Protocol Optimizations for the CRL Distributed Shared Memory System

by

Sandeep K. Gupta

B.S., Electrical and Computer Engineering (1994)
B.A., Computer Science (1994)
B.A., Cognitive Sciences (1994)
Rice University

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

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1996

© Massachusetts Institute of Technology 1996. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
August 28, 1996

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

Accepted by... ....
R. Morgenthaler
Chairman, Department Committee on Graduate Theses

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

OCT 1 5 1996

Eng

LIBRARIES

Protocol Optimizations for the CRL Distributed Shared Memory System

by

Sandeep K. Gupta

Submitted to the Department of Electrical Engineering and Computer Science
on August 28, 1996, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Many optimizations and enhancements have been proposed for distributed shared memory (DSM) systems. This thesis describes the design, implementation, and performance of a C Region Library (CRL) port to the IBM SP/2 (CRL-SP/2). In addition, it discusses the design, implementation, and evaluation of a Three-Message-Invalidation optimization for CRL-SP/2, and a Floating-Home-Node optimization for CRL-SP/2. The experiments show that each optimization can provide applications with a significant improvement in running times. The Three-Message-Invalidation protocol decreