shared memory programming model as a given and provides a controlled comparison of hardware and software implementations.

Finally, Schoinas *et al.* [70] describe a taxonomy of shared-memory systems that is similar in spirit to that provided in Section 2.2. Their scheme differs from that in Section 2.2 in its focus on processor-side actions and emphasis of specific implementation techniques instead of general mechanisms.

Chapter 8

Conclusions

With this chapter, the thesis comes to a close. The first section summarizes the major results and contributions of the thesis. The second section revisits the goals that drove the design and implementation of CRL in light of the results and experience reported herein. Finally, the third section discusses directions for future work.

8.1 Summary

This thesis has presented C Region Library (CRL), a new all-software distributed shared memory system. The design and implementation of CRL has been described in detail; of particular importance is CRL's focus on the key properties of simplicity, portability, efficiency, and scalability. Although there are modest differences between the programming model provided by CRL and "standard" shared memory models, experience porting the (originally) shared-memory applications used in this study to CRL suggests that any additional programming overhead because of these differences is also quite modest.

Using CRL and the MIT Alewife machine as vehicles, the thesis has presented the first completely controlled comparison of scalable hardware and software DSM systems, a comparison in which only the communication interfaces used by the programming systems are changed; all other system components (*e.g.,* compiler, processor, cache, interconnection network) remain fixed.

The results of this comparison are extremely encouraging, demonstrating that when built upon efficient communication mechanisms, CRL is capable of delivering performance competitive with hardware-based systems: CRL achieves speedups within 15 percent of those provided by Alewife's native support for shared memory, even for challenging applications (*e.g.,* Barnes-Hut) and small problem sizes (one-quarter of the suggested problem size).

Further experimental results show the impact of less efficient communication mechanisms on CRL application performance. These results indicate that even doubling communication costs over the baseline Alewife CRL system (which already is slowed somewhat by software workarounds required for first-run CMMU parts) results in at most a 20 percent performance degradation. Further, these results indicate that even for the significantly lower level of communication performance that is possible with current-generation networks-of-workstations technology, systems like CRL should be able to deliver reasonable performance, even for relatively challenging applications.

For domains and applications that can tolerate a modest deviation from "standard" shared memory programming models, these results indicate the substantial promise of all-software approaches to providing shared memory functionality and suggest that special-purpose hardware support for shared memory may not be necessary in many cases.

8.2 Goals Revisited

Section 3.1 described three goals that guided the design and development of CRL: (1) preserving the essential "feel" of the shared memory programming model; (2) minimizing the functionality required from the underlying hardware and operating system, thus enhancing portability; and (3) developing a simple and lean implementation that eliminates as much software overhead as possible. We believe the current CRL implementation largely meets these goals. Experience porting several applications to CRL and judiciously inserting preprocessor directives so the same sources can be compiled for use with either CRL or shared memory confirm that CRL preserves the essential "feel" of shared memory. The implementation meets the all-software criterion: porting CRL to other message passing environments (*e.g.,* workstations communicating with one another using TCP) has proven to be straightforward. Finally, the performance results shown in the previous section validate the notion that CRL is amenable to simple and lean implementations where the amount of software overhead between applications and the message-passing infrastructure is kept to a minimum.

As discussed in Section 3.5, the programming model provided by CRL is not exactly the same as any "standard" shared memory programming model. The primary differences are two-fold: CRL requires programmers to explicitly manage translations between the shared address space (region identifiers) and the local address space, and CRL requires programmers to insert annotations delimiting accesses to shared data. These modest deviations from standard (*i.e.,* hardware-based) DSM systems are not without reason or benefit. First, they enable CRL's simple and portable library-based implementation style that requires no functionality from the underlying system beyond that necessary for sending and receiving messages. Second, they allow CRL implementations to amortize the cost of providing the mechanisms described in Section 2.2 over entire operations (typically multiple loads and stores) instead of incurring comparable overhead on every reference to potentially shared memory. Furthermore, annotations similar to those required for CRL operations are necessary in some aggressive hardware DSM implementations when writing to shared data (*e.g.,* those providing release consistency). CRL requires such annotations whether reading or writing shared data, similar to the entry consistency model used in Midway [5]. Based on the experience and results described in this thesis, we feel that the additional programming overhead caused by these modest deviations are more than offset by the excellent simplicity, portability, and performance properties of CRL.

8.3 Future Work

A large number of interesting directions for future research follow from the work described in this thesis.

One limitation of the work presented in this thesis is the relatively small number of applications used. Clearly, extending these results with other applications would help improve our understanding of how CRL performs (both when compared to Alewife's native shared memory support and in terms of sensitivity to changing communication costs). It is important to note, however, that the three applications described in Section 6.2 were not hand-picked to show a system like CRL in a favorable light: Water was the most challenging application used in a previous study involving a sophisticated page-based mostly software DSM (TreadMarks) [18]; Barnes-Hut and related hierarchical n-body methods have been advanced as sufficiently important and challenging to serve as a cornerstone of an argument in favor of aggressive hardware-based DSM systems [72, 73].

As part of an effort to get more applications running under CRL, it would probably be interesting to investigate the migratory data issue raised in Section 6.6 further by explicitly looking for applications that might have migratory sharing problems, quantifying the performance impact of migratory sharing on the CRL versions of said applications, and (if possible) identifying efficient communication primitives (*e.g.,* reductions) that can be used to address the problem.

Another interesting direction involves treating CRL as a compiler target for higher-level programming languages. Such an effort could probably be profitably pursued either for relatively standard object-oriented languages (*e.g.,* a parallel version of C++) or various parallel FORTRAN dialects (*e.g.,* FORTRAN-90). Of particular interest would be trying to combine static and dynamic software DSM approaches in a single system that attempts to statically identify communication patterns and pre-schedule efficient message-based communication when possible, but reverts to something like CRL when doing so is not possible.

Developing a better understanding of the relative merits of all-software and mostly software DSM systems would be useful. It seems possible that for systems with much lower-performance communication mechanisms (especially in terms of latency) than those discussed in this thesis, more complex page-based systems like TreadMarks would perform better than CRL. In contrast, given high performance communication, it seems clear that the software overhead of higher-complexity systems could severely limit performance in some cases. Identifying where these regions cross over (if indeed the conjectures are true!) in a space parameterized by latency, bandwidth, and something qualitative along the lines of "communication pattern" would certainly advance the state of the art. Using such a result to build hybrid systems that synthesize aspects of both systems to realize improved performance would be even better.

An implementation of CRL targeting networks of workstations interconnected with some type of fast local area network would be interesting for several reasons, including further validation of systems like CRL as practical (portable and efficient) platforms for parallel computing and as a test workload to drive the development of high performance network interfaces (both hardware and software).

Finally, two items that would probably make CRL a more appealing platform for (non-computer-systems hacker) end users but would not be research *per se*. First, it would be useful to modify the coherence protocol used by CRL to allow non-fixed ("COMA-style") home locations for regions. This relatively straightforward change would allow end users to essentially ignore issues of initial data placement that arise in systems with fixed-home protocols. Second, it would be useful to add support for other platforms to the CRL software distribution. Ultimately, the best demonstration of the viability of a system like CRL would be to convince real users to use it; support for relatively widely-available platforms other than the CM-5 (*e.g.,* IBM SP-1 and SP-2, Intel Paragon, *etc.*) would certainly make it easier for this to happen.

Appendix A

Coherence Protocol

This appendix provides a detailed description of the protocol used by CRL to maintain the coherence of cached data. The first section describes the protocol states and events. The second and third sections describe the home- and remote-side protocol state machines, respectively. The remaining eight sections address various implementation details, including continuations, the mechanisms used to provide atomicity between protocol message handlers and normal threads, and how unexpected and out-of-order protocol messages are handled. Readers that are not interested in this level of detail may find that skimming this material (or skipping it entirely) is more useful than a careful, detailed reading.

To simplify the presentation of the protocol, details regarding the detection and correct handling of out-of-order message delivery are elided from most of the presentation given here. The techniques used to address out-of-order delivery are discussed in Section A.11; those interested in further detail are referred to the CRL source code [33].

A.1 Protocol States and Events

The CRL coherence protocol uses eight states apiece in the home-side and remote-state state machines. These states are described in Tables A-1 and A-2, respectively.

Transitions between protocol states are caused by *events*. Two kinds of events are possible: *calls*, which correspond to user actions on the local processor (*e.g.,* initiating and terminating operations), and *messages*, which correspond to protocol messages sent by other processors. Table A-3 describes the five types of call events used in CRL. In the current CRL implementation, protocol messages in CRL are always either sent from a home node to a remote node (for a given region), or vice versa. Protocol messages related to a particular region are never exchanged between remote nodes. Table A-4 describes the types of protocol messages sent from home nodes to remote nodes (six types of messages); Table A-5 describes those sent from remote nodes to home nodes (eight types of messages).

State	Description
HomeExclusive	This node (the home node) has the only valid copy of the region
HomeExclusiveRip	Like HomeExclusive, plus one or more read operations are in progress locally
HomeExclusiveWip	Like HomeExclusive, plus a write operation is in progress locally
HomeShared	Both the home node and some number of remote nodes have a valid copies of the region
HomeSharedRip	Like HomeShared, plus one or more read operations are in progress locally
HomeIip	An invalidation of remote copies of the region is in progress (to obtain an exclusive copy)
HomeIipSpecial	An invalidation of remote copies of the region is in progress (to obtain a shared copy)
HomeInvalid	A single remote node has a valid copy of the region

Table A-1. CRL home-side protocol states.

State	Description
RemoteInvalid	This node does not have a valid copy of the region
RemoteInvalidReq	Like RemoteInvalid, but a request to obtain a valid copy of the region has been sent
RemoteShared	This node, the home node, and possibly other remote nodes have valid copies of the region
RemoteSharedReq	Like RemoteShared, but a request to obtain an exclusive copy of the region has been sent
RemoteSharedRip	Like RemoteShared, plus one or more read operations are in progress locally
RemoteModified	This node has the only valid copy of the region, and it has been modified
RemoteModifiedRip	Like RemoteModified, plus one or more read operations are in progress locally
RemoteModifiedWip	Like RemoteModified, plus a write operation is in progress locally

Table A-2. CRL remote-side protocol states.

Message	Description
CallStartRead	Initiate a read operation (corresponds to `rgn_start_read`)
CallEndRead	Terminate a read operation (corresponds to `rgn_end_read`)
CallStartWrite	Initiate a write operation (corresponds to `rgn_start_write`)
CallEndWrite	Terminate a write operation (corresponds to `rgn_end_write`)
CallFlush	Flush the region back to the home node (corresponds to `rgn_flush`)

Table A-3. CRL call events.

Message	Description
MsgRInvalidate	Invalidate a remote copy of a region (to obtain a shared copy)
MsgWInvalidate	Invalidate a remote copy of a region (to obtain an exclusive copy)
MsgSharedAckData	Acknowledge a request for a shared copy of a region (includes a copy of the region data)
MsgExclusiveAckData	Acknowledge a request for an exclusive copy of a region (includes a copy of the region data)
MsgModifyAck	Acknowledge a request to upgrade a remote copy of a region from shared to exclusive (does not include a copy of the region data)
MsgModifyAckData	Like MsgModifyAck, but includes a copy of the region data

Table A-4. CRL home-to-remote protocol messages.

Message	Description
MsgInvalidateAck	Acknowledge a message invalidating the local copy of a region (leaves the local copy invalid, does not include a copy of the region data)
MsgInvalidateAckData	Acknowledge a message invalidating the local copy of a region (leaves the local copy invalid, includes a copy of the region data)
MsgRelease	Acknowledge a message invalidating the local copy of a region (leaves a shared copy valid locally, includes a copy of the region data)
MsgSharedReq	Request a shared copy of a region
MsgExclusiveReq	Request an exclusive copy of a region
MsgModifyReq	Request an upgrade of the local copy of a region from shared to exclusive
MsgFlush	Inform the home node that the local copy of a region has been dropped (does not include a copy of the region data)
MsgFlushData	Inform the home node that the local copy of a region has been dropped (includes a copy of the region data)

Table A-5. CRL remote-to-home protocol messages.

A.2 Home-Side State Machine

Figures A-1 through A-8 show the state transition diagrams for the eight home-side protocol states. In each figure, solid arrows indicate state transitions taken in response to protocol events; dashed arrows indicate actions taken because of a "continuation" (the second phase of a two-phase event). Each arrow is labeled with the names of the protocol events which would cause the corresponding state transition to take place; numbers in parentheses after an event name indicate one of multiple possible actions which might happen in response to a protocol event. At the bottom of each figure, two boxes (labeled Ignore and Queue) indicate which protocol events are either ignored (*i.e.,* have no effect) or queued for later processing (see Section A.9). Any protocol events that are not shown in a particular state transition diagram cause a protocol error if they occur; in practice this should only happen if a user attempts an invalid sequence of operations on a region (*e.g.,* a thread that already has a read operation in progress on a particular region attempting to initiate a write operation on the same region without first terminating the read operation).

Figures A-1 through A-8 show only the state transitions that occur in response to protocol events. For other effects, such as manipulations of other protocol metadata or sending protocol messages to other nodes, one should consult Tables A-6 through A-13. These tables provide pseudocode for the actions taken in response to different protocol events for each of the eight home-side states. Any events that are not listed for a particular state cause a protocol error if they occur (as in Figures A-1 through A-8).

Each of Tables A-6 through A-13 consists of three columns. The first and second columns contain the names of the relevant protocol state and event types, respectively. The third column contains pseudocode for the actions that should be taken when the corresponding event occurs in the corresponding state.

Beyond the protocol state, several other components of the home-side protocol metadata associated with each region are referenced in Tables A-6 through A-13. These components are summarized below:

read_cnt: This field is used to count the number of local read operations in progress (simultaneously) for the associated region.

num_ptrs: This field is used to count the number of invalidation messages that have not been acknowledged yet.

tx_cont: This field is used to hold the "continuation" (a pointer to a procedure that implements the second phase) of a two-phase set of actions (*e.g.,* one in which some number of invalidation messages are sent during the first phase, but the second phase cannot be run until all invalidations have been acknowledged). This mechanism is only used in the HomeShared and HomeInvalid states; a "cont:EventType" nomenclature is used to denote the continuation for events of type EventType.

pointer set: The home-side metadata for a region contains a set of "pointers" to remote copies (aka the *directory* for the region); CRL uses a singly-linked list to implement the pointer set. Operations supported on pointer sets include insertion of a new pointer (insert pointer) and deletion of an existing pointer (delete pointer).

message queue: The home-side metadata for a region contains a FIFO message queue that is used to buffer protocol messages that cannot be processed immediately upon reception. Operations supported on message queues include enqueuing a new message (queue message) and attempting to drain the queue by retrying messages from the head of the queue until the queue is empty or the message at the head of the queue cannot be processed (retry queued messages).

Newly created regions (caused by calls to `rgn_create`) start in the HomeExclusive state.

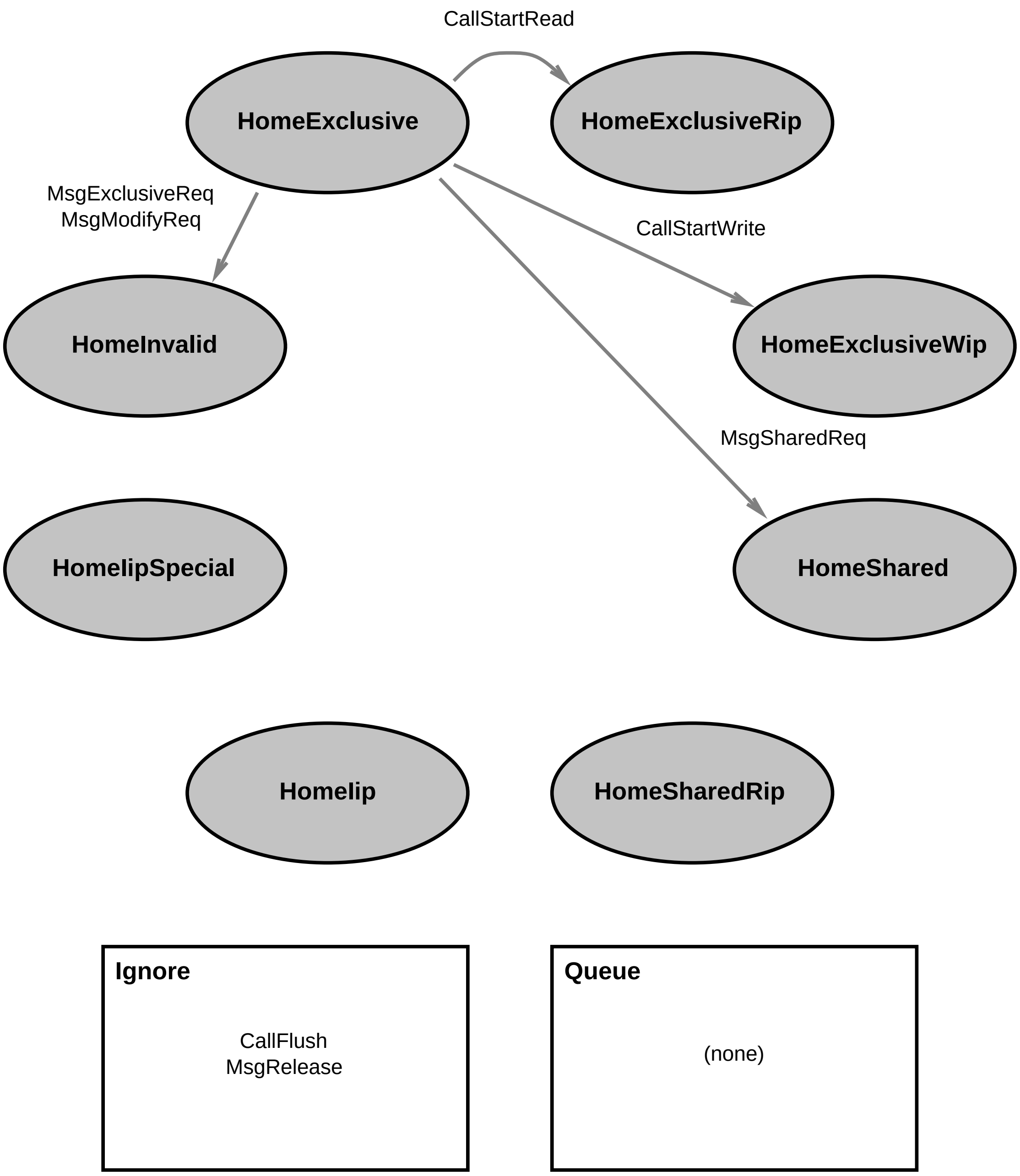


Figure A-1. HomeExclusive: state transition diagram.

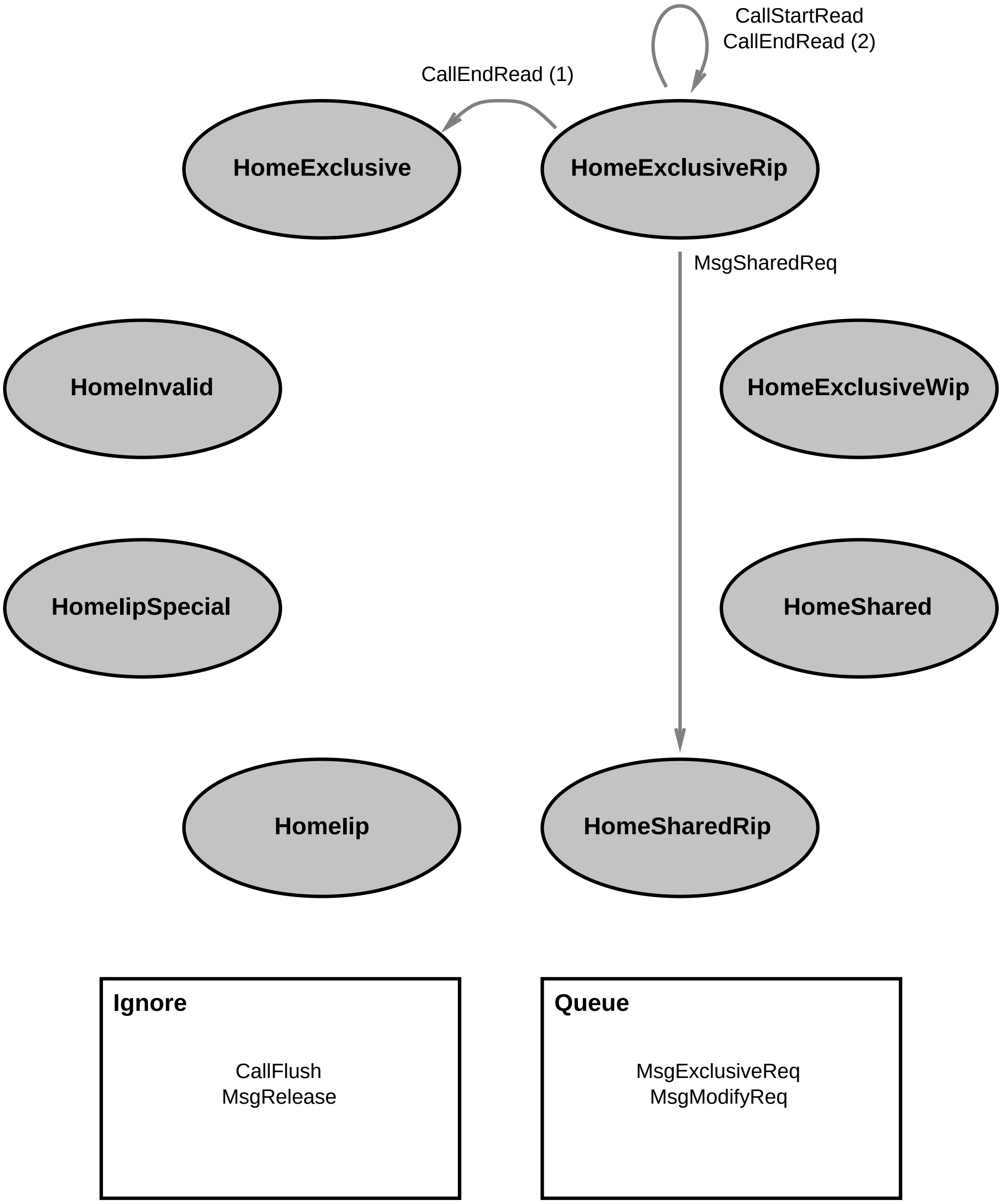


Figure A-2. HomeExclusiveRip: state transition diagram.

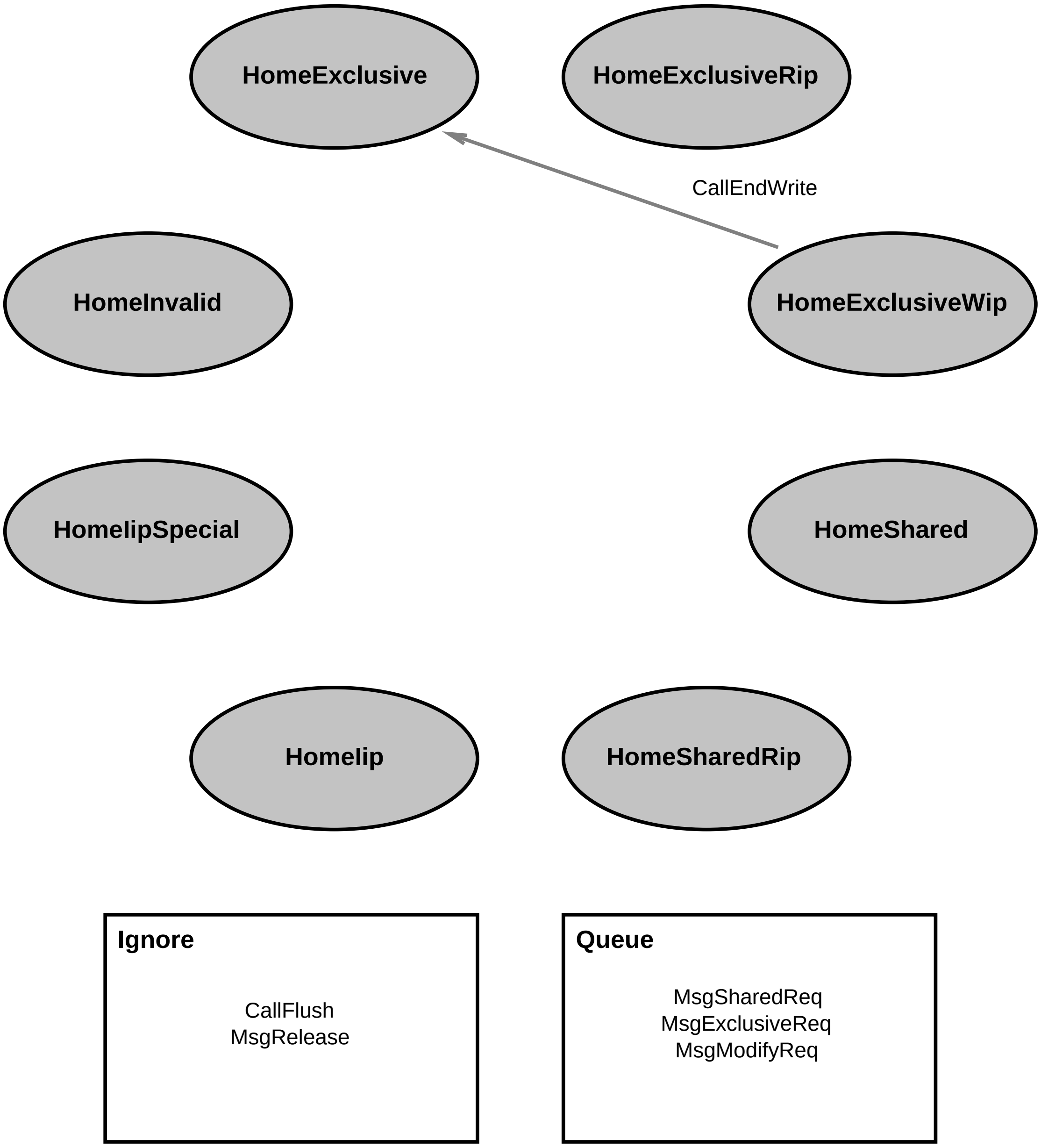


Figure A-3. HomeExclusiveWip: state transition diagram.

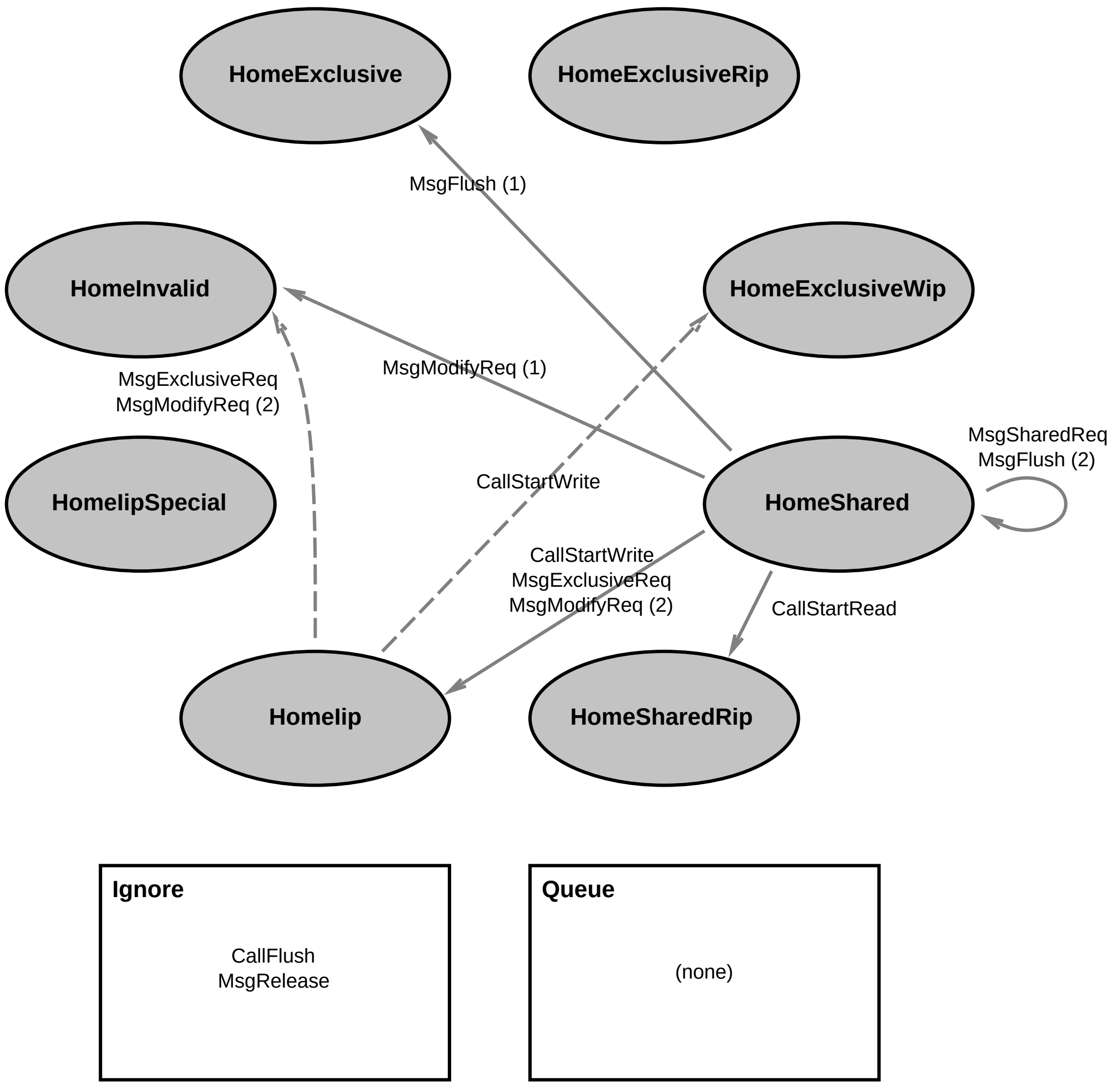


Figure A-4. HomeShared: state transition diagram.

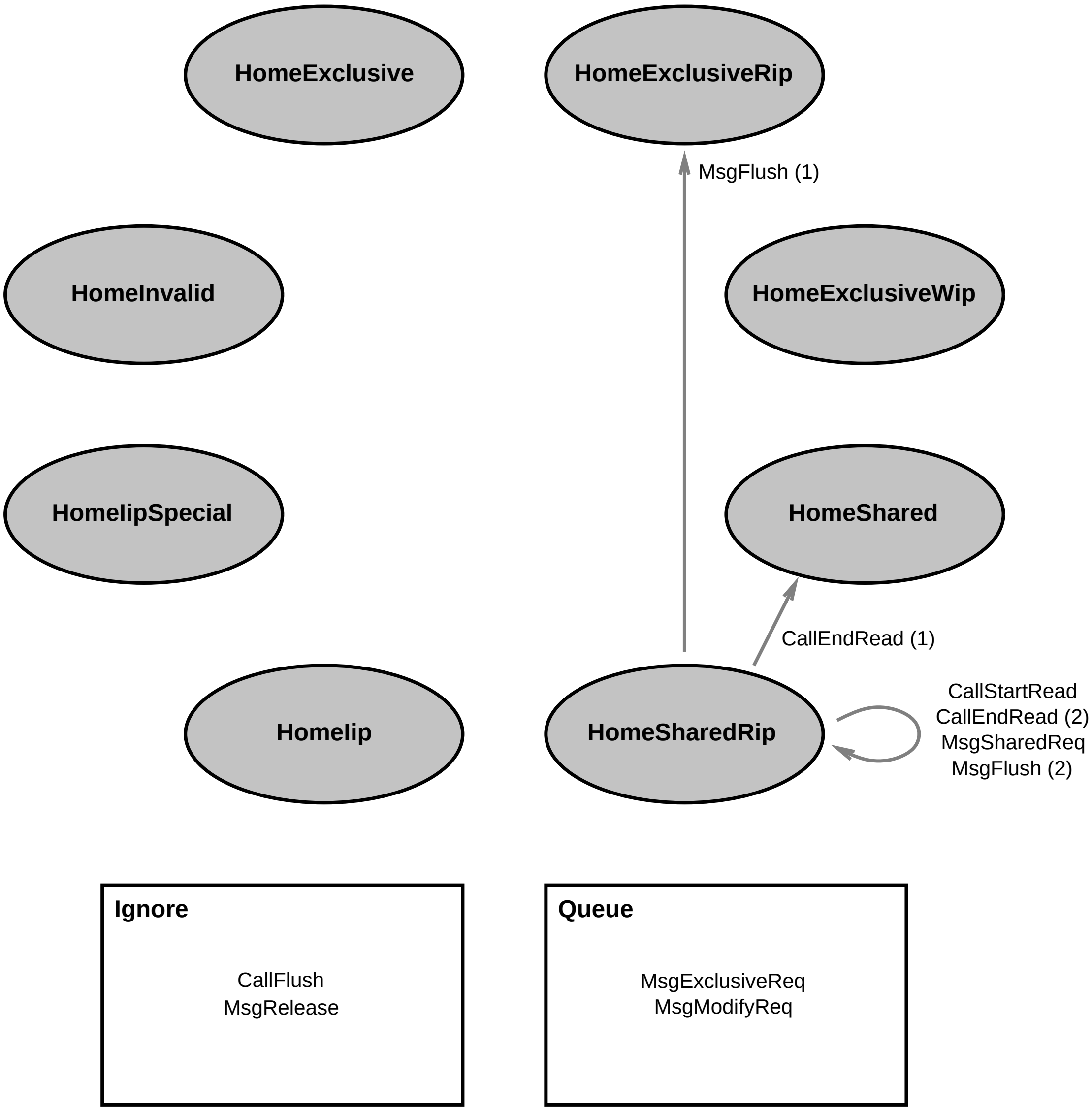


Figure A-5. HomeSharedRip: state transition diagram.

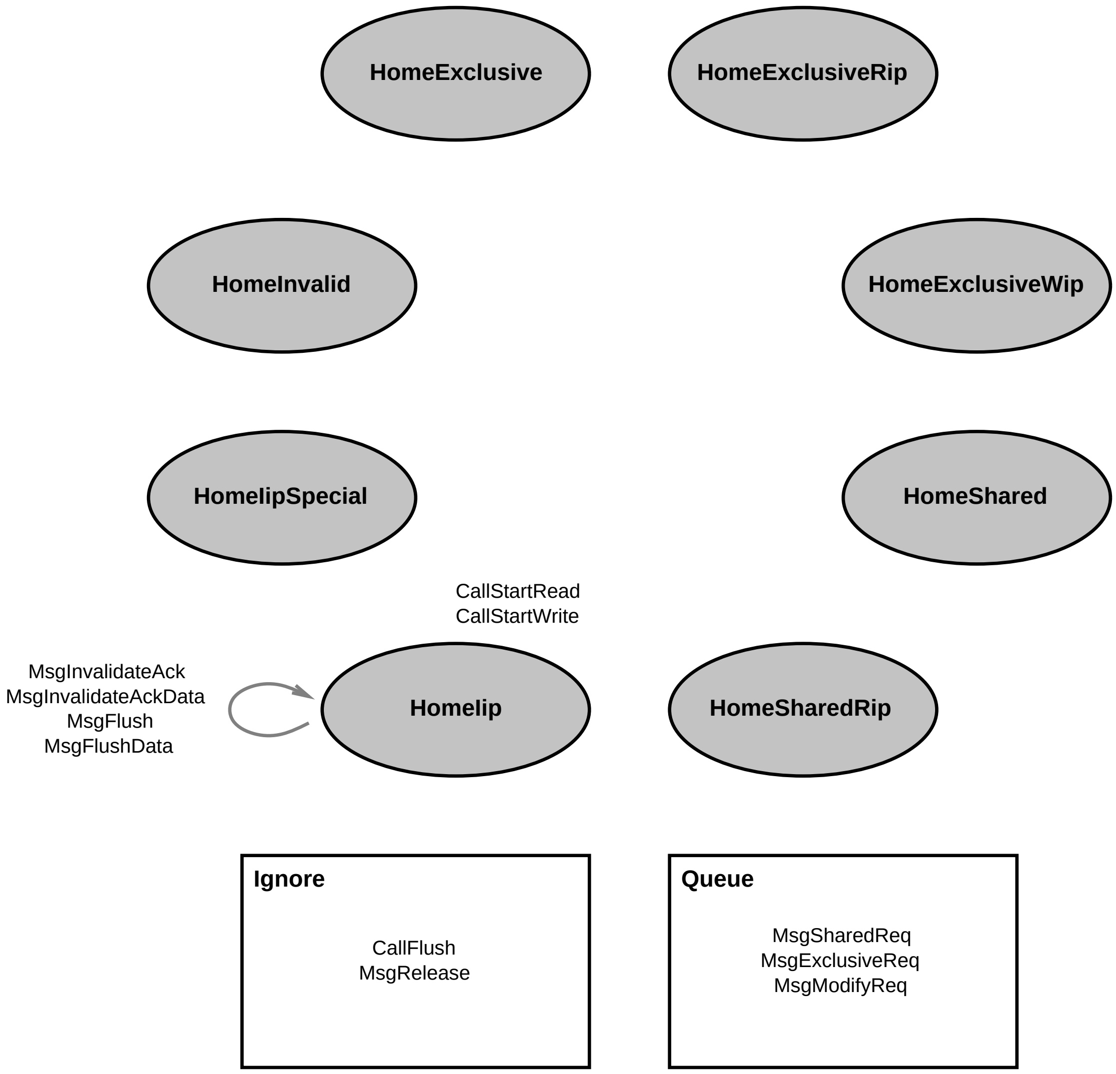


Figure A-6. HomeIip: state transition diagram (see Section A.4 for details).

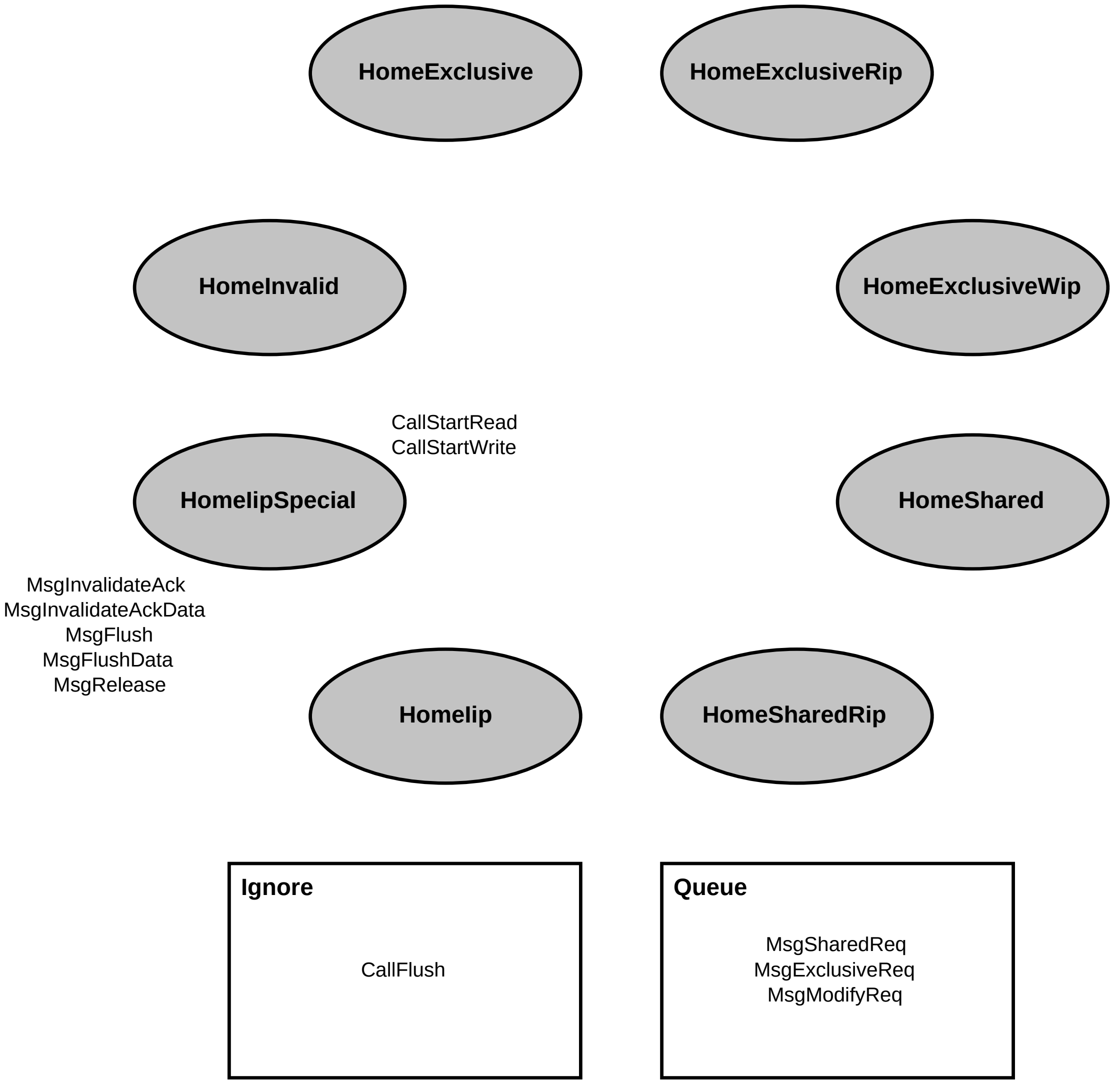


Figure A-7. HomeIipSpecial: state transition diagram (see Section A.4 for details).

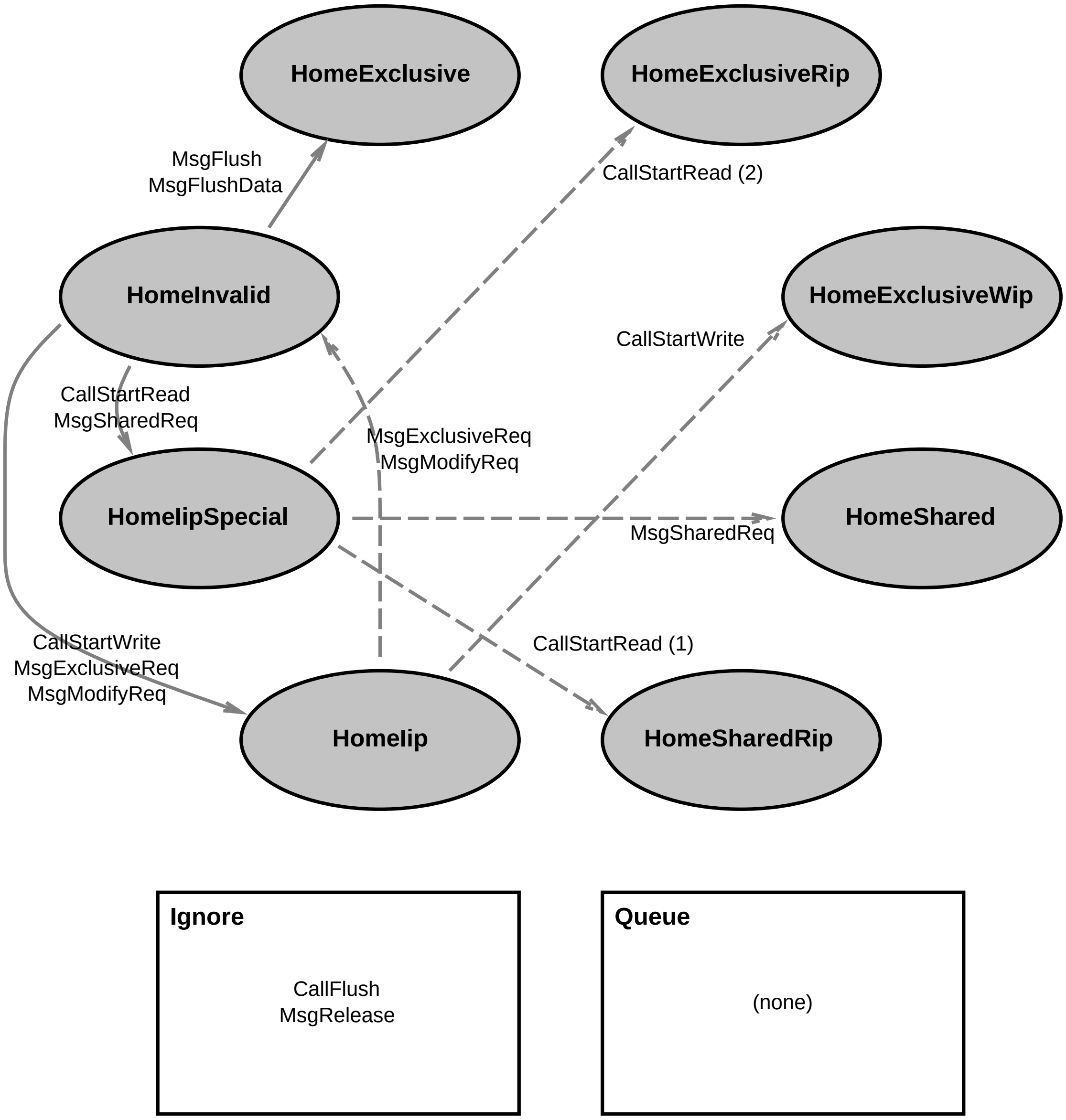


Figure A-8. HomeInvalid: state transition diagram.

State	Event	Actions
HomeExclusive	CallStartRead	*read_cnt* = 1 *state* = HomeExclusiveRip
	CallStartWrite	*state* = HomeExclusiveWip
	CallFlush	do nothing
	MsgSharedReq	send MsgSharedAckData insert pointer *state* = HomeShared
	MsgExclusiveReq	send MsgExclusiveAckData insert pointer *state* = HomeInvalid
	MsgModifyReq	send MsgExclusiveAckData insert pointer *state* = HomeInvalid
	MsgRelease	do nothing

Table A-6. HomeExclusive: protocol events and actions.

State	Event	Actions
HomeExclusiveRip	CallStartRead	*read_cnt* += 1

	CallEndRead	*read_cnt* -= 1
		if (*read_cnt* == 0)
		  state = HomeExclusive
		  retry queued messages

	CallFlush	do nothing
	MsgSharedReq	send MsgSharedAckData insert pointer *state* = HomeSharedRip
	MsgExclusiveReq, MsgModifyReq	queue message
	MsgRelease	do nothing

Table A-7. HomeExclusiveRip: protocol events and actions.

State	Event	Actions
HomeExclusiveWip	CallEndWrite	*state* = HomeExclusive retry queued messages
	CallFlush	do nothing
	MsgSharedReq, MsgExclusiveReq, MsgModifyReq	queue message
	MsgRelease	do nothing

Table A-8. HomeExclusiveWip: protocol events and actions.

State	Event/Continuation	Actions
HomeShared	CallStartRead	*read_cnt* = 1 *state* = HomeSharedRip
	CallStartWrite	send MsgWInvalidates to remote copies *num_ptrs* = # of MsgWInvalidates sent *tx_cont* = cont:CallStartWrite *state* = HomeIip poll until *tx_cont* has been invoked
	cont:CallStartWrite	*state* = HomeExclusiveWip
	CallFlush	do nothing
	MsgSharedReq	send MsgSharedAckData insert pointer
	MsgExclusiveReq	send MsgWInvalidates to remote copies *num_ptrs* = # of MsgWInvalidates sent *tx_cont* = cont:MsgExclusiveReq *state* = HomeIip
	cont:MsgExclusiveReq	send MsgExclusiveAckData insert pointer *state* = HomeInvalid retry queued messages

	MsgModifyReq	if (requesting node is the only pointer)
		  send MsgModifyAck
		  insert pointer
		  state = HomeInvalid
		else
		  send MsgWInvalidates to remote copies
		  num_ptrs = # of MsgWInvalidates sent
		  tx_cont = cont:MsgModifyReq
		  state = HomeIip

	cont:MsgModifyReq	if (requesting node already has a copy)
		  send MsgModifyAck
		else
		  send MsgModifyAckData
		insert pointer
		state = HomeInvalid

	MsgFlush	delete pointer if (no more pointers) *state* = HomeExclusive retry queued messages
	MsgRelease	do nothing

Table A-9. HomeShared: protocol events and actions.

State	Event	Actions
HomeSharedRip	CallStartRead	*read_cnt* += 1

	CallEndRead	*read_cnt* -= 1
		if (*read_cnt* == 0)
		  state = HomeShared
		  retry queued messages

	CallFlush	do nothing
	MsgSharedReq	send MsgSharedAckData insert pointer
	MsgFlush	delete pointer if (no more pointers) *state* = HomeExclusiveRip retry queued messages
	MsgExclusiveReq, MsgModifyReq	queue message
	MsgRelease	do nothing

Table A-10. HomeSharedRip: protocol events and actions.

State	Event	Actions
HomeIip	CallStartRead	wait until *state* != HomeIip retry CallStartRead
	CallStartWrite	wait until *state* != HomeIip retry CallStartWrite
	CallFlush	do nothing
	MsgInvalidateAck, MsgInvalidateAckData, MsgFlush, MsgFlushData	*num_ptrs* -= 1 if (*num_ptrs* == 0) invoke *tx_cont*
	MsgSharedReq, MsgExclusiveReq, MsgModifyReq	queue message
	MsgRelease	do nothing

Table A-11. HomeIip: protocol events and actions.

State	Event	Actions
HomeIipSpecial	CallStartRead	wait until *state* != HomeIip retry CallStartRead
	CallStartWrite	wait until *state* != HomeIip retry CallStartWrite
	CallFlush	do nothing
	MsgInvalidateAck, MsgInvalidateAckData, MsgFlush, MsgFlushData	invoke *tx_cont* with an arg of 0
	MsgRelease	invoke *tx_cont* with an arg of 1
	MsgSharedReq, MsgExclusiveReq, MsgModifyReq	queue message

Table A-12. HomeIipSpecial: protocol events and actions.

State	Event/Continuation	Actions
HomeInvalid	CallStartRead	send MsgRInvalidate to remote copy *tx_cont* = cont:CallStartRead *state* = HomeIipSpecial poll until *tx_cont* has been invoked

	cont:CallStartRead	if (*tx_cont* arg == 1)
		  state = HomeSharedRip
		else
		  state = HomeExclusiveRip
		read_cnt = 1
		retry queued messages

	CallStartWrite	send MsgWInvalidate to remote copy *tx_cont* = cont:CallStartWrite *state* = HomeIip poll until *tx_cont* has been invoked
	cont:CallStartWrite	*state* = HomeExclusiveWip
	CallFlush	do nothing
	MsgSharedReq	send MsgRInvalidate to remote copy *tx_cont* = cont:MsgSharedReq *state* = HomeIipSpecial
	cont:MsgSharedReq	send MsgSharedAckData insert pointer *state* = HomeShared
	MsgExclusiveReq	send MsgWInvalidate to remote copy *tx_cont* = cont:MsgExclusiveReq *state* = HomeIip
	cont:MsgExclusiveReq	send MsgExclusiveAckData insert pointer *state* = HomeInvalid retry queued messages
	MsgModifyReq	send MsgWInvalidate to remote copy *tx_cont* = cont:MsgModifyReq *state* = HomeIip
	cont:MsgModifyReq	send MsgModifyAckData insert pointer *state* = HomeInvalid retry queued messages
	MsgFlush, MsgFlushData	delete pointer *state* = HomeExclusive
	MsgRelease	do nothing

Table A-13. HomeInvalid: protocol events and actions.

A.3 Remote-Side State Machine

Figures A-9 through A-16 show the state transition diagrams for the eight remote-side protocol states. These figures are similar to those shown for the home-side state machine (Figures A-1 through A-8), with two minor differences. First, because "continuations" are not employed on the remote side (as is discussed further in Section A.5), none of the remote-side state transition diagrams include dashed arrows. Second, because the remote side of the CRL protocol only employs a limited form of message queuing (setting a flag when an invalidation message was received at an inconvenient time; see Section A.9 for details), the Queue box is instead labeled Set rcvd_inv flag.

As was the case in Figures A-1 through A-8 (for the home-side state machine), Figures A-9 through A-16 show only the state transitions that occur in response to protocol events. A more complete description of the remote-side state machine (in the form of psuedocode) can be found in Tables A-14 through A-21.

Each of Tables A-14 through A-21 consists of three columns. The first and second columns contain the names of the relevant protocol state and event types, respectively. The third column contains pseudocode for the actions that should be taken when the corresponding event occurs in the corresponding state.

Beyond the protocol state, two other components of the remote-side protocol metadata associated with each region are referenced in Tables A-14 through A-21. These components are summarized below:

read_cnt: This field is used to count the number of local read operations in progress (simultaneously) for the associated region.

rcvd_inv: This field is used to "buffer" an invalidation message that cannot be processed immediately upon reception because an operation is in progress on the corresponding region.

Newly allocated remote copies of regions (caused by calls to `rgn_map` that cannot be satisfied locally) start in the RemoteInvalid state.

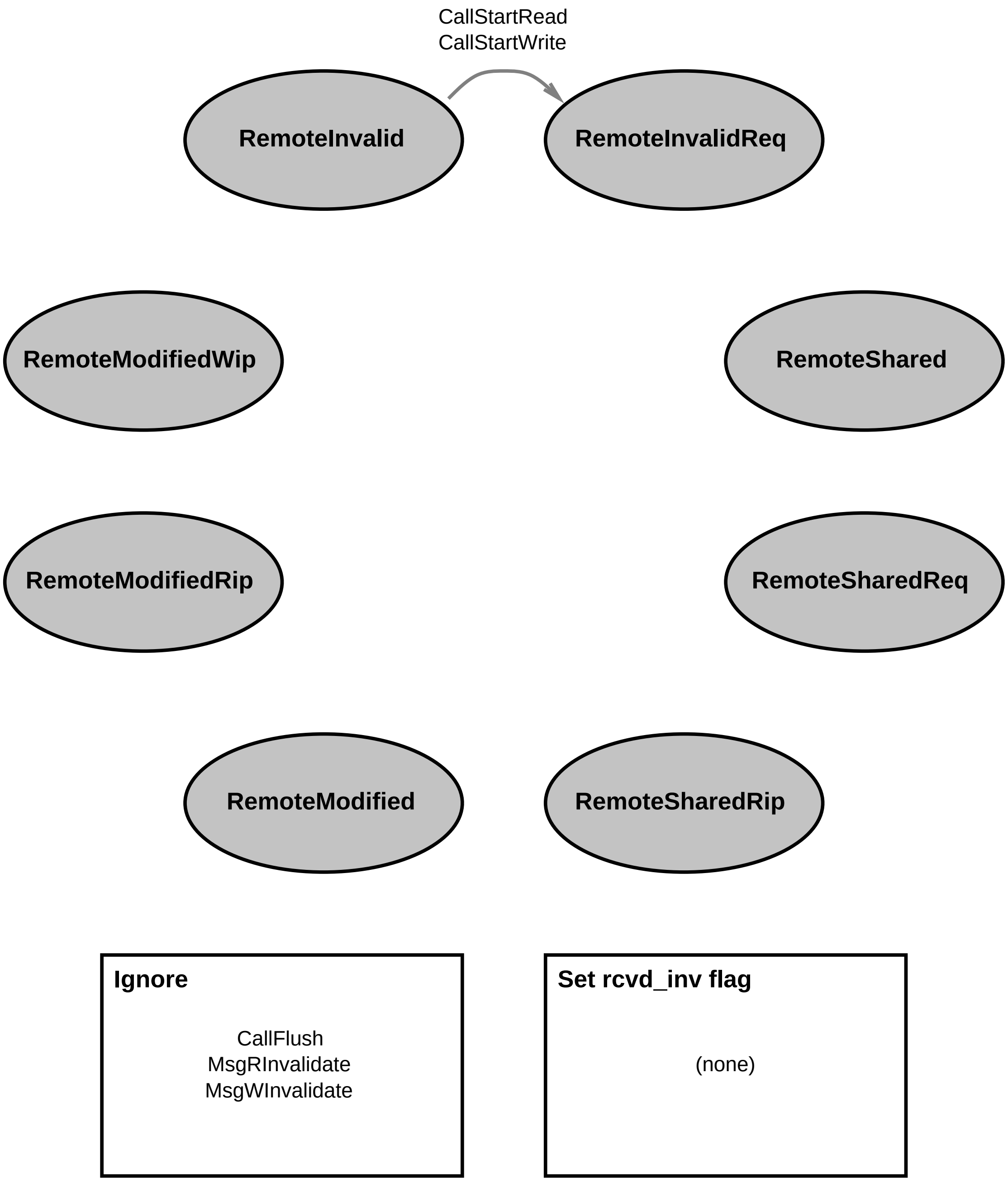


Figure A-9. RemoteInvalid: state transition diagram.

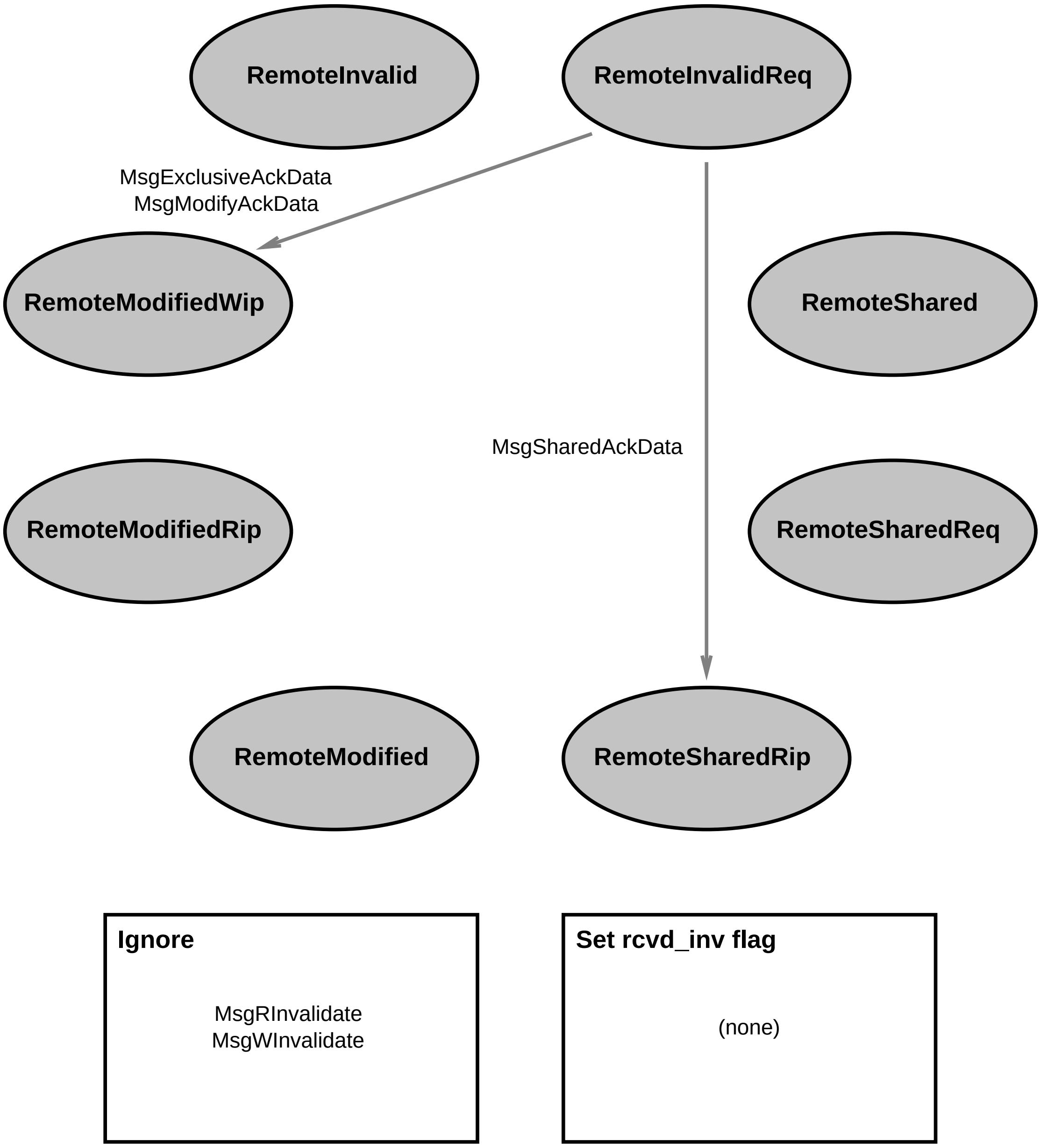


Figure A-10. RemoteInvalidReq: state transition diagram.

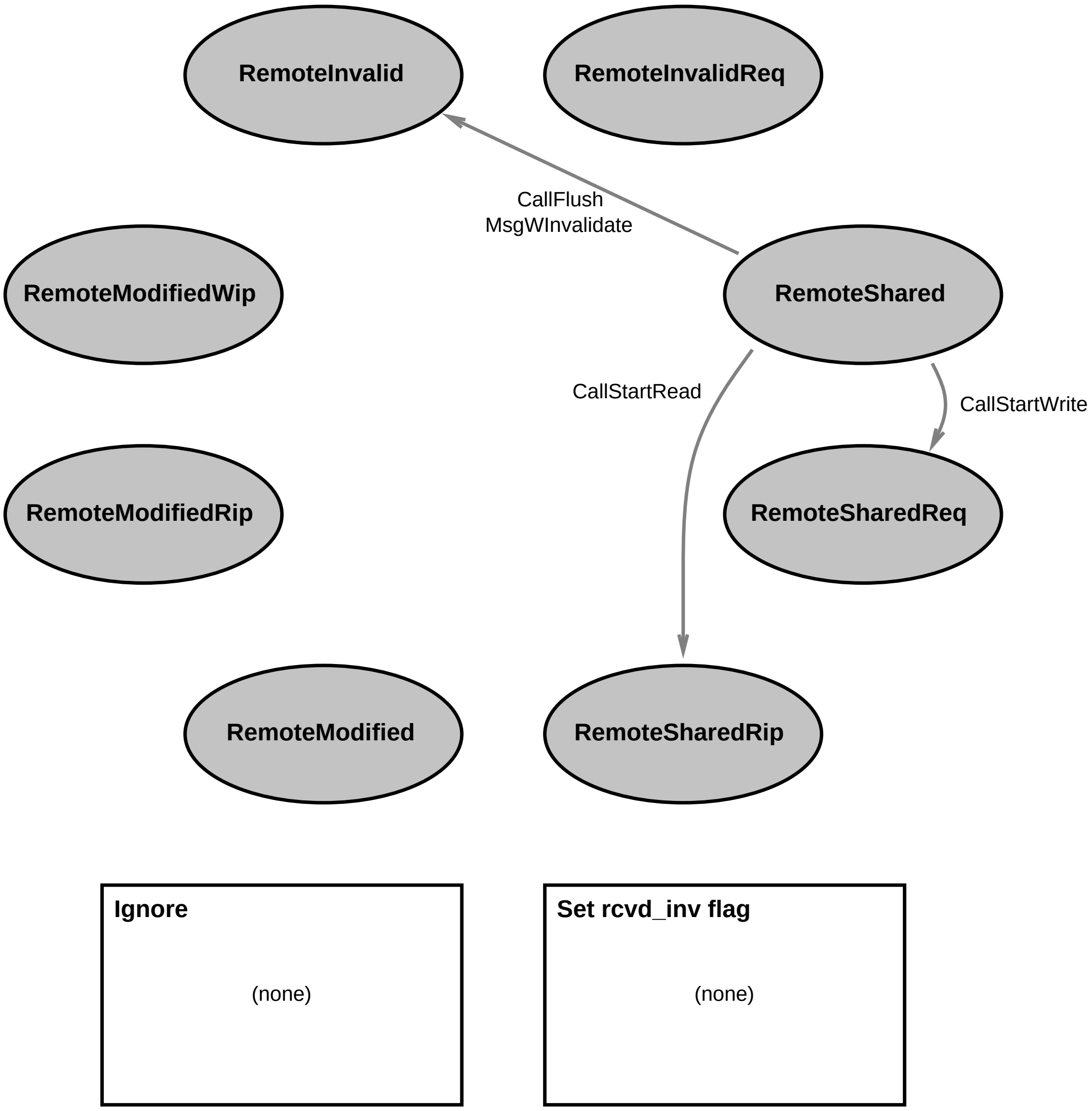


Figure A-11. RemoteShared: state transition diagram.

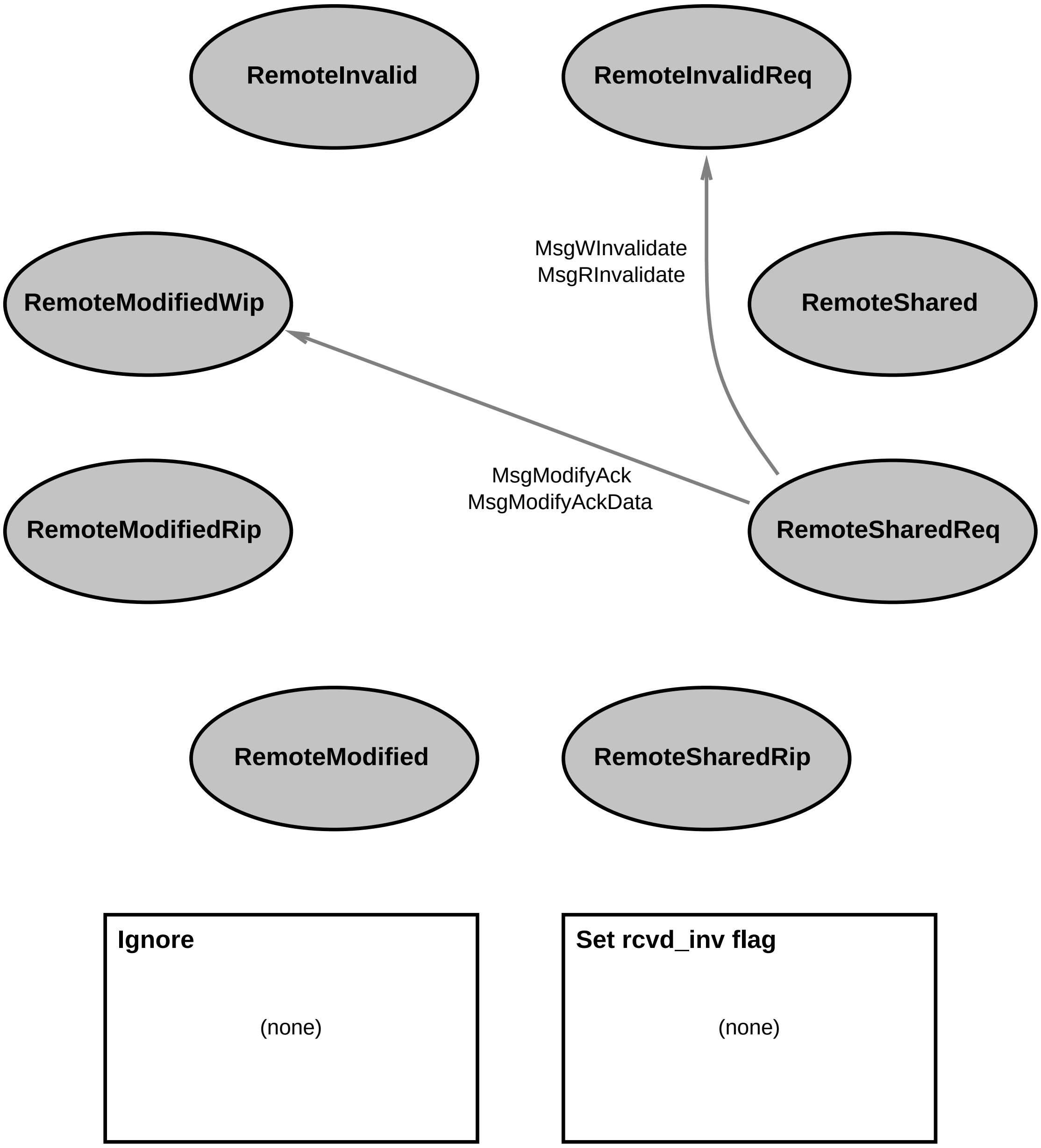


Figure A-12. RemoteSharedReq: state transition diagram.

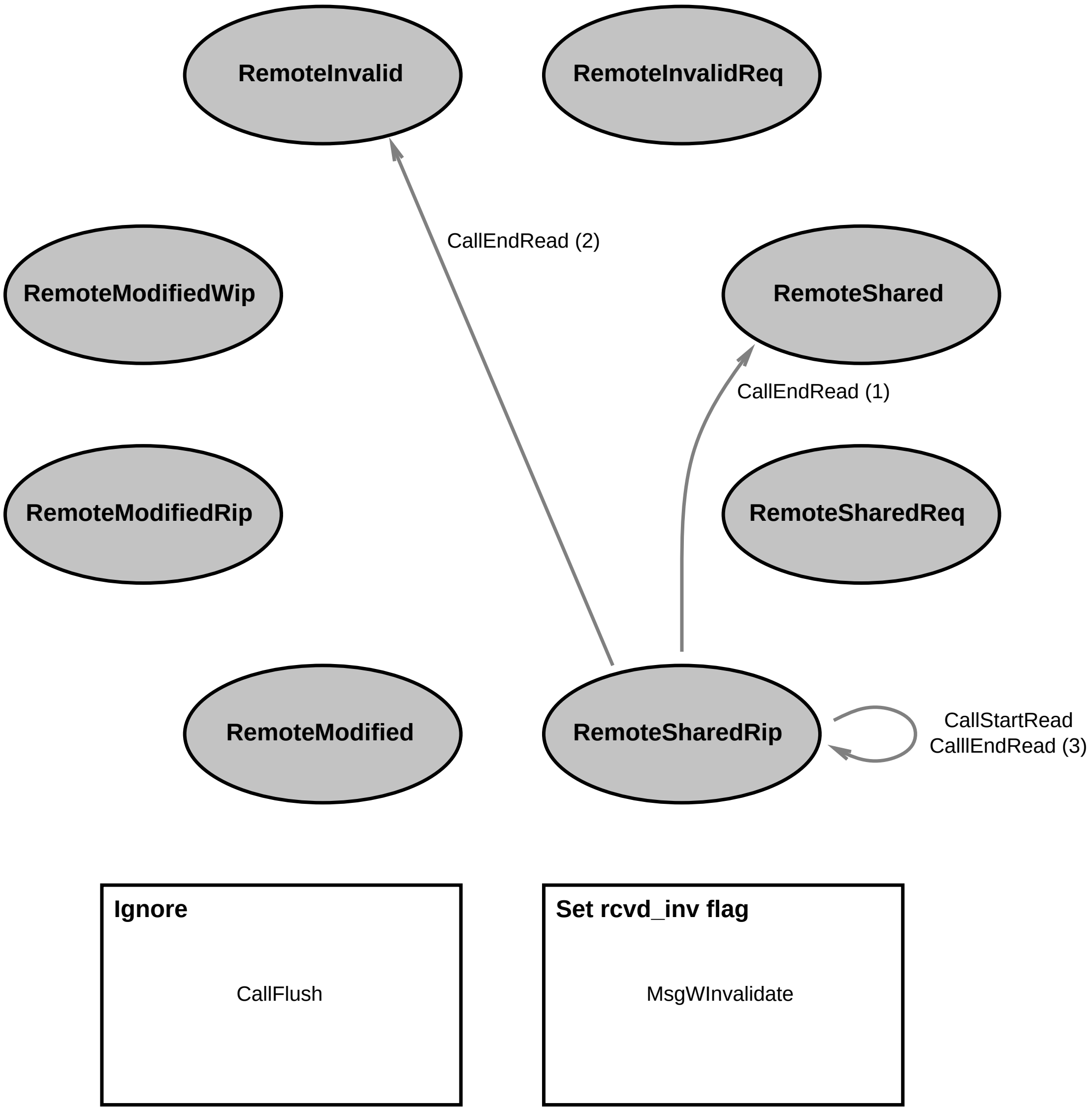


Figure A-13. RemoteSharedRip: state transition diagram.

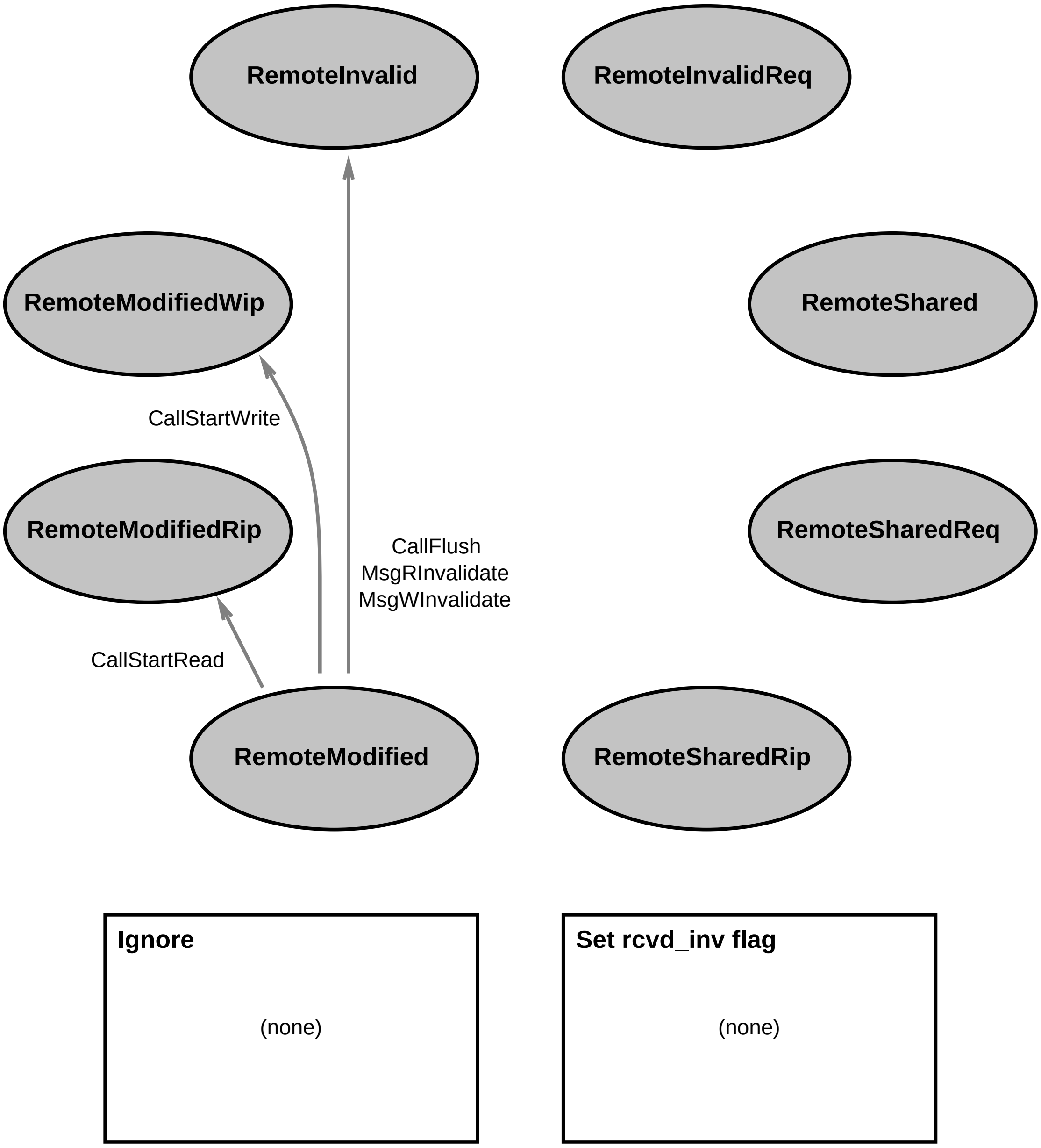


Figure A-14. RemoteModified: state transition diagram.

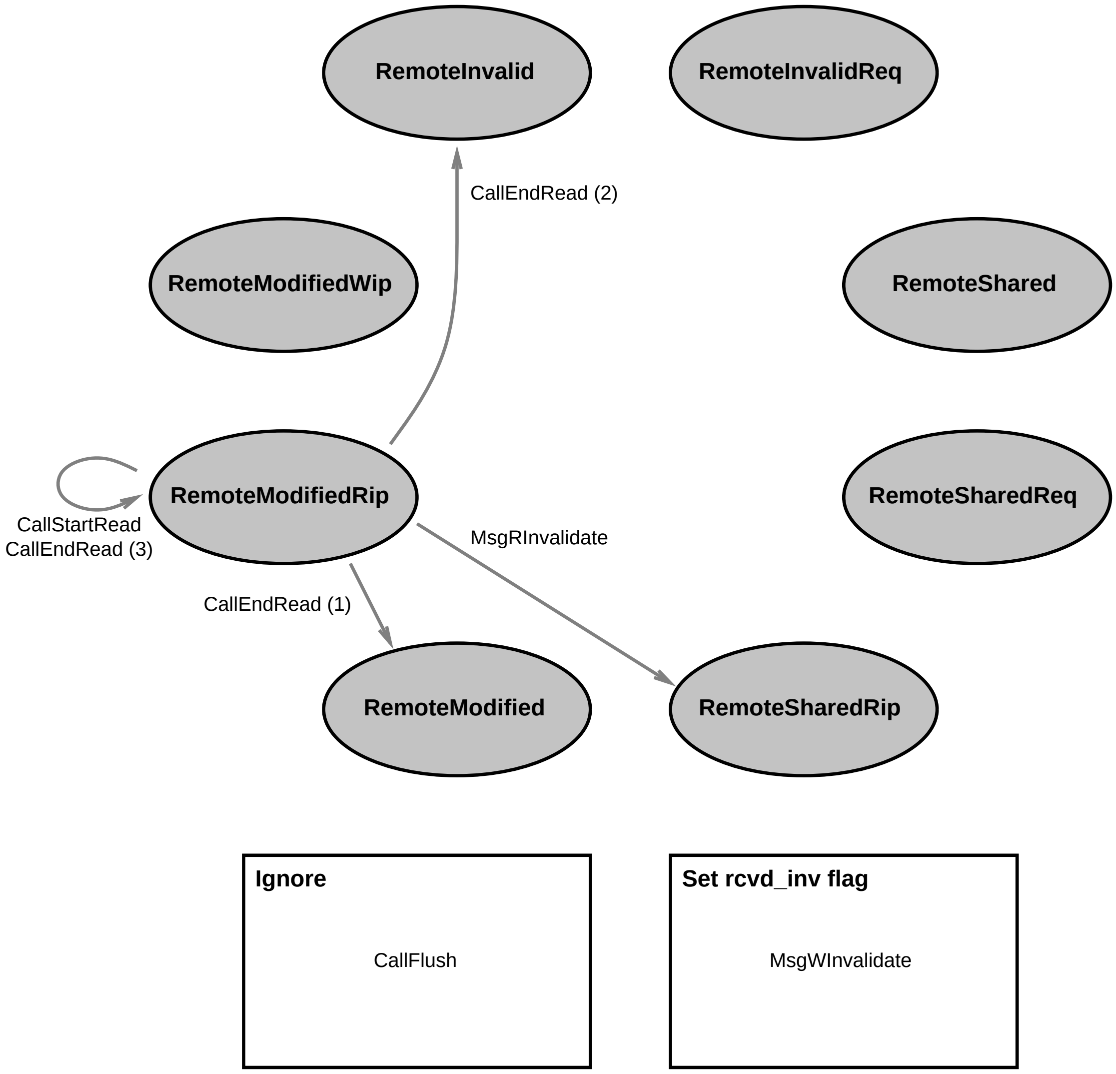


Figure A-15. RemoteModifiedRip: state transition diagram.

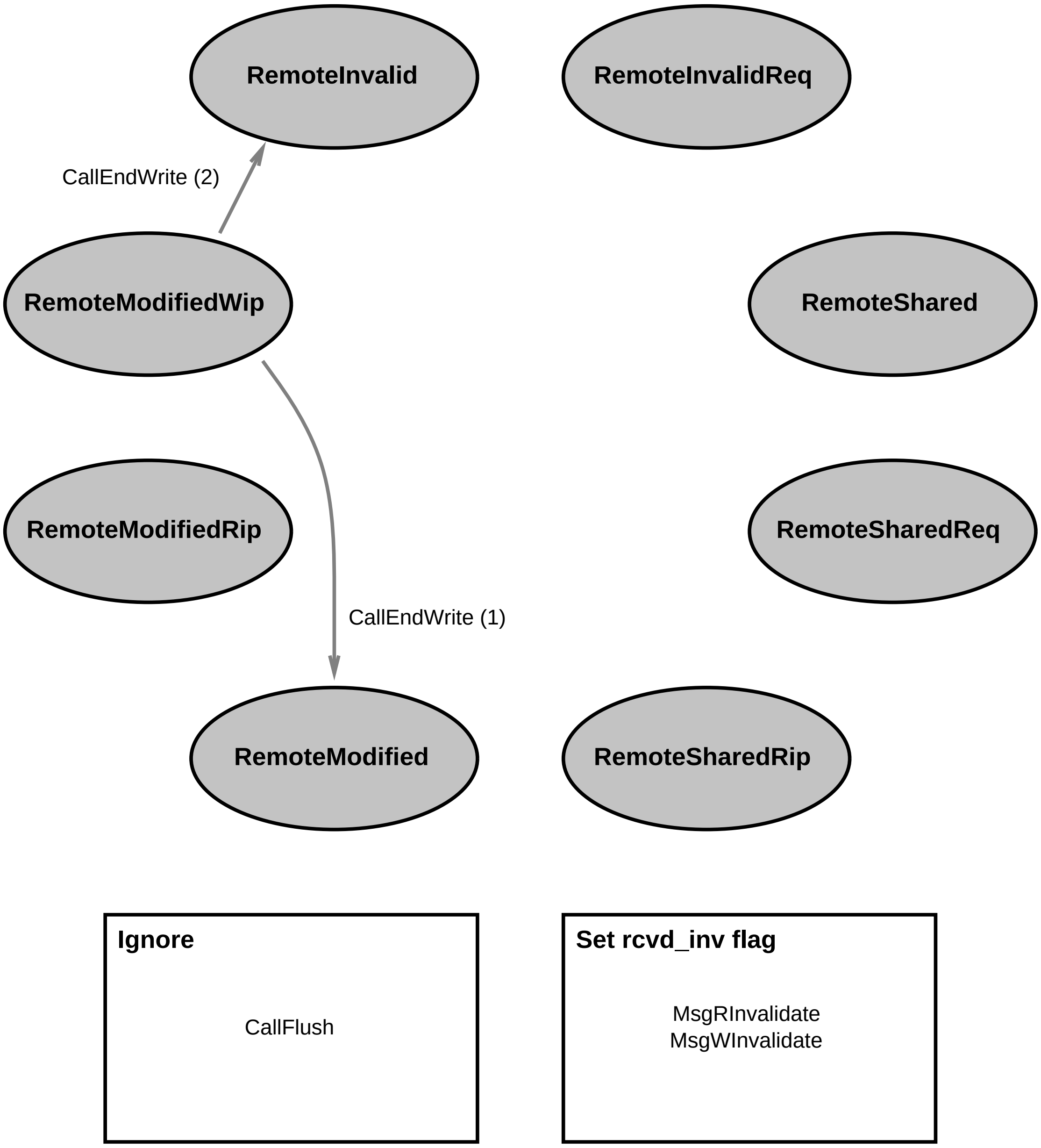


Figure A-16. RemoteModifiedWip: state transition diagram.

State	Event	Actions
RemoteInvalid	CallStartRead	send MsgSharedReq to home *state* = RemoteInvalidReq poll until reply is received
	CallStartWrite	send MsgExclusiveReq to home *state* = RemoteInvalidReq poll until reply is received
	CallFlush	do nothing
	MsgRInvalidate, MsgWInvalidate	do nothing

Table A-14. RemoteInvalid: protocol events and actions.

State	Event	Actions
RemoteInvalidReq	MsgSharedAckData	*read_cnt* = 1 *state* = RemoteSharedRip
	MsgExclusiveAckData, MsgModifyAckData	*state* = RemoteModifiedWip
	MsgRInvalidate, MsgWInvalidate	do nothing

Table A-15. RemoteInvalidReq: protocol events and actions.

State	Event	Actions
RemoteShared	CallStartRead	*read_cnt* = 1 *state* = RemoteSharedRip
	CallStartWrite	send MsgModifyReq to home *state* = RemoteSharedReq poll until reply is received
	CallFlush	send MsgFlush to home *state* = RemoteInvalid
	MsgWInvalidate	send MsgInvalidateAck to home *state* = RemoteInvalid

Table A-16. RemoteShared: protocol events and actions.

State	Event	Actions
RemoteSharedReq	MsgWInvalidate, MsgRInvalidate	send MsgInvalidateAck to home *state* = RemoteInvalidReq
	MsgModifyAck, MsgModifyAckData	*state* = RemoteModifiedWip

Table A-17. RemoteSharedReq: protocol events and actions.

State	Event	Actions
RemoteSharedRip	CallStartRead	*read_cnt* += 1

	CallEndRead	*read_cnt* -= 1
		if (*read_cnt* == 0)
		    if (*rcvd_inv* == 0)
		        state = RemoteShared
		    else
		        send MsgInvalidateAck to home
		        rcvd_inv = 0
		        state = RemoteInvalid

	CallFlush	do nothing
	MsgWInvalidate	*rcvd_inv* = 1

Table A-18. RemoteSharedRip: protocol events and actions.

State	Event	Actions
RemoteModified	CallStartRead	*read_cnt* = 1 *state* = RemoteModifiedRip
	CallStartWrite	*state* = RemoteModifiedWip
	CallFlush	send MsgFlushData to home *state* = RemoteInvalid
	MsgRInvalidate, MsgWInvalidate	send MsgInvalidateAckData to home *state* = RemoteInvalid

Table A-19. RemoteModified: protocol events and actions.

State	Event	Actions
RemoteModifiedRip	CallStartRead	*read_cnt* += 1

	CallEndRead	*read_cnt* -= 1
		if (*read_cnt* == 0)
		  if (*rcvd_inv* == 0)
		    state = RemoteModified
		  else
		    send MsgInvalidateAckData to home
		    rcvd_inv = 0
		    state = RemoteInvalid

	CallFlush	do nothing
	MsgRInvalidate	send MsgRelease to home *state* = RemoteSharedRip
	MsgWInvalidate	*rcvd_inv* = 1

Table A-20. RemoteModifiedRip: protocol events and actions.

State	Event	Actions

RemoteModifiedWip	CallEndWrite	if (*rcvd_inv* == 0)
		  state = RemoteModified
		else
		  send MsgInvalidateAckData to home
		  rcvd_inv = 0
		  state = RemoteInvalid

	CallFlush	do nothing
	MsgRInvalidate, MsgWInvalidate	*rcvd_inv* = 1

Table A-21. RemoteModifiedWip: protocol events and actions.

A.4 Continuations and Home-Side 'Iip' States

Figures A-4 and A-8 contain dashed arrows that indicate "continuations." In the context of the CRL coherence protocol, a continuation is the second phase of a two-phase set of protocol actions (*e.g.,* one in which some number of invalidation messages are sent during the first phase, but the second phase cannot be executed until all invalidations have been acknowledged).

Each continuation is implemented as a separate procedure. Protocol handlers that implement the first phase (*e.g.,* sending out invalidation messages) of a two-phase set of protocol actions are responsible for storing a pointer to an appropriate continuation in the metadata area before effecting a state transition into an "invalidation in progress" ('Iip') state. In turn, the 'Iip' protocol handlers are responsible for collecting invalidation acknowledgements and invoking the stored continuation after all invalidation messages have been acknowledged. Thus, the dashed arrows in Figures A-4 and A-8 represent the transitions out of the 'Iip' states. Because these "continuation state transitions" are part of a two-phase set of protocol actions, they are shown in these figures instead of those for the 'Iip' states.

In the state transition diagram for the HomeIip state (Figures A-6), only a single (loop-back) state transition arrow is shown. This arrow indicates the action taken in response to all invalidation acknowledgement messages (and the like) before all acknowledgements have been received. Upon receiving the last acknowledgement, the stored continuation is invoked and the appropriate "continuation state transition" is caused (per the dashed arrows in Figures A-4 and A-8). No state transition arrows are shown for the CallStartRead and CallStartWrite events; when one of these events occurs in the HomeIip state, the calling procedure is responsible for spinning (and polling for incoming messages) until the state changes, at which time the appropriate call event is retried.

In the state transition diagram for the HomeIipSpecial state (Figures A-7), no state transition arrows are shown. This is because the HomeIipSpecial state can only be entered in cases where a single invalidation message has been sent, so the stored continuation will be invoked immediately upon receipt of the first invalidation acknowledgement message. Once again, no state transition arrows are shown for the CallStartRead and CallStartWrite events; these events are handled the same way in the HomeIipSpecial state that the are in the HomeIip state.

Strictly speaking, the use of continuations is not necessary. They could be eliminated by introducing a set of new, specialized 'Iip' states, one for each continuation. Except for the actions taken after the last invalidation message is acknowledged, each new 'Iip' state would look essentially identical to the others. Eliminating continuations in this manner may yield a slight performance improvement by eliminating an indirect jump (procedure call to a PC loaded from memory) from the critical path of two-phase protocol transitions. However, for the sake of simplicity, the current CRL implementation does not implement this optimization.

A.5 Remote-Side 'Req' States

The "continuation" mechanism used to implement home-side two-phase state transitions is not used on the remote side of the protocol. Because the only two-phase transitions on the remote side are those that occur because of a call event on the local node (*e.g.,* a CallStartRead or CallStartWrite that cannot be satisfied locally, so a request must be sent to the home node, and the desired state transition does not occur until after the corresponding reply has been received), a simpler scheme is employed: processors invoking call events that require waiting for a reply message spin, polling for incoming protocol messages until the desired reply has been received.

Since the current CRL implementation only supports single-threaded applications in which a single user thread or process runs on each processor in the system, this type of blocking/spinning mechanism is acceptable. In an implementation that supports multiple user threads per processor in the system, more sophisticated techniques that involve descheduling the requesting thread until the reply has been received may be necessary. Competitive schemes [36, 54] in which requesting threads poll for some period of time in hope of receiving a quick response before being descheduled may also be useful in multithread implementations.

A.6 Protocol Message Format

Protocol messages that do not include a copy of the region data ("non-data-carrying protocol messages") are encoded in the standard five-word format (32-bit words) shown in Figure A-17 (small enough to fit in a single active message on the CM-5). Protocol messages that include a copy of the region data ("data-carrying protocol messages") include the same information, but the exact format may depend on the particulars of bulk data transport mechanism available on different platforms.

Since protocol messages are implemented using active messages, the first word of each protocol message is the program counter (PC) of the procedure that should be invoked upon message delivery. As discussed in Section A.8, most non-data-carrying protocol messages are received using a common handler (`rgn_msg_stub`), but a specialized handler (`rgn_inv_stub`) is used for invalidation messages.

The second word of each protocol message is the protocol message type, a small integer indicating one of the 14 possible types of protocol messages shown in Tables A-4 and A-5. The third word of each protocol message is either the starting address of the metadata area for the region on the destination node or the region identifier for the region (see Section A.8). The fourth word of each protocol message is the source node number so the destination node knows which node sent the message; active message communication models typically do not provide out-of-band mechanisms for obtaining such information. Finally, the fifth word of each protocol message contains either a version number for the

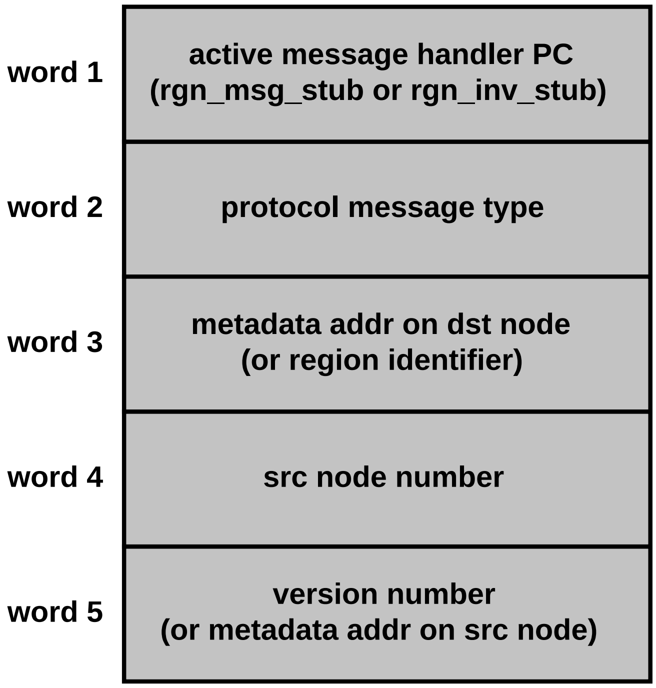


Figure A-17. Standard (non-data-carrying) protocol message format.

region (used to resolve out-of-order message delivery issues; see Section A.11) or the starting address of the metadata area on the sending node (see Section A.8).

A slightly more compact encoding (four words instead of five) for protocol messages could be obtained by either (1) packing both the protocol message type and the source node number into a single word or (2) having specialized active message handlers for each possible type of protocol message, and thus encode the message type in the active message handler PC. No pressing need for a more compact encoding has arisen, however, so the current CRL implementation retains the five-word encoding shown in Figure A-17.

A.7 Atomicity

In order to allow normal threads or processes running on a particular node to achieve atomicity with respect to handlers for incoming active messages, both the CM-5 and Alewife platforms allow message delivery to be temporarily disabled (on the CM-5, this is colloquially known as "disabling interrupts"). On both platforms, however, disabling message delivery for prolonged periods can lead to increased congestion in the interconnection network, which in turn can cause severe performance degradation. Furthermore, on some platforms (*e.g.,* the CM-5), the cost of disabling and reenabling message delivery can be prohibitively high (*e.g.,* 10 microseconds).

To address this issue, the prototype CRL implementation employs a special software-based "incoming message queue" and an associated enabled/disabled flag on each node; these are used to implement an application-specific form of *optimistic active messages* [82]. When the incoming message queue is enabled, the "control" portions of incoming protocol messages (those elements shown in Figure A-17) are placed on this queue for later processing instead of being acted upon immediately. Thus, non-data-carrying protocol messages are enqueued in their entirety, but only the "control" (non-data) portions of data-carrying protocol messages are queued. The data component of data-carrying protocol messages is always deposited directly in the target region's user data area, independent of the state of the incoming message queue.

When a protocol message arrives and the incoming message queue is disabled, the queue is enabled, the appropriate protocol handler is executed, any messages on the incoming queue are processed, and then the queue is disabled again. Enabling the incoming message queue during protocol handler execution is necessary to provide atomicity with respect to other incoming protocol messages (*e.g.,* on the CM-5, interrupts are disabled during active message handlers, but incoming active message handlers can still be invoked if an active message handler attempts to send another active message).

Given this mechanism, a thread or process can easily achieve atomicity with respect to incoming protocol messages by simply enabling the incoming message queue. After the code requiring atomicity is complete, the thread or process is responsible for processing any messages on the incoming message queue and then disabling the queue (as was the case for protocol handlers).

Care must be taken when draining messages from the incoming message queue and disabling further queuing that no messages remain in the queue after the process is complete. A straightforward implementation might require being able to atomically check that no messages remain in the incoming queue and, if so, disabling the queuing of future messages. Because achieving such atomicity (by disabling and reenabling interrupts) is relatively expensive on the CM-5, the CM-5 implementation of CRL employs a more complex scheme that only requires message delivery to be disabled if the incoming queue is non-empty. Thus, in cases where atomicity was only required for a brief period of time and no protocol messages were queued, the CM-5 implementation avoids the cost of disabling and reenabling interrupts.

By using an incoming message queue on each node, the prototype CRL implementation can efficiently provide atomicity with respect to incoming message delivery while avoiding the potentially costly solutions that involve disabling incoming message delivery for prolonged periods of time.

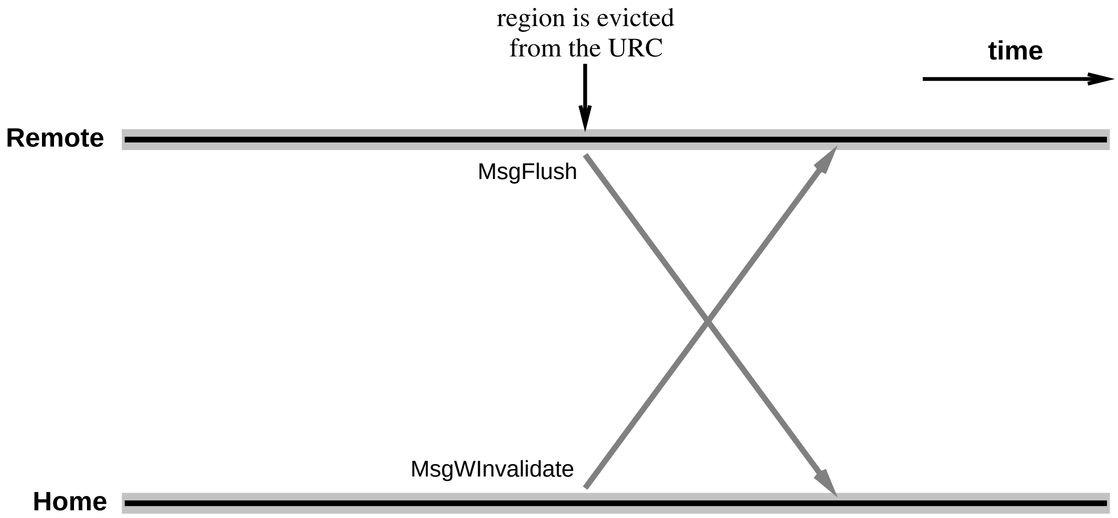


Figure A-18. The "flush-invalidation" problem.

A.8 Naming Regions

To streamline the handling of protocol messages, most protocol messages "name" the target region by the base address of the metadata area on the destination node. Remote regions dedicate a field in the metadata area for the address of the corresponding home region metadata area (on the home node); home regions maintain this information on a per-remote-copy basis using a field in the directory entry associated with each remote copy. Remote nodes obtain the address of the home region metadata area at region mapping time; it is included in the auxiliary information obtained from the home node when a mapping is not already present in the region table (as discussed in Section 4.4). In order to make similar information about remote nodes available to the home node, all request messages (MsgSharedReq, MsgExclusiveReq, and MsgModifyReq) include the starting address of the remote region's metadata area in place of the version number (see Figure A-17).

Because unmapped regions can be evicted from the unmapped region cache (and thus the memory that was allocated for them reclaimed), care must be taken to ensure protocol messages sent by the home node do not contain information that will cause a protocol handler to incorrectly reference memory that has been reclaimed. Of the six kinds of home-to-remote protocol messages (see Table A-4), only the invalidation messages (MsgRInvalidate and MsgWInvalidate) have the potential to cause this type of problem. As is illustrated in Figure A-18, a region could be evicted from a remote URC, causing a MsgFlush to be sent to the home node, but the home node could send an invalidation message before the MsgFlush arrives and is processed. In such a scenario, the appropriate course of action is to ignore the invalidation message on the remote node, but the region metadata that might have been used to determine this fact would have already been reclaimed.

To avoid the problem of metadata disappearing before a message can be processed, invalidation messages in CRL (both MsgWInvalidate and MsgRInvalidate) do not name the target region by the address of its metadata on the destination node, as is done with other protocol messages. Instead, invalidation messages use region identifiers to name their target regions. Further, instead of using the message handler used by all other protocol messages (`rgn_msg_stub`), invalidation messages use a special message handler (`rgn_inv_stub`). This special handler is responsible for translating the region identifier contained in the invalidation message into the address of the target region's metadata area before proceeding. If a valid translation for the region identifier cannot be found in the region table, it is assumed that the target region was evicted from the URC and the invalidation message is (correctly) ignored.

A.9 Unexpected Messages

When protocol messages show up at times when they would be inconvenient or difficult to handle, they are queued for later processing. Protocol handlers that cause transitions out of a state in which such messages might have been queued are responsible for processing as many messages as possible from the head of the queue after entering the new state.

On the home side of the protocol, protocol messages that cannot be processed immediately are placed in a "blocked message queue" associated with every home region (a singly-linked queue of blocked messages is constructed using a queue head and tail pointer in the home-side region metadata). Only request messages (MsgSharedReq, MsgExclusiveReq, and MsgModifyReq) can ever get queued; this occurs if a request message arrives that conflicts with an operation that is already in progress at the home node (*i.e.,* MsgExclusiveReq and MsgModifyReq conflict with a read operation; any request conflicts with a write operation) or an invalidation is currently in progress for the region (*i.e.,* the home node is the HomeIip state or HomeIipSpecial state). Because the current CRL implementation only allows each processor in a system to have a single request in flight, the maximum number of request messages that could be queued in any one blocked message queue is one less than the total number of processors in the system.

On the remote side of the protocol, the only protocol messages that cannot always be handled immediately are invalidation messages (MsgWInvalidate and MsgRInvalidate). In general, this occurs whenever an invalidation message arrives at a remote node where one or more operations on the target region are already in progress. In such a situation, the effect of the invalidation message must be delayed until all operations have been terminated. Message queuing on the remote side of the protocol can be greatly simplified by taking advantage of the following observation: On any given remote region, at most one invalidation message will ever need to be queued for later processing. Thus, unlike the home side of the protocol (where the full generality of a queue is needed to hold a potentially large number of messages), the "queue" on the remote side can be implemented with a single flag in the region metadata that indicates whether or not an invalidation message has been received but not acted upon yet. (In Figures A-9 through A-16 and Tables A-14 through A-21, this flag is referred to as the *rcvd_inv* flag.)

In hardware-based systems, inconvenient or unexpected protocol messages are often handled by sending a negative acknowledgement (nack) back to the sender. Upon being nack-ed, the original sender of a protocol message is responsible for resending it. Because the overhead of receiving an active message can be significant, even in the most efficient of systems, employing such an approach in CRL could raise the possibility of livelock situations in which a large number of remote nodes could "gang up" on a home node, saturating it with requests (that always get nack-ed and thus resent) in such a way that forward progress is impeded indefinitely. Other solutions to this problem are possible (*e.g.,* nack inconvenient requests, but use a backoff strategy when resending nack-ed messages), but they have not been investigated.

A.10 Two Kinds of Invalidation Messages

In general, invalidation messages that arrive at a remote region when an operation is in progress simply cause the *rcvd_inv* flag in the remote region's metadata area to be set, as described above. In one situation, however, applying this policy can lead to significant performance degradation by unnecessarily throttling concurrency. The particular situation in which this might happen is as follows: A region is in the RemoteModifiedRip state on a remote node (perhaps because that node had a write operation in progress, then initiated a read operation on the region immediately after terminating the write operation), while many other nodes are attempting to initiate read operations on the same region (as might be the case if the region were being used to broadcast information to a collection of other nodes). According to the CRL programming model (Section 3.2), the new read operations should be allowed to proceed concurrently with the existing read operation.

In such a situation, the remote node with the operation in progress has a dirty (modified) copy of the region data, so the home node is in the HomeInvalid state. Thus, when the first MsgSharedReq message arrived at the home node (or the application code running on the home node invoked a CallStartRead), an invalidation message would be sent to the remote node with the read operation in progress and the home node would transition into the HomeIip state; subsequent MsgSharedReq messages arriving at the home node would be placed in the blocked message queue for later processing. When the invalidation message arrives at the remote node, the intended effect is for the dirty data to be written back to the home node in order to allow the pending read requests to be satisfied. However, since a read operation is in progress on the remote node receiving the invalidation message, this intended effect would not happen if the invalidation message only caused the *rcvd_inv* flag to be set.

To address this difficulty, the CRL coherence protocol employs two kinds of invalidation messages: MsgWInvalidate messages that are sent in response to requests for an exclusive copy of the region data (*i.e.,* for a write operation), and MsgRInvalidate

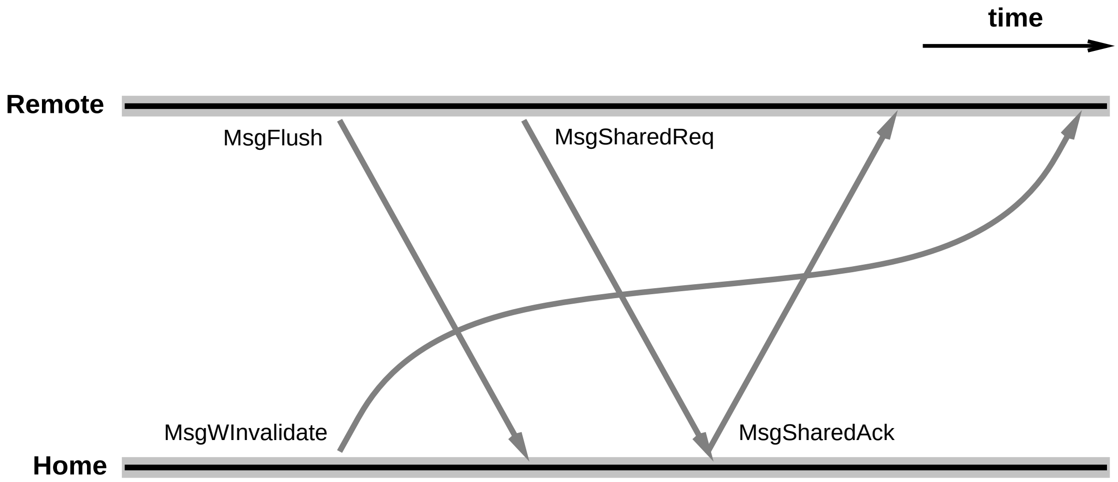


Figure A-19. The "late invalidation" problem.

messages that are sent in response to requests for a shared copy (*i.e.,* for a read operation). In all remote-side protocol states except for RemoteModifiedRip, the same protocol handlers are used for both kinds of invalidation messages. In the RemoteModifiedRip state, MsgWInvalidate messages are handled in the usual way (causing the *rcvd_inv* flag to be set). In contrast, the protocol handler for MsgRInvalidate messages causes (1) a MsgRelease message to be returned to the home node with a copy of the region data and (2) a transition into the RemoteSharedRip state (see Figure A-15 and Table A-20). The MsgRelease message serves two purposes: first, to write back the modified region data to the home node; and second, to inform the home node that the remote node no longer has an exclusive copy of the region but did retain a shared copy, thus allowing the other read operations to proceed concurrently.

A.11 Out-of-Order Message Delivery

In order to handle out-of-order message delivery, a common occurrence when programming with active messages on the CM-5, CRL maintains a 32-bit version number for each region. Each time a remote processor requests a copy of the region, the current version number is recorded in the directory entry allocated for the copy and returned along with the reply message; the current version number is then incremented. By including the version number for a remote copy of a region in all other protocol messages related to that copy, misordered protocol messages can be easily identified and either buffered or dropped, as appropriate. Figures A-19 through A-22 show examples of the four types of protocol message reordering that can occur.

Figure A-19 shows an example of the "late invalidation" problem. In this situation, the delivery of an invalidation message is delayed long enough for the target remote node to (1) drop the copy of the data that was the intended target of the invalidation message and

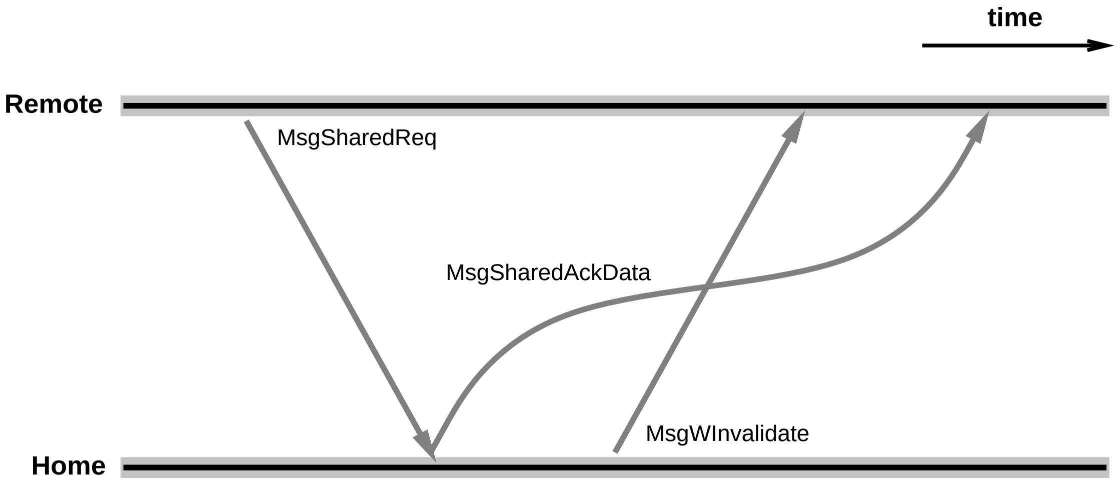


Figure A-20. The "early invalidation" problem.

(2) obtain a new copy of the data from the home node; care must be taken to ensure that the new copy of the data is not invalidated because of the late invalidation message. This class of reordering problem is solved by comparing the version number included in the invalidation message with that of the remote copy: invalidation messages that contain a version number earlier than that stored in the remote region's metadata are late and should thus be ignored.

Figure A-20 shows an example of the "early invalidation" problem. In this situation, the delivery of a reply to a request for a copy of the data is delayed long enough for a message intended to invalidate that copy of the data to arrive first; care must be taken to ensure the invalidation message is buffered until appropriate action can be taken. Like the late invalidation problem, this class of reordering problem is also solved by comparing the region version number of included in the invalidation message with that of the remote copy: invalidation messages that contain a version number later than that stored in the remote region's metadata are early, so the *rcvd_inv* flag is set. When the reply message arrives, the state of the remote region is changed to an appropriate operation-in-progress state (independent of the value of the *rcvd_inv* flag), so the net effect is as if the invalidation message did not show up until after the reply.

Figure A-21 shows an example of the "late release" problem. In this situation, the delivery of a MsgRelease message dropping exclusive ownership of a region is delayed long enough for a MsgFlush message for the same region to arrive at the home node first; care must be taken to ensure that any release messages that arrive late have no effect on the protocol state. This class of problems is addressed in two ways. First, instead of using the HomeIip state to collect invalidation message acknowledgements, protocol handlers that send MsgRInvalidate messages (in response to which MsgRelease responses might be sent) cause a transition in the HomeIipSpecial state. MsgRelease messages are only acted upon when they arrive at a region in the HomeIipSpecial state; in all other protocol states, MsgRelease messages are dropped without effect. Second, when a MsgRelease

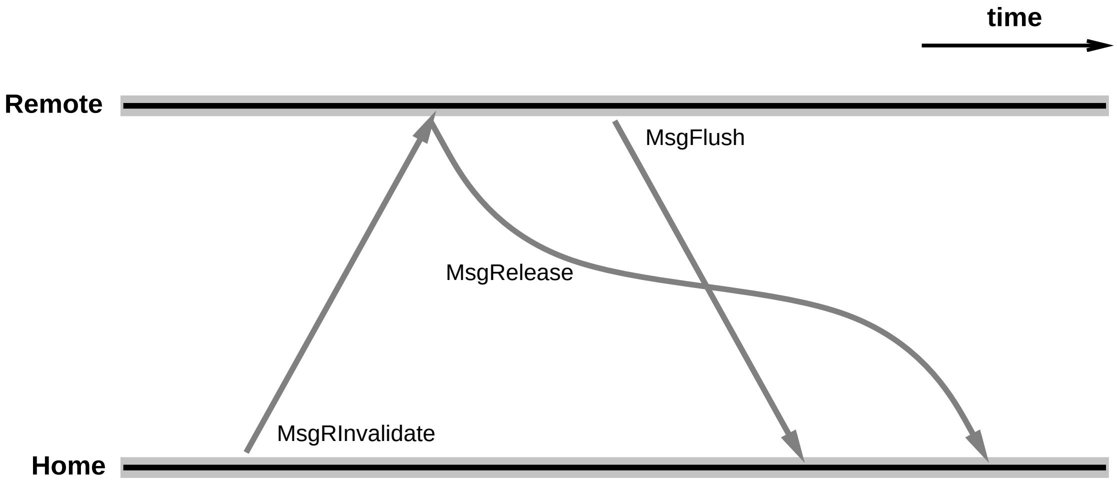


Figure A-21. The "late release" problem.

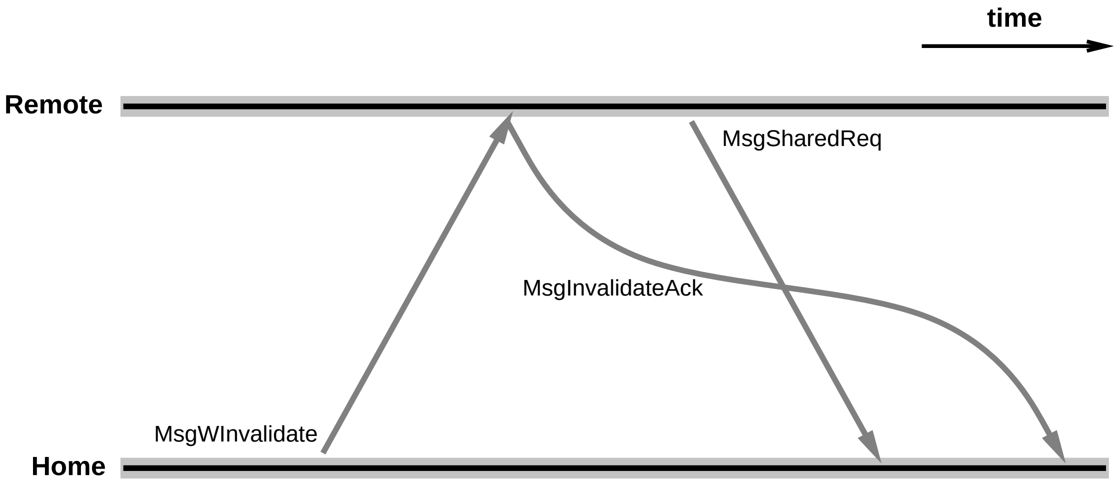


Figure A-22. The "late invalidation acknowledgement" problem.

message does arrive at a region in the HomeIipSpecial state, it is ignored unless the version number (of region on the sending node) matches that stored in the directory entry. The combination of these policies ensures that late MsgRelease messages—those that arrive in states other than HomeIipSpecial or with an incorrect version number—are correctly ignored.

Finally, Figure A-22 shows an example of the "late invalidation acknowledgement" problem. This problem occurs when the delivery of a MsgInvalidateAck or MsgInvalidateAckData message is delayed long enough that a request message for a new copy of the region (sent by the same node that sent the invalidation acknowledgement message) arrives at the home node first. This situation is handled by only allowing each home copy of a region to have at most one directory entry per remote node. If a protocol handler attempts to insert a second directory entry for a particular remote node, the protocol handler is aborted and the invoking protocol message is queued for later processing.

Another approach to dealing with the “late invalidation acknowledgement” problem might be to, in essence, do nothing, and simply allow a home region’s directory to (at least temporarily) contain multiple entries for a remote node (each of which corresponds to a different “copy” of the region). Since this type of problem is not expected to occur too frequently, however, the current CRL implementation retains the more conservative (and somewhat simpler to reason about) approach of allowing at most one directory entry per remote region.

Appendix B

Raw Data

The tables in this appendix contain the raw data used to produce the summary figures and tables presented in Chapters 5 and 6.

B.1 Interrupts vs. Polling

This section presents the raw performance data obtained using the synthetic workload described in Section 5.1.2. Table B-1 shows the time per outer loop iteration (in microseconds) using two-message communication events for a wide range of polling rates (determined by the amount of useful work per poll) and computation-to-communication ratios (determined by the amount of useful work per communication event) and each of of the 'null', 'intr', 'none', and 'poll' cases. Tables B-2 and B-3 present the same data for three- and four-message communication events. All values are averages measured over 10,000 outer loop iterations.

Useful work per poll (cycles)		Useful work per communication event (cycles)							
		125	250	500	1000	2000	4000	8000	16000
15	null	4.3	8.2	16.1	31.8	63.1	125.7	250.6	500.9
	intr	40.4	46.4	56.9	76.2	112.1	178.2	305.0	557.5
	none	21.9	27.8	40.6	67.3	121.5	230.6	449.3	887.0
	poll	38.3	59.0	100.1	182.0	344.8	670.8	1322.1	2626.3
25	null	4.3	8.2	16.0	31.7	63.0	125.6	250.5	501.0
	intr	40.2	46.2	56.7	76.3	112.1	178.1	305.0	557.8
	none	22.0	27.7	40.5	67.1	121.5	230.2	450.0	886.1
	poll	31.5	45.8	73.7	129.2	239.9	461.6	904.3	1791.0
45	null	4.3	8.2	16.0	31.8	63.0	125.7	251.1	502.0
	intr	40.3	46.2	56.7	76.2	112.1	178.2	305.4	558.7
	none	22.0	27.8	40.4	68.0	122.0	231.3	450.0	886.2
	poll	27.1	36.8	56.1	94.0	169.6	320.3	622.0	1225.9
85	null	4.3	8.2	16.0	31.7	63.1	125.8	250.8	500.9
	intr	40.2	46.2	56.7	76.2	112.1	178.3	305.2	557.6
	none	21.9	27.7	40.6	67.2	121.6	231.6	449.2	888.8
	poll	24.6	31.8	45.8	73.6	128.6	238.4	457.3	895.3
165	null	4.3	8.3	16.1	31.9	63.5	126.2	251.8	503.1
	intr	40.2	46.2	56.7	76.4	112.3	178.7	306.1	559.7
	none	22.0	27.8	41.5	67.4	122.3	231.2	452.0	886.4
	poll	23.2	29.3	40.8	63.2	107.3	194.8	368.8	717.3
325	null	4.3	8.2	16.1	31.8	63.2	126.0	251.4	502.4
	intr	40.2	46.2	56.7	76.2	112.2	178.5	305.9	558.9
	none	22.0	27.7	40.6	68.0	121.5	230.7	448.5	887.9
	poll	22.5	28.1	38.5	58.3	96.8	172.5	323.0	623.4
645	null	4.3	8.2	16.0	31.7	62.9	125.6	250.6	500.9
	intr	40.0	46.2	56.6	76.1	112.0	178.1	305.1	557.5
	none	22.1	27.8	40.5	67.0	121.1	230.3	448.3	883.9
	poll	22.1	27.6	37.8	57.1	93.4	163.7	302.2	577.7
1285	null	4.3	8.2	16.0	31.7	63.0	125.6	250.7	501.1
	intr	40.2	46.2	56.6	76.2	112.1	178.1	305.2	557.9
	none	22.0	27.8	40.6	67.2	121.3	231.1	448.6	885.7
	poll	22.1	27.6	38.4	58.4	94.9	163.8	297.6	561.8
2565	null	4.3	8.3	16.0	31.6	62.9	125.4	250.5	500.4
	intr	40.2	46.2	56.6	76.2	112.0	178.0	304.9	557.2
	none	21.9	27.8	40.4	67.3	122.1	231.1	447.9	886.6
	poll	22.1	27.6	39.3	60.8	99.8	170.8	304.8	565.1
5125	null	4.3	8.2	16.0	31.7	63.2	125.7	251.1	502.1
	intr	40.2	46.1	56.9	76.3	112.2	178.3	305.5	558.8
	none	21.9	27.9	40.4	67.2	122.4	231.1	451.5	888.1
	poll	21.9	27.6	40.0	63.7	106.6	183.4	323.3	588.3
10245	null	4.3	8.2	16.2	31.8	63.2	126.0	251.7	502.6
	intr	40.0	46.2	56.7	76.3	112.3	178.5	305.9	559.4
	none	22.1	27.8	40.5	67.3	121.8	231.5	449.0	889.4
	poll	21.9	27.8	40.3	65.5	113.2	199.1	350.7	629.0
20485	null	4.3	8.2	16.0	31.6	62.9	125.3	250.5	500.5
	intr	40.2	46.2	56.7	76.2	112.0	177.9	305.0	557.2
	none	22.0	27.7	40.5	67.1	121.5	230.3	449.5	885.0
	poll	21.9	27.8	40.3	66.1	117.3	212.6	381.2	683.7

Table B-1. Synthetic workload performance data (average time per outer loop iteration, in microseconds), two-message communication events.

Useful work per poll (cycles)		Useful work per communication event (cycles)								
		125	250	500	1000	2000	4000	8000	16000	32000
15	null	4.3	8.2	16.2	31.9	63.1	125.6	250.9	500.8	1000.4
	intr	56.1	62.4	75.0	97.3	136.4	206.6	338.5	593.9	1096.9
	none	31.4	38.6	55.1	90.8	163.9	310.9	602.3	1187.4	2367.0
	poll	47.7	68.7	110.1	192.3	355.5	681.8	1334.0	2637.2	5241.5
25	null	4.3	8.2	16.1	31.8	63.0	125.6	250.8	501.0	1001.1
	intr	56.0	62.3	75.0	97.1	136.3	206.4	338.4	594.3	1097.7
	none	31.4	38.5	55.2	90.9	164.3	311.6	602.0	1191.6	2371.2
	poll	40.8	55.3	83.5	139.4	250.4	472.0	915.4	1801.6	3572.5
45	null	4.3	8.3	16.1	31.7	63.1	125.7	250.8	501.2	999.0
	intr	55.9	62.3	74.9	97.3	136.3	206.7	338.6	594.2	1095.6
	none	31.3	38.6	55.2	90.8	164.2	310.0	605.3	1192.0	2365.0
	poll	36.4	46.4	65.9	104.3	180.3	331.4	632.5	1234.9	2433.0
85	null	4.3	8.2	16.0	31.6	63.0	125.3	250.5	500.5	1005.3
	intr	56.0	62.3	74.9	97.2	136.2	206.3	338.3	593.7	1101.9
	none	31.4	38.5	55.1	91.0	163.6	309.2	601.5	1187.3	2378.7
	poll	33.7	41.1	55.7	83.9	139.3	249.0	468.0	905.7	1789.8
165	null	4.3	8.3	16.1	31.7	63.1	126.0	251.6	502.7	1002.5
	intr	55.9	62.3	74.9	97.3	136.4	206.9	339.3	595.9	1099.1
	none	31.3	39.7	55.1	90.7	163.8	309.5	606.1	1195.3	2358.6
	poll	32.4	38.7	50.7	73.9	118.6	206.5	380.9	729.2	1422.2
325	null	4.3	8.2	16.2	31.8	63.0	125.5	250.7	500.9	1000.6
	intr	56.1	62.3	74.9	97.2	136.1	206.4	338.3	594.1	1097.2
	none	31.3	38.5	55.1	90.8	164.2	311.8	604.0	1191.7	2358.5
	poll	31.8	37.6	48.9	69.7	109.2	185.6	336.5	636.4	1235.1
645	null	4.4	8.3	16.1	31.7	63.1	125.7	251.0	501.3	1004.8
	intr	56.0	62.3	75.1	97.4	136.4	206.7	338.6	594.5	1101.4
	none	31.3	38.6	55.3	90.8	164.1	309.9	606.3	1193.3	2375.9
	poll	31.6	37.5	49.2	70.1	108.3	180.4	320.5	597.3	1152.0
1285	null	4.4	8.3	16.1	31.7	63.1	125.8	251.1	501.6	1003.0
	intr	56.0	62.2	75.1	97.3	136.4	206.9	339.0	594.8	1099.6
	none	32.8	38.6	55.3	90.8	164.4	312.6	605.6	1197.2	2366.6
	poll	31.5	37.8	50.7	73.1	113.1	185.4	322.4	589.4	1117.9
2565	null	4.4	8.4	16.1	31.9	63.3	126.1	251.8	503.0	1002.2
	intr	56.0	62.3	75.0	97.4	136.5	207.1	339.7	596.2	1098.8
	none	31.4	38.6	55.4	90.9	165.7	311.4	607.1	1194.1	2372.0
	poll	31.3	38.2	52.6	78.5	122.6	199.7	340.7	607.8	1126.1
5125	null	4.4	8.3	16.1	31.8	63.3	126.2	251.7	503.1	1004.9
	intr	56.0	62.4	74.9	97.5	136.5	207.2	339.4	596.3	1101.7
	none	31.3	38.5	55.4	92.1	164.7	311.9	603.9	1200.0	2369.5
	poll	31.4	38.5	54.1	84.8	135.7	221.9	373.5	650.6	1176.6
10245	null	4.3	8.3	16.1	31.7	62.9	125.4	250.2	500.3	1005.0
	intr	55.9	62.3	74.9	97.1	136.0	206.2	338.0	593.5	1101.7
	none	32.8	38.4	55.1	90.9	162.8	308.6	602.1	1191.0	2366.6
	poll	31.4	38.4	54.4	87.3	148.0	249.3	419.7	717.2	1269.7
20485	null	4.4	8.3	16.1	31.9	63.3	126.2	251.8	502.7	1001.6
	intr	56.0	62.4	75.2	97.3	136.6	207.2	339.6	595.8	1098.4
	none	31.4	38.7	55.3	92.1	165.0	312.1	606.5	1197.4	2368.3
	poll	31.3	38.6	55.1	89.6	157.1	278.8	482.6	819.6	1411.8

Table B-2. Synthetic workload performance data (average time per outer loop iteration, in microseconds), three-message communication events.

Useful work per poll (cycles)		Useful work per communication event (cycles)								
		125	**250**	**500**	**1000**	**2000**	**4000**	**8000**	**16000**	**32000**
15	null	4.3	8.2	16.2	31.8	63.0	125.7	250.7	500.7	1001.1
	intr	51.4	57.8	70.2	91.9	129.2	198.1	326.9	582.2	1084.1
	none	29.5	36.5	52.9	88.2	159.9	305.8	593.0	1174.2	2342.6
	poll	44.5	65.0	106.1	187.7	350.8	676.8	1328.1	2631.2	5240.0
25	null	4.3	8.2	16.0	31.7	63.0	125.4	250.5	500.7	1000.4
	intr	51.4	57.6	70.2	92.0	129.0	198.0	327.0	582.1	1083.5
	none	29.5	36.5	53.0	88.8	159.7	305.0	592.5	1171.6	2327.4
	poll	37.8	51.8	79.7	135.2	245.7	466.9	909.9	1795.9	3565.5
45	null	4.3	8.2	16.1	31.8	63.1	125.5	250.6	500.9	999.3
	intr	51.3	57.6	70.3	92.1	129.2	198.2	327.2	582.5	1082.4
	none	29.4	36.4	53.0	88.3	161.7	305.0	594.0	1166.7	2325.5
	poll	33.6	43.1	62.2	100.2	175.5	326.2	627.2	1229.7	2428.8
85	null	4.3	8.3	16.1	31.8	63.2	125.7	250.8	501.0	1006.4
	intr	51.3	57.6	70.2	91.9	129.2	198.4	327.3	582.4	1089.7
	none	29.5	36.5	53.0	88.0	161.5	304.5	592.1	1172.0	2337.7
	poll	31.1	38.0	52.0	79.7	134.7	244.5	463.3	901.2	1786.5
165	null	4.3	8.2	16.1	31.7	63.1	125.4	250.4	500.8	1002.4
	intr	51.3	57.6	70.2	92.0	129.1	198.1	326.8	582.2	1085.4
	none	29.5	36.4	54.3	88.1	160.4	302.8	591.7	1174.0	2318.7
	poll	29.8	35.5	46.8	69.1	112.9	199.8	373.1	720.5	1415.7
325	null	4.3	8.3	16.2	31.7	63.0	125.5	250.5	500.3	1001.6
	intr	51.3	57.5	70.3	91.9	129.1	198.0	327.0	581.8	1084.6
	none	29.4	38.2	52.8	88.2	160.1	304.9	595.8	1176.0	2332.7
	poll	29.5	34.6	44.8	64.6	103.1	178.7	328.7	627.9	1227.9
645	null	4.4	8.4	16.1	31.9	63.2	125.9	251.4	502.3	1005.0
	intr	51.4	57.7	70.2	92.0	129.3	198.5	327.9	583.9	1088.0
	none	29.4	36.4	53.0	88.2	167.5	306.4	596.7	1181.3	2340.1
	poll	31.5	34.8	45.2	64.6	101.2	172.0	311.1	587.9	1140.8
1285	null	4.4	8.3	16.2	31.8	63.1	125.8	251.1	502.0	1003.0
	intr	51.4	57.7	70.3	92.0	129.2	198.4	327.7	583.4	1086.2
	none	29.4	36.5	52.9	88.2	161.6	306.3	596.0	1177.7	2331.8
	poll	29.5	35.4	47.1	67.9	105.2	174.8	309.3	574.3	1101.0
2565	null	4.3	8.3	16.1	31.9	63.2	125.8	251.2	501.8	1002.1
	intr	51.4	57.6	70.3	92.1	129.4	198.5	327.7	583.3	1085.3
	none	29.4	36.5	53.0	88.8	161.2	305.0	593.1	1175.4	2323.1
	poll	29.5	35.9	49.7	73.8	114.6	187.3	322.6	584.4	1099.7
5125	null	4.4	8.3	16.1	31.8	63.3	126.1	251.5	502.9	1004.5
	intr	51.5	57.6	70.3	92.1	129.4	198.8	328.1	584.3	1087.7
	none	29.5	36.4	53.2	89.0	161.7	306.3	594.4	1180.8	2335.8
	poll	29.4	36.3	51.6	80.6	129.1	210.0	352.0	619.5	1135.7
10245	null	4.3	8.2	16.0	31.8	62.9	125.4	250.2	500.0	1005.2
	intr	51.4	57.7	70.3	91.8	128.9	197.8	326.7	581.5	1088.4
	none	29.4	36.5	52.9	89.2	160.0	305.4	593.1	1170.0	2326.4
	poll	29.5	36.3	52.1	85.8	144.0	239.3	397.4	678.8	1211.3
20485	null	4.4	8.3	16.1	31.9	63.2	126.0	251.7	502.9	1001.7
	intr	51.4	57.6	70.2	92.1	129.3	198.8	328.1	584.4	1084.9
	none	29.4	36.6	53.3	88.6	161.7	305.5	598.6	1181.1	2327.1
	poll	29.6	36.3	52.7	87.7	153.4	270.7	463.1	779.2	1337.2

Table B-3. Synthetic workload performance data (average time per outer loop iteration, in microseconds), four-message communication events.

	Blocked LU		Water		Barnes-Hut	
	HW	**SW**	**HW**	**SW**	**HW**	**SW**
1 proc	25.29	25.29	13.75	13.87	24.29	24.36
2 procs	13.96	13.96	7.36	7.38	14.98	15.02
4 procs	7.74	7.72	4.01	4.02	8.27	8.24
8 procs	4.49	4.43	2.27	2.27	4.66	4.70
16 procs	2.58	2.56	1.59	1.53	2.58	2.59
32 procs	1.75	1.70	1.17	1.05	1.52	1.51

Table B-4. CM-5 CRL application running times (in seconds), HW *vs.* SW synchronization. HW figures obtained with baseline CRL version; SW figures obtained with version that implements global synchronization primitives in software. All values are averages computed over three consecutive runs.

B.2 Global Synchronization: Hardware vs. Software

As is discussed in Section 4.7, the baseline CRL implementation for the CM-5 takes advantage of the CM-5's hardware support for global synchronization and communication to implement CRL's global synchronization primitives. This section presents results from a set of experiments comparing the performance of the baseline CM-5 implementation with one in which global synchronization primitives are implemented entirely in software using active messages.

Table B-4 presents the results from these experiments—absolute running times for the three applications when hardware (HW) or software (SW) is used to implement global synchronization primitives. Slight variations (averaging approximately 0.9 percent) between the HW figures shown in this table and those shown in Table 6-5 for CM-5 CRL are due to small variations in the running times of the applications; the values shown in the two tables were obtained in different sets of experiments.

The differences between the two sets of figures (HW and SW) are quite modest. On the average, the SW running times are approximately 0.9 percent smaller than the HW ones; eliminating one outlier (Water on 32 processors, where the SW figure is just over 10 percent smaller than the HW one) reduces this overall average to 0.4 percent. Given that this figure is of the same order of magnitude as the variation between the HW values in Table B-4 and those for CM-5 CRL shown in Table 6-5, it is entirely possible that the apparent differences between the HW and SW figures are not statistically significant.

The data in Table B-4 indicates that using software techniques to implement global synchronization primitives does not have a significant impact on delivered application performance. The reasons for this are twofold. First, as can be seen in Table B-5, all three applications invoke the global synchronization primitives relatively infrequently. Even Blocked LU, the most synchronization-intensive of these applications, only executes 150 barrier synchronizations on 32 processors, corresponding to an average time between barriers of over 10 milliseconds. Second, in the software implementation of the global

Type	Blocked LU	Water	Barnes-Hut
Barrier	150	5	4
Reduction	0	0	6
Broadcast	0	0	0

Table B-5. Dynamic counts of global synchronization primitives (on 32 processors).

synchronization primitives, processors that are waiting for a global synchronization event to complete are continually polling for incoming messages. Thus, the overhead of receiving any messages unrelated to the global synchronization that arrive during such an interval is somewhat lower than when hardware-based global synchronization primitives are used, where similar messages are delivered in an interrupt-driven style.

B.3 Basic CRL Latencies

This section contains the raw data obtained with the simple latency microbenchmark described in Section 6.1. Table B-6 describes the 26 different types of events measured by the microbenchmark. Table B-7 shows the latencies obtained when running the microbenchmark on the CM-5.

Table B-8 shows the latencies obtained when running the microbenchmark on Alewife when message buffers are flushed and protocol message handlers are transitioned into threads (as discussed in Section 5.2.2). These figures represent the latencies actually seen by CRL applications running on the current Alewife hardware (using first-run CMMU parts).

Table B-9 shows the latencies obtained by running the microbenchmark on Alewife when message buffers *are not* flushed and protocol message handers *are not* transitioned into threads. These measurements are therefore indicative of the latencies that should be seen by CRL applications running on Alewife hardware after the CMMU respin effort is complete.

Event		Description
Map	miss	Map a region that is not already mapped locally and not present in the URC
Map	hit [a]	Map a region that is not already mapped locally but is present in the URC
Map	hit [b]	Map a region that is already mapped locally
Unmap	[c]	Unmap a region that is mapped more than once locally
Unmap	[d]	Unmap a region that is only mapped once locally (and insert it into the URC)
Start read	miss, 0 copies	Initiate a read operation on a region in the RemoteInvalid state, only the home node has a valid (exclusive) copy of the region data
Start read	miss, 1 copies	As above, but both the home node and one other remote region have valid (shared) copies of the region data
Start read	miss, 2 copies	As above, but both the home node and two other remote regions have valid (shared) copies of the region data
Start read	miss, 3 copies	As above, but both the home node and three other remote regions have valid (shared) copies of the region data
Start read	miss, 4 copies	As above, but both the home node and four other remote regions have valid (shared) copies of the region data
Start read	miss, 5 copies	As above, but both the home node and five other remote regions have valid (shared) copies of the region data
Start read	miss, 6 copies	As above, but both the home node and six other remote regions have valid (shared) copies of the region data
Start read	hit [e]	Initiate a read operation on a region in the RemoteShared state
Start read	hit [f]	Initiate a read operation on a region in the RemoteSharedRip state
End read	[g]	Terminate a read operation, leaving the region in the RemoteSharedRip state
End read	[h]	Terminate a read operation, leaving the region in the RemoteShared state
Start write	miss, 0 inv	Initiate a write operation on a region in the RemoteInvalid state, only the home node has a valid (exclusive) copy of the region data
Start write	miss, 1 inv	As above, but both the home node and one other remote region have valid (shared) copies of the region data
Start write	miss, 2 inv	As above, but both the home node and two other remote regions have valid (shared) copies of the region data
Start write	miss, 3 inv	As above, but both the home node and three other remote regions have valid (shared) copies of the region data
Start write	miss, 4 inv	As above, but both the home node and four other remote regions have valid (shared) copies of the region data
Start write	miss, 5 inv	As above, but both the home node and five other remote regions have valid (shared) copies of the region data
Start write	miss, 6 inv	As above, but both the home node and six other remote regions have valid (shared) copies of the region data
Start write	modify	Initiate a write operation on a region in the RemoteShared state, no other remote nodes have a valid copy of the region data
Start write	hit	Initiate a write operation on a region in the RemoteModified state
End write		Terminate a write operation, leaving the region in the RemoteModified state

Table B-6. Events measured by latency microbenchmark.

Event		Region Size (bytes) 16	64	256	1024
Map	miss	2244.2	2185.3	2198.1	2225.0
Map	hit [a]	106.2	108.2	150.4	135.4
Map	hit [b]	74.6	78.7	87.0	94.1
Unmap	[c]	27.2	27.5	30.7	33.0
Unmap	[d]	76.5	79.7	86.4	89.3
Start read	miss, 0 copies	1925.4	2280.6	3963.5	6943.7
Start read	miss, 1 copies	2158.4	2479.0	4146.6	7172.8
Start read	miss, 2 copies	2160.3	2504.3	4194.9	8278.4
Start read	miss, 3 copies	2166.4	2500.8	4174.1	7298.2
Start read	miss, 4 copies	2193.0	2526.4	4204.8	7350.1
Start read	miss, 5 copies	2218.6	2544.3	5507.2	7378.6
Start read	miss, 6 copies	2225.9	2557.1	4222.4	7447.0
Start read	hit [e]	79.0	77.8	80.6	81.6
Start read	hit [f]	80.6	75.5	78.1	80.3
End read	[g]	84.2	79.4	83.2	104.0
End read	[h]	98.9	86.7	95.0	96.3
Start write	miss, 0 inv	1885.4	2206.1	3909.1	6953.0
Start write	miss, 1 inv	3619.8	3950.1	5644.5	8702.4
Start write	miss, 2 inv	3894.4	4201.0	5884.5	8965.8
Start write	miss, 3 inv	4085.1	4394.9	5901.4	9171.5
Start write	miss, 4 inv	4298.6	4570.9	6239.7	9382.1
Start write	miss, 5 inv	4466.9	4786.9	6442.6	9581.1
Start write	miss, 6 inv	4663.0	4955.2	6647.4	9777.6
Start write	modify	1441.3	1402.2	1438.1	1447.4
Start write	hit	74.6	72.0	75.8	78.1
End write		89.0	79.7	83.5	87.0

Table B-7. CM-5 CRL latencies (in cycles @ 32 MHz).

Event		Region Size (bytes) 16	64	256	1024
Map	miss	825.5	833.5	839.9	837.2
Map	hit [a]	137.6	137.4	136.5	139.8
Map	hit [b]	104.4	107.4	105.5	105.6
Unmap	[c]	37.5	39.4	39.8	38.1
Unmap	[d]	76.7	77.9	80.2	77.2
Start read	miss, 0 copies	1029.9	1054.9	1173.9	1865.3
Start read	miss, 1 copies	1123.4	1149.9	1265.2	1958.7
Start read	miss, 2 copies	1135.1	1166.0	1277.4	1974.6
Start read	miss, 3 copies	1150.5	1175.2	1287.2	1992.1
Start read	miss, 4 copies	1164.9	1191.5	1297.4	2001.0
Start read	miss, 5 copies	1175.6	1211.0	1308.7	2016.8
Start read	miss, 6 copies	1190.3	1216.8	1318.5	2029.6
Start read	hit [e]	47.0	45.9	47.2	47.0
Start read	hit [f]	51.8	51.1	51.2	50.1
End read	[g]	45.1	44.3	44.8	45.0
End read	[h]	50.9	51.3	51.6	50.7
Start write	miss, 0 inv	1021.6	1041.1	1175.7	1860.4
Start write	miss, 1 inv	1759.7	1779.8	1914.4	2601.3
Start write	miss, 2 inv	2132.2	2158.4	2291.3	2982.6
Start write	miss, 3 inv	2432.1	2451.4	2592.5	3286.4
Start write	miss, 4 inv	2704.2	2728.9	2863.5	3560.3
Start write	miss, 5 inv	2980.7	3012.7	3139.0	3838.4
Start write	miss, 6 inv	3288.4	3309.9	3419.3	4138.6
Start write	modify	1052.5	1044.0	1049.8	1045.5
Start write	hit	43.7	43.0	42.1	43.7
End write		52.9	53.3	52.5	52.9

Table B-8. Alewife CRL latencies (in cycles @ 20 MHz), with CMMU workarounds.

Event		Region Size (bytes) 16	64	256	1024
Map	miss	825.2	832.6	834.6	835.0
Map	hit [a]	136.6	136.3	136.9	140.9
Map	hit [b]	103.9	106.0	105.7	106.1
Unmap	[c]	37.6	38.8	39.7	39.5
Unmap	[d]	76.7	77.9	79.8	78.8
Start read	miss, 0 copies	602.7	593.9	649.0	959.6
Start read	miss, 1 copies	699.4	699.6	735.6	1046.2
Start read	miss, 2 copies	708.5	705.3	747.6	1061.7
Start read	miss, 3 copies	721.0	715.2	756.7	1077.5
Start read	miss, 4 copies	732.6	732.9	767.5	1088.5
Start read	miss, 5 copies	747.6	750.3	778.0	1104.9
Start read	miss, 6 copies	762.1	762.6	790.9	1120.7
Start read	hit [e]	50.1	50.1	49.6	51.6
Start read	hit [f]	52.6	52.2	51.4	53.1
End read	[g]	48.0	46.9	47.1	47.7
End read	[h]	53.8	52.9	55.3	54.0
Start write	miss, 0 inv	596.2	595.9	644.8	954.8
Start write	miss, 1 inv	1002.4	997.2	1044.5	1361.3
Start write	miss, 2 inv	1187.8	1183.9	1229.7	1548.5
Start write	miss, 3 inv	1376.0	1373.8	1416.7	1736.6
Start write	miss, 4 inv	1566.1	1563.0	1603.8	1929.8
Start write	miss, 5 inv	1768.0	1772.7	1795.4	2132.4
Start write	miss, 6 inv	1994.4	2002.7	2021.3	2358.1
Start write	modify	567.2	562.0	562.1	561.0
Start write	hit	47.0	46.3	46.9	45.3
End write		56.1	55.6	54.5	55.3

Table B-9. Alewife CRL latencies (in cycles @ 20 MHz), without CMMU workarounds.

B.4 Application Characteristics

This section contains the raw data obtained with the instrumented version of the CRL library described in Section 6.2.4.

Table B-10 shows call event counts for Blocked LU running on one and 32 processors. The first section of the table indicates how many times `rgn_map` was called and, of those calls, how many (1) referenced remote regions and (2) were misses. The second section of the table indicates how many times `rgn_start_read` was called and, of those calls, how many (1) reference remote regions, (2) were misses, and (3) were invoked on a region in either the HomeIip or HomeIipSpecial state. The third section of the table indicates how many times `rgn_start_write` was called and, like the previous section, provides further information about how many of the calls referenced remote regions, could not be satisfied locally (missed), or were invoked on a home region in an 'Iip' state. Finally, the fourth section of the table indicates how many times `rgn_flush` was called. As can be seen from Figure 6-3, none of the applications call `rgn_flush` directly, so any calls that are counted in this section of the table are because calls to `rgn_flush` when evicting regions from the URC (see Section 4.5). Tables B-11 and B-12 show the same information for Water and Barnes-Hut, respectively.

Table B-13 shows message counts for all three applications running on one and 32 processors. The first and second sections of the table provide counts for home-to-remote and remote-to-home protocol messages, respectively. The third section of the table shows the total number of protocol messages and, of those messages, how many were placed in the incoming message queue upon arriving at their intended destination (see Section A.7). Finally, the fourth section of the table shows counts for the MsgRgnInfoReq and MsgRgnInfoAck messages required when calls to `rgn_map` cannot be satisfied locally (see Section 4.5).

Event	CM-5 1 proc	CM-5 32 procs	Alewife 1 proc	Alewife 32 procs
Map	84576	2807	84576	2807
(remote)	0	1286	0	1286
(miss)	0	430	0	430
(unmap)	84576	2807	84576	2807
Start read	41701	1435	41701	1435
(remote)	0	1253	0	1253
(miss)	0	397	0	397
(iip)	0	0	0	0
(end)	41701	1435	41701	1435
Start write	42925	1341	42925	1341
(remote)	0	0	0	0
(miss)	0	0	0	0
(iip)	0	0	0	0
(end)	42925	1341	42925	1341
Flush	0	0	0	0

Table B-10. Call event counts for Blocked LU; all values are per-processor averages computed over three consecutive runs.

Event	CM-5 1 proc	CM-5 32 procs	Alewife 1 proc	Alewife 32 procs
Map	—	—	—	—
(remote)	—	—	—	—
(miss)	—	—	—	—
(unmap)	—	—	—	—
Start read	263682	8242	263682	8242
(remote)	0	3970	0	3970
(miss)	0	380	0	514
(iip)	0	0	0	1
(end)	263682	8242	263682	8242
Start write	5640	437	5640	437
(remote)	0	260	0	260
(miss)	0	291	0	290
(iip)	0	0	0	0
(end)	5640	437	5640	437
Flush	0	0	0	0

Table B-11. Call event counts for Water; all values are per-processor averages computed over three consecutive runs.

Event	CM-5 1 proc	CM-5 32 procs	Alewife 1 proc	Alewife 32 procs
Map	983598	30763	983598	30763
(remote)	0	27376	0	27378
(miss)	0	344	0	340
(unmap)	983598	30763	983598	30763
Start read	959988	30027	959988	30027
(remote)	0	26908	0	26909
(miss)	0	1135	0	1142
(iip)	0	3	0	4
(end)	959988	30027	959988	30027
Start write	32236	1313	32236	1189
(remote)	0	950	0	826
(miss)	0	301	0	300
(iip)	0	0	0	0
(end)	32236	1313	32236	1189
Flush	0	344	0	340

Table B-12. Call event counts for Barnes-Hut; all values are per-processor averages computed over three consecutive runs.

Msg Type	Blocked LU CM-5	Blocked LU Alewife	Water CM-5	Water Alewife	Barnes-Hut CM-5	Barnes-Hut Alewife
MsgRInvalidate	0	0	50	99	209	207
MsgWInvalidate	0	0	576	649	872	875
MsgSharedAckData	397	397	378	508	1118	1123
MsgExclusiveAckData	0	0	248	240	88	83
MsgModifyAck	0	0	13	20	148	150
MsgModifyAckData	0	0	0	0	3	3
MsgInvalidateAck	0	0	365	488	852	855
MsgInvalidateAckData	0	0	260	260	228	226
MsgRelease	0	0	0	0	1	1
MsgSharedReq	397	397	378	508	1118	1123
MsgExclusiveReq	0	0	248	240	88	83
MsgModifyReq	0	0	13	20	151	153
MsgFlush	0	0	0	0	113	113
MsgFlushData	0	0	0	0	11	10
total protocol msgs	793	793	2527	3034	4999	5007
(queued)	4	201	30	1067	274	1239
MsgRgnInfoReq	430	430	0	0	344	340
(ack)	430	430	0	0	344	340

Table B-13. Message counts for 32 processors; all values are per-processor averages computed over three consecutive runs.

	Total	User	CRL, ops	CRL, map
1 proc	54.67	53.57	0.44	0.66
2 procs	28.42	27.69	0.33	0.41
4 procs	14.83	14.38	0.22	0.23
8 procs	7.89	7.58	0.18	0.13
16 procs	4.25	4.05	0.12	0.08
32 procs	2.40	2.24	0.10	0.06

(a) Blocked LU (500x500 matrix, 10x10 blocks)

	Total	User	CRL, ops	CRL, map
1 proc	24.16	22.87	1.29	0.00
2 procs	12.84	12.10	0.74	0.00
4 procs	6.93	6.49	0.44	0.00
8 procs	3.68	3.38	0.30	0.00
16 procs	2.00	1.76	0.24	0.00
32 procs	1.18	0.95	0.23	0.00

(b) Water (512 molecules)

	Total	User	CRL, ops	CRL, map
1 proc	34.80	23.57	4.40	6.83
2 procs	19.05	11.85	2.85	4.34
4 procs	10.02	6.03	1.64	2.34
8 procs	5.42	3.15	0.97	1.30
16 procs	2.85	1.57	0.61	0.66
32 procs	1.58	0.82	0.43	0.33

(c) Barnes-Hut (4,096 bodies)

Table B-14. Breakdown of Alewife CRL running times (in seconds).

B.5 Alewife CRL Profiling

Table B-14 shows the breakdown of running times for the three applications obtained using the profiled version of the CRL library described in Section 6.3. For each application, the "Total" column shows the total running time (as shown in Table 6-5), the "User" column shows the total time spent running application code, the "CRL, ops" column shows the total time spent in the CRL library starting and ending operations, and the "CRL, map" column shows the total time spent in the CRL library executing map/unmap code. The "CRL, map" and "CRL, ops" figures include "spin time" spent waiting for communication events (*i.e.,* those related to calls to `rgn_map` or `rgn_start_`*op* that miss) to complete.

Bulk Transfer	One-way Message Latency (cycles)				
Cost (cycles/byte)	**264**	**314**	**364**	**414**	**464**
1.110	85.05 ± 0.01	85.35 ± 0.01	85.61 ± 0.00	85.87 ± 0.00	86.13 ± 0.00
1.235	85.15 ± 0.01	85.42 ± 0.01	85.70 ± 0.01	85.94 ± 0.01	86.22 ± 0.00
1.485	85.30 ± 0.00	85.64 ± 0.01	85.92 ± 0.01	86.17 ± 0.01	86.45 ± 0.01
1.735	85.54 ± 0.00	85.82 ± 0.01	86.08 ± 0.01	86.34 ± 0.00	86.61 ± 0.01
1.985	85.77 ± 0.01	86.04 ± 0.01	86.31 ± 0.01	86.56 ± 0.01	86.83 ± 0.01
2.235	85.95 ± 0.00	86.20 ± 0.01	86.47 ± 0.01	86.73 ± 0.02	86.98 ± 0.01
2.485	86.11 ± 0.01	86.39 ± 0.00	86.65 ± 0.01	86.91 ± 0.00	87.19 ± 0.01
2.735	86.30 ± 0.01	86.61 ± 0.01	86.86 ± 0.01	87.13 ± 0.00	87.38 ± 0.01
2.985	86.50 ± 0.01	86.73 ± 0.00	86.99 ± 0.01	87.26 ± 0.02	87.52 ± 0.01

Bulk Transfer	One-way Message Latency (cycles)			
Cost (cycles/byte)	**514**	**564**	**614**	**664**
1.110	86.38 ± 0.00	86.65 ± 0.00	86.90 ± 0.00	87.17 ± 0.00
1.235	86.46 ± 0.01	86.74 ± 0.01	86.98 ± 0.01	87.26 ± 0.01
1.485	86.70 ± 0.00	86.98 ± 0.00	87.20 ± 0.01	87.50 ± 0.01
1.735	86.88 ± 0.00	87.13 ± 0.02	87.40 ± 0.01	87.67 ± 0.00
1.985	87.10 ± 0.01	87.36 ± 0.01	87.61 ± 0.01	87.90 ± 0.01
2.235	87.25 ± 0.01	87.53 ± 0.01	87.77 ± 0.01	88.06 ± 0.01
2.485	87.44 ± 0.02	87.70 ± 0.01	87.96 ± 0.02	88.24 ± 0.01
2.735	87.64 ± 0.02	87.91 ± 0.01	88.16 ± 0.01	88.44 ± 0.01
2.985	87.76 ± 0.01	88.04 ± 0.02	88.28 ± 0.01	88.54 ± 0.01

Table B-15. Running times (in Mcycles @ 20 MHz) for Blocked LU (500x500 matrix, 10x10 blocks) on 16 processors using the modified Alewife CRL implementation.

B.6 Sensitivity Analysis

The tables in this section present the raw data obtained with the modified CRL implementation in the sensitivity analysis experiments described in Section 6.5. Measurements for both 16 and 32 processors are provided for Blocked LU (Tables B-15 and B-16), Water (Tables B-17 and B-18), Barnes-Hut with 4,096 bodies (Tables B-19 and B-20), and Barnes-Hut with 16,384 bodies (Tables B-21 and B-22).

Each table provides an entry for all combinations of one-way message latencies (264, 314, 364, 414, 464, 514, 564, 614, and 664 cycles) and bulk transfer costs (1.110, 1.235, 1.485, 1.735, 1.985, 2.235, 2.485, 2.735, and 2.985 cycles per byte) that were used. Each table entry indicates the average running time (in millions of cycles at 20 MHz) computed over three consecutive runs for the appropriate application and system configuration; numbers after the ± represent the standard deviation of the three measured running times.

Bulk Transfer	One-way Message Latency (cycles)				
Cost (cycles/byte)	**264**	**314**	**364**	**414**	**464**
1.110	47.97 $\pm$ 0.00	48.32 $\pm$ 0.01	48.69 $\pm$ 0.01	49.02 $\pm$ 0.01	49.37 $\pm$ 0.01
1.235	48.07 $\pm$ 0.01	48.44 $\pm$ 0.00	48.80 $\pm$ 0.01	49.11 $\pm$ 0.01	49.45 $\pm$ 0.01
1.485	48.27 $\pm$ 0.00	48.67 $\pm$ 0.01	49.02 $\pm$ 0.00	49.34 $\pm$ 0.01	49.67 $\pm$ 0.01
1.735	48.51 $\pm$ 0.01	48.90 $\pm$ 0.02	49.25 $\pm$ 0.02	49.56 $\pm$ 0.01	49.87 $\pm$ 0.00
1.985	48.77 $\pm$ 0.00	49.13 $\pm$ 0.01	49.49 $\pm$ 0.02	49.79 $\pm$ 0.00	50.09 $\pm$ 0.02
2.235	49.03 $\pm$ 0.02	49.35 $\pm$ 0.01	49.67 $\pm$ 0.01	49.99 $\pm$ 0.02	50.31 $\pm$ 0.01
2.485	49.23 $\pm$ 0.02	49.59 $\pm$ 0.02	49.92 $\pm$ 0.01	50.23 $\pm$ 0.01	50.53 $\pm$ 0.02
2.735	49.48 $\pm$ 0.01	49.86 $\pm$ 0.03	50.17 $\pm$ 0.02	50.48 $\pm$ 0.01	50.82 $\pm$ 0.01
2.985	49.70 $\pm$ 0.00	50.05 $\pm$ 0.02	50.38 $\pm$ 0.01	50.68 $\pm$ 0.02	50.98 $\pm$ 0.01

Bulk Transfer	One-way Message Latency (cycles)			
Cost (cycles/byte)	**514**	**564**	**614**	**664**
1.110	49.73 $\pm$ 0.01	50.10 $\pm$ 0.02	50.47 $\pm$ 0.01	50.86 $\pm$ 0.02
1.235	49.79 $\pm$ 0.01	50.19 $\pm$ 0.02	50.53 $\pm$ 0.03	50.93 $\pm$ 0.03
1.485	50.01 $\pm$ 0.01	50.36 $\pm$ 0.02	50.70 $\pm$ 0.01	51.08 $\pm$ 0.02
1.735	50.19 $\pm$ 0.02	50.56 $\pm$ 0.01	50.91 $\pm$ 0.02	51.26 $\pm$ 0.02
1.985	50.42 $\pm$ 0.01	50.76 $\pm$ 0.01	51.14 $\pm$ 0.02	51.45 $\pm$ 0.01
2.235	50.64 $\pm$ 0.01	50.97 $\pm$ 0.01	51.28 $\pm$ 0.03	51.65 $\pm$ 0.02
2.485	50.85 $\pm$ 0.02	51.18 $\pm$ 0.00	51.53 $\pm$ 0.01	51.89 $\pm$ 0.01
2.735	51.10 $\pm$ 0.02	51.45 $\pm$ 0.02	51.80 $\pm$ 0.01	52.11 $\pm$ 0.01
2.985	51.29 $\pm$ 0.01	51.63 $\pm$ 0.01	51.96 $\pm$ 0.00	52.33 $\pm$ 0.03

Table B-16. Running times (in Mcycles @ 20 MHz) for Blocked LU (500x500 matrix, 10x10 blocks) on 32 processors using the modified Alewife CRL implementation.

Bulk Transfer	One-way Message Latency (cycles)				
Cost (cycles/byte)	264	314	364	414	464
1.110	39.89 ± 0.04	40.15 ± 0.04	40.35 ± 0.01	40.91 ± 0.07	41.13 ± 0.03
1.235	40.15 ± 0.12	40.30 ± 0.07	40.66 ± 0.06	40.98 ± 0.04	41.40 ± 0.05
1.485	40.51 ± 0.12	40.74 ± 0.19	41.13 ± 0.19	41.60 ± 0.11	41.91 ± 0.12
1.735	40.85 ± 0.27	41.24 ± 0.23	41.74 ± 0.04	42.08 ± 0.11	42.35 ± 0.32
1.985	41.40 ± 0.26	41.93 ± 0.07	42.27 ± 0.05	42.63 ± 0.07	42.92 ± 0.14
2.235	41.93 ± 0.16	42.40 ± 0.13	42.85 ± 0.13	43.06 ± 0.29	43.12 ± 0.24
2.485	42.62 ± 0.11	42.99 ± 0.13	43.27 ± 0.20	43.59 ± 0.08	44.21 ± 0.28
2.735	43.09 ± 0.15	43.64 ± 0.28	44.02 ± 0.10	44.36 ± 0.26	44.99 ± 0.14
2.985	43.62 ± 0.54	44.61 ± 0.19	45.27 ± 0.36	45.42 ± 0.29	45.42 ± 0.16

Bulk Transfer	One-way Message Latency (cycles)			
Cost (cycles/byte)	514	564	614	664
1.110	41.57 ± 0.10	41.91 ± 0.12	42.54 ± 0.20	42.40 ± 0.48
1.235	41.64 ± 0.18	41.82 ± 0.05	42.42 ± 0.26	42.48 ± 0.62
1.485	42.20 ± 0.21	42.41 ± 0.51	42.57 ± 0.62	42.92 ± 0.61
1.735	42.56 ± 0.20	43.06 ± 0.60	43.50 ± 0.65	43.63 ± 0.63
1.985	43.46 ± 0.14	43.85 ± 0.14	44.21 ± 0.17	44.04 ± 0.51
2.235	43.45 ± 0.25	43.70 ± 0.48	44.46 ± 0.63	44.60 ± 0.52
2.485	44.58 ± 0.37	44.93 ± 0.32	45.52 ± 0.35	45.62 ± 0.19
2.735	45.23 ± 0.28	45.42 ± 0.28	45.64 ± 0.22	46.04 ± 0.21
2.985	45.83 ± 0.20	46.08 ± 0.08	46.41 ± 0.11	47.00 ± 0.63

Table B-17. Running times (in Mcycles @ 20 MHz) for Water (512 molecules) on 16 processors using the modified Alewife CRL implementation.

Bulk Transfer	One-way Message Latency (cycles)				
Cost (cycles/byte)	**264**	**314**	**364**	**414**	**464**
1.110	23.68 ± 0.21	23.77 ± 0.21	24.12 ± 0.25	24.83 ± 0.21	25.11 ± 0.25
1.235	23.84 ± 0.11	24.40 ± 0.30	24.34 ± 0.17	25.01 ± 0.26	25.47 ± 0.17
1.485	24.48 ± 0.16	24.79 ± 0.37	24.97 ± 0.31	25.44 ± 0.17	25.68 ± 0.52
1.735	24.91 ± 0.15	25.22 ± 0.64	25.29 ± 0.20	25.90 ± 0.23	26.74 ± 0.57
1.985	25.62 ± 0.25	25.83 ± 0.53	25.87 ± 0.38	26.51 ± 0.27	26.62 ± 0.45
2.235	26.42 ± 0.22	26.16 ± 0.23	26.27 ± 0.14	27.14 ± 0.42	27.41 ± 0.24
2.485	26.46 ± 0.45	26.85 ± 0.47	27.40 ± 0.34	28.00 ± 0.39	28.03 ± 0.62
2.735	27.17 ± 0.63	27.78 ± 0.75	28.13 ± 0.47	28.53 ± 0.27	29.50 ± 0.48
2.985	28.10 ± 0.47	28.50 ± 0.53	29.35 ± 0.54	29.58 ± 0.57	30.03 ± 0.23

Bulk Transfer	One-way Message Latency (cycles)			
Cost (cycles/byte)	**514**	**564**	**614**	**664**
1.110	25.69 ± 0.35	26.36 ± 0.41	26.85 ± 0.65	27.57 ± 0.81
1.235	25.80 ± 0.34	26.70 ± 0.40	27.20 ± 0.21	27.86 ± 0.32
1.485	26.33 ± 0.50	26.93 ± 0.51	28.20 ± 0.54	28.66 ± 0.35
1.735	26.74 ± 0.62	27.01 ± 0.26	28.01 ± 0.56	28.80 ± 0.59
1.985	27.35 ± 0.70	28.47 ± 0.54	28.76 ± 0.37	29.55 ± 0.29
2.235	28.10 ± 0.61	28.79 ± 0.74	29.68 ± 0.98	30.04 ± 1.03
2.485	28.84 ± 0.27	30.12 ± 0.93	30.91 ± 0.52	30.95 ± 0.84
2.735	30.38 ± 0.47	30.16 ± 0.55	31.59 ± 0.28	31.67 ± 0.42
2.985	30.64 ± 0.60	31.32 ± 1.21	32.55 ± 0.80	33.05 ± 0.28

Table B-18. Running times (in Mcycles @ 20 MHz) for Water (512 molecules) on 32 processors using the modified Alewife CRL implementation.

Bulk Transfer	One-way Message Latency (cycles)				
Cost (cycles/byte)	**264**	**314**	**364**	**414**	**464**
1.110	57.76 ± 0.16	58.29 ± 0.24	59.15 ± 0.20	59.90 ± 0.18	60.81 ± 0.11
1.235	57.90 ± 0.16	58.41 ± 0.18	59.20 ± 0.14	60.07 ± 0.16	60.90 ± 0.12
1.485	57.94 ± 0.17	58.28 ± 0.17	59.25 ± 0.21	60.03 ± 0.22	60.97 ± 0.17
1.735	57.68 ± 0.17	58.51 ± 0.20	59.36 ± 0.23	60.10 ± 0.18	60.91 ± 0.14
1.985	58.65 ± 0.17	58.59 ± 0.19	59.48 ± 0.16	60.25 ± 0.20	61.17 ± 0.18
2.235	59.49 ± 0.18	58.63 ± 0.10	59.69 ± 0.23	60.41 ± 0.25	61.31 ± 0.24
2.485	60.19 ± 0.20	58.78 ± 0.11	59.64 ± 0.17	60.49 ± 0.16	61.18 ± 0.17
2.735	60.25 ± 0.26	58.93 ± 0.18	59.80 ± 0.12	60.57 ± 0.13	61.53 ± 0.26
2.985	60.35 ± 0.28	59.08 ± 0.15	59.80 ± 0.17	60.73 ± 0.15	61.47 ± 0.20

Bulk Transfer	One-way Message Latency (cycles)			
Cost (cycles/byte)	**514**	**564**	**614**	**664**
1.110	61.65 ± 0.17	62.60 ± 0.21	63.24 ± 0.16	64.27 ± 0.34
1.235	61.64 ± 0.22	62.70 ± 0.24	63.30 ± 0.33	64.37 ± 0.27
1.485	61.74 ± 0.20	62.63 ± 0.22	63.31 ± 0.17	64.24 ± 0.24
1.735	61.76 ± 0.17	62.66 ± 0.23	63.44 ± 0.15	64.54 ± 0.34
1.985	61.96 ± 0.24	62.74 ± 0.14	63.76 ± 0.29	64.51 ± 0.22
2.235	62.10 ± 0.23	62.89 ± 0.22	63.70 ± 0.20	64.63 ± 0.25
2.485	62.21 ± 0.30	62.85 ± 0.19	63.89 ± 0.22	64.60 ± 0.26
2.735	62.36 ± 0.36	63.19 ± 0.24	64.09 ± 0.29	64.85 ± 0.35
2.985	62.35 ± 0.31	63.24 ± 0.37	63.92 ± 0.14	64.91 ± 0.32

Table B-19. Running times (in Mcycles @ 20 MHz) for Barnes-Hut (4,096 bodies) on 16 processors using the modified Alewife CRL implementation.

Bulk Transfer Cost (cycles/byte)	One-way Message Latency (cycles) 264	314	364	414	464
1.110	31.86 ± 0.12	32.54 ± 0.16	33.22 ± 0.14	33.94 ± 0.12	34.60 ± 0.11
1.235	31.91 ± 0.07	32.60 ± 0.13	33.14 ± 0.10	34.06 ± 0.07	34.58 ± 0.15
1.485	32.05 ± 0.08	32.66 ± 0.10	33.39 ± 0.17	33.91 ± 0.11	34.77 ± 0.13
1.735	32.00 ± 0.11	32.72 ± 0.09	33.37 ± 0.14	34.21 ± 0.10	34.83 ± 0.13
1.985	32.49 ± 0.07	32.79 ± 0.13	33.56 ± 0.07	34.18 ± 0.10	34.84 ± 0.09
2.235	32.94 ± 0.07	32.95 ± 0.16	33.58 ± 0.06	34.30 ± 0.15	35.14 ± 0.13
2.485	33.43 ± 0.15	33.01 ± 0.08	33.73 ± 0.19	34.43 ± 0.20	35.10 ± 0.18
2.735	33.38 ± 0.16	33.08 ± 0.09	33.89 ± 0.04	34.45 ± 0.12	35.37 ± 0.18
2.985	33.55 ± 0.18	33.11 ± 0.08	33.81 ± 0.10	34.58 ± 0.15	35.31 ± 0.16

Bulk Transfer Cost (cycles/byte)	One-way Message Latency (cycles) 514	564	614	664
1.110	35.24 ± 0.17	36.12 ± 0.19	36.64 ± 0.10	37.55 ± 0.14
1.235	35.41 ± 0.13	36.04 ± 0.05	36.83 ± 0.07	37.67 ± 0.09
1.485	35.52 ± 0.20	36.26 ± 0.15	36.96 ± 0.15	37.76 ± 0.17
1.735	35.61 ± 0.12	36.21 ± 0.19	36.99 ± 0.18	37.83 ± 0.15
1.985	35.59 ± 0.13	36.37 ± 0.08	36.98 ± 0.17	37.91 ± 0.23
2.235	35.75 ± 0.16	36.50 ± 0.20	37.19 ± 0.22	38.01 ± 0.14
2.485	35.79 ± 0.08	36.45 ± 0.13	37.28 ± 0.11	37.99 ± 0.23
2.735	36.01 ± 0.23	36.60 ± 0.15	37.24 ± 0.12	38.28 ± 0.21
2.985	36.01 ± 0.17	36.77 ± 0.20	37.36 ± 0.13	38.31 ± 0.13

Table B-20. Running times (in Mcycles @ 20 MHz) for Barnes-Hut (4,096 bodies) on 32 processors using the modified Alewife CRL implementation.

Bulk Transfer	One-way Message Latency (cycles)				
Cost (cycles/byte)	**264**	**314**	**364**	**414**	**464**
1.110	268.27 ± 0.34	271.19 ± 0.82	274.16 ± 0.72	277.06 ± 0.58	281.01 ± 0.44
1.235	269.74 ± 0.38	270.74 ± 0.41	274.89 ± 0.57	278.10 ± 0.59	281.32 ± 0.72
1.485	270.53 ± 0.97	271.18 ± 0.32	274.66 ± 0.41	278.39 ± 0.98	281.47 ± 0.51
1.735	270.27 ± 0.64	271.89 ± 0.42	274.87 ± 0.84	278.37 ± 0.69	281.96 ± 0.62
1.985	272.81 ± 0.57	272.32 ± 0.58	275.89 ± 0.74	278.53 ± 0.44	282.38 ± 0.66
2.235	275.87 ± 0.75	272.65 ± 0.70	276.26 ± 0.67	279.04 ± 0.37	282.23 ± 0.36
2.485	279.29 ± 0.37	272.72 ± 0.59	276.95 ± 0.61	279.45 ± 0.51	283.46 ± 0.72
2.735	278.99 ± 0.08	274.07 ± 0.58	277.15 ± 0.63	280.93 ± 0.56	284.25 ± 0.70
2.985	278.89 ± 0.37	274.20 ± 0.60	277.56 ± 0.67	280.85 ± 0.42	284.21 ± 0.32

Bulk Transfer	One-way Message Latency (cycles)			
Cost (cycles/byte)	**514**	**564**	**614**	**664**
1.110	284.26 ± 0.63	287.70 ± 0.49	290.73 ± 0.30	293.97 ± 0.60
1.235	284.53 ± 0.66	288.12 ± 0.37	291.78 ± 0.52	294.69 ± 0.74
1.485	284.74 ± 0.23	288.56 ± 0.30	291.36 ± 0.30	294.41 ± 0.60
1.735	284.91 ± 0.64	288.35 ± 0.50	292.56 ± 0.81	295.02 ± 0.76
1.985	285.66 ± 0.74	289.14 ± 0.55	292.04 ± 0.70	295.64 ± 0.65
2.235	286.13 ± 0.54	289.29 ± 0.56	293.26 ± 0.55	296.18 ± 0.61
2.485	286.51 ± 0.58	289.76 ± 0.46	293.23 ± 0.66	296.76 ± 0.32
2.735	287.14 ± 0.28	290.79 ± 0.73	293.95 ± 0.47	297.52 ± 1.03
2.985	287.56 ± 0.33	290.40 ± 0.32	294.02 ± 0.44	297.12 ± 0.56

Table B-21. Running times (in Mcycles @ 20 MHz) for Barnes-Hut (16,384 bodies) on 16 processors using the modified Alewife CRL implementation.

Bulk Transfer	One-way Message Latency (cycles)				
Cost (cycles/byte)	**264**	**314**	**364**	**414**	**464**
1.110	138.41 ± 0.41	139.95 ± 0.29	141.75 ± 0.29	143.63 ± 0.20	145.29 ± 0.27
1.235	138.44 ± 0.19	139.86 ± 0.25	141.91 ± 0.40	143.89 ± 0.38	145.80 ± 0.27
1.485	138.96 ± 0.32	140.38 ± 0.22	142.18 ± 0.44	144.18 ± 0.28	146.11 ± 0.28
1.735	138.85 ± 0.26	140.30 ± 0.43	142.13 ± 0.36	144.26 ± 0.42	146.17 ± 0.19
1.985	140.60 ± 0.44	140.63 ± 0.20	142.59 ± 0.24	144.55 ± 0.38	146.33 ± 0.30
2.235	142.44 ± 0.25	140.82 ± 0.47	142.72 ± 0.46	144.53 ± 0.35	146.85 ± 0.45
2.485	144.60 ± 0.48	140.99 ± 0.26	142.87 ± 0.24	144.87 ± 0.32	146.77 ± 0.28
2.735	144.41 ± 0.45	141.49 ± 0.33	143.62 ± 0.31	145.36 ± 0.40	147.06 ± 0.19
2.985	144.78 ± 0.34	141.68 ± 0.23	143.76 ± 0.30	145.20 ± 0.26	147.49 ± 0.27

Bulk Transfer	One-way Message Latency (cycles)			
Cost (cycles/byte)	**514**	**564**	**614**	**664**
1.110	147.34 ± 0.33	149.49 ± 0.48	151.42 ± 0.60	153.00 ± 0.44
1.235	147.73 ± 0.33	149.81 ± 0.48	151.36 ± 0.25	153.44 ± 0.27
1.485	147.82 ± 0.18	149.93 ± 0.29	151.77 ± 0.36	153.47 ± 0.34
1.735	147.91 ± 0.30	149.71 ± 0.43	151.91 ± 0.19	153.59 ± 0.27
1.985	148.24 ± 0.50	150.09 ± 0.31	152.05 ± 0.31	153.76 ± 0.25
2.235	148.54 ± 0.41	150.39 ± 0.36	152.37 ± 0.48	154.24 ± 0.25
2.485	148.73 ± 0.23	150.57 ± 0.44	152.66 ± 0.29	154.37 ± 0.29
2.735	149.36 ± 0.39	150.87 ± 0.37	152.77 ± 0.32	154.73 ± 0.41
2.985	149.30 ± 0.32	151.05 ± 0.28	152.87 ± 0.49	154.74 ± 0.30

Table B-22. Running times (in Mcycles @ 20 MHz) for Barnes-Hut (16,384 bodies) on 32 processors using the modified Alewife CRL implementation.

References

[1] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Ken Mackenzie, and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[2] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An Evolutionary Processor Design for Multiprocessors. *IEEE Micro*, pages 48–61, June 1993.

[3] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, pages 190–205, March 1992.

[4] Henri E. Bal and M. Frans Kaashoek. Object Distribution in Orca using Compile-Time and Run-Time Techniques. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, pages 162–177, September 1993.

[5] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the 38th IEEE Computer Society International Conference (COMPCON'93)*, pages 528–537, February 1993.

[6] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, pages 1526–1538, November 1989.

[7] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, pages 29–36, February 1995.

[8] Per Brinch Hansen. Concurrent Programming Concepts. *ACM Computing Surveys*, 5(4):223–245, 1973.

[9] David Callahan and Ken Kennedy. Compiling Programs for Distributed-Memory Multiprocessors. *Journal of Supercomputing*, pages 151–169, October 1988.

[10] Martin C. Carlisle, Anne Rogers, John H. Reppy, and Laurie J. Hendren. Early Experiences with Olden. In *Conference Record of the Sixth Workshop on Languages and Compilers for Parallel Computing*, August 1993.

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

[12] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-Based Cache-Coherence in Large-Scale Multiprocessors. *IEEE Computer*, pages 41–58, June 1990.

[13] David L. Chaiken and Anant Agarwal. Software-Extended Coherent Shared Memory: Performance and Cost. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 314–324, April 1994.

[14] Rohit Chandra, Kourosh Gharachorloo, Vijayaraghavan Soundararajan, and Anoop Gupta. Performance Evaluation of Hybrid Hardware and Software Distributed Shared Memory Protocols. In *Proceedings of the Eighth International Conference on Supercomputing*, pages 274–288, July 1994.

[15] Rohit Chandra, Anoop Gupta, and John L. Hennessy. Data Locality and Load Balancing in COOL. In *Proceedings of the Fourth Symposium on Principles and Practices of Parallel Programming*, pages 249–259, May 1993.

[16] Satish Chandra, James R. Larus, and Anne Rogers. Where is Time Spent in Message-Passing and Shared-Memory Programs? In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 61–73, October 1994.

[17] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[18] Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, and Willy Zwaenepoel. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 106–117, April 1994.

[19] Alan L. Cox and Robert J. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 32–44, December 1989.

[20] Alan L. Cox and Robert J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 98–108, May 1993.

[21] Sandhya Dwarkadas, Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 144–155, May 1993.

[22] S. J. Eggers and R. H. Katz. Evaluating the Performance of Four Snooping Cache Coherency Protocols. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, June 1989.

[23] Michael Jay Franklin. *Caching and Memory Management in Client-Server Database Systems*. PhD thesis, University of Wisconsin - Madison, July 1993.

[24] A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[25] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, June 1990.

[26] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer – Designing a MIMD Shared-Memory Parallel Machine. *IEEE Transactions on Computers*, pages 175–189, February 1983.

[27] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 262–273, October 1992.

[28] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, pages 549–557, October 1974.

[29] Waldemar Horwat. Concurrent Smalltalk on the Message-Driven Processor. Technical Report 1321, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, September 1991.

[30] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, pages 48–61, February 1988.

[31] Wilson C. Hsieh. *Dynamic Computation Migration in Distributed Shared Memory Systems*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1995.

[32] Wilson C. Hsieh, Paul Wang, and William E. Weihl. Computation Migration: Enhancing Locality for Distributed-Memory Parallel Systems. In *Proceedings of the Fourth Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 239–248, May 1993.

[33] Kirk L. Johnson, Joseph Adler, and Sandeep K. Gupta. CRL 1.0 Software Distribution, August 1995. Available on the World Wide Web at URL http://www.pdos.lcs.mit.edu/crl/.

[34] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-Performance All-Software Distributed Sharded Memory. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, December 1995.

[35] Vijay Karamcheti and Andrew Chien. Concert – Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware. In *Proceedings of Supercomputing '93*, pages 598–607, November 1993.

[36] Anna Karlin, Kai Li, Mark Manasse, and Susan Owicki. Empirical Studies of Competitive Spinning for A Shared-Memory Multiprocessor. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 41–55, October 1991.

[37] Pete Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwaenepoel. Memo: Distributed Shared Memory on Standard Workstations and Operating Systems. Technical Report TR93-206, Department of Computer Science, Rice University, June 1993.

[38] Pete Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.

[39] Kendall Square Research. KSR-1 Technical Summary, 1992.

[40] Alexander C. Klaiber and Henry M. Levy. A Comparison of Message Passing and Shared Memory Architectures for Data Parallel Programs. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 94–105, April 1994.

[41] K. Knobe, J. Lukas, and G. Steele Jr. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, pages 102–118, 1990.

[42] Leonidas I. Kontothanassis and Michael L. Scott. Software Cache Coherence for Large Scale Multiprocessors. In *Proceedings of the First Symposium on High-Performance Computer Architecture*, pages 286–295, January 1995.

[43] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory: Early Experience. In *Proceedings of the Fourth Symposium on Principles and Practice of Parallel Programming*, pages 54–63, May 1993.

[44] David A. Kranz. *ORBIT: An Optimizing Compiler for Scheme*. PhD thesis, Yale University, February 1988. Also available as Technical Report YALEU/DCS/RR-632.

[45] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *Proceedings of the International Conference on Supercomputing*, pages 195–206, July 1993.

[46] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[47] J. William Lee. Concord: Re-Thinking the Division of Labor in a Distributed Shared Memory System. Technical Report UW-CSE-93-12-05, University of Washington, December 1993.

[48] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, June 1992.

[49] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, pages 63–79, March 1992.

[50] Shun-tak Leung and John Zahorjan. Improving the Performance of Runtime Parallelization. In *Proceedings of the Fourth Symposium on Principles and Practice of Parallel Programming*, pages 83–91, May 1993.

[51] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, pages 361–376, July 1991.

[52] Kai Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the International Conference on Parallel Computing*, pages 94–101, 1988.

[53] Barbara H. Liskov. Distributed Programming in Argus. *Communications of the ACM*, pages 300–313, March 1988.

[54] Mark S. Manasse, Lyle A. McGeoch, and Daniel D. Sleator. Competitive Algorithms for On-line Problems. In *Proceedings of the 20th Annual Symposium on Theory of Computing*, pages 322–333, May 1988.

[55] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, May 1994.

[56] Ron Minnich, Dan Burns, and Frank Hady. The Memory-Integrated Network Interface. *IEEE Micro*, pages 11–20, February 1995.

[57] David Mosberger. Memory Consistency Models. *Operating Systems Review*, pages 18–26, January 1993.

[58] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 62–73, Octbober 1992.

[59] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, pages 134–154, February 1988.

[60] Rishiyur S. Nikhil. Cid: A Parallel, "Shared-memory" C for Distributed-Memory Machines. In *Proceedings of the Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, August 1994.

[61] Brian W. O'Krafka and A. Richard Newton. An Empirical Evaluation of Two Memory-Efficient Directory Methods. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 138–147, June 1990.

[62] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings of the International Conference on Parallel Processing*, pages 764–771, August 1985.

[63] Steve K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, April 1994.

[64] Anne Rogers and Keshav Pingali. Compiling for Distributed Memory Architectures. *IEEE Transactions on Parallel and Distributed Systems*, pages 281–298, March 1994.

[65] Anne Rogers, John H. Reppy, and Laurie J. Hendren. Supporting SPMD Execution for Dynamic Data Structures. In *Conference Record of the Fifth Workshop on Languages and Compilers for Parallel Computing*, August 1992. Also appears in Springer Verlag LNCS 757 (pp. 192-207).

[66] Edward Rothberg, Jaswinder Pal Singh, and Anoop Gupta. Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 14–25, May 1993.

[67] J. Saltz, R. Mirchandaney, and K. Crowley. Runtime Parallelization and Scheduling of Loops. *IEEE Transactions on Computers*, pages 603–612, May 1991.

[68] Harjinder S. Sandhu, Benjamin Gamsa, and Songnian Zhou. The Shared Regions Approach to Software Cache Coherence on Multiprocessors. In *Proceedings of the Fourth Symposium on Principles and Practices of Parallel Programming*, pages 229–238, May 1993.

[69] Daniel J. Scales and Monica S. Lam. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 101–114, November 1994.

[70] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, October 1994.

[71] Charles L. Seitz and Wen-King Su. A Family of Routing and Communication Chips Based on the Mosaic. In *Proceedings of the 1993 Symposium on Reseach on Integrated Systems*, pages 320–337, January 1993.

[72] Jaswinder Pal Singh, Anoop Gupta, and John L. Hennessy. Implications of Hierarchical N-Body Techniques for Multiprocessor Architecture. *ACM Transactions on Computer Systems*, pages 141–202, May 1995.

[73] Jaswinder Pal Singh, Anoop Gupta, and Marc Levoy. Parallel Visualization Algorithms: Performance and Architectural Implications. *IEEE Computer*, pages 45–55, July 1994.

[74] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, pages 5–44, March 1992.

[75] Alfred Z. Spector. Performing Remote Operations Efficiently on a Local Computer Network. *Communications of the ACM*, pages 246–260, April 1982.

[76] Per Stenström, Mats Brorsson, and Lars Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[77] Chandramohan A. Thekkath and Henry M. Levy. Limits to Low-Latency Communication on High-Speed Networks. *ACM Transactions on Computer Systems*, pages 179–203, May 1993.

[78] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Separating Data and Control Transfer in Distributed Operating Systems. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, October 1994.

[79] Chau-Wen Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, January 1993.

[80] Thorsten von Eicken, Anindya Basu, and Vineet Buch. Low-Latency Communication Over ATM Networks Using Active Messages. *IEEE Micro*, pages 46–53, February 1995.

[81] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[82] Deborah A. Wallach, Wilson C. Hsieh, Kirk L. Johnson, M. Frans Kaashoek, and William E. Weihl. Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation. In *Proceedings of the Fifth Symposium on Principles and Practices of Parallel Programming*, pages 217–226, July 1995.

[83] Wolf-Dietrich Weber and Anoop Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.

[84] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[85] David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, and Steven K. Reinhardt. Mechanisms for Cooperative Shared Memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–167, May 1993.

[86] Matthew J. Zekauskas, Wayne A. Sawdon, and Brian N. Bershad. Software Write Detection for a Distributed Shared Memory. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 87–100, November 1994.

[87] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 1988.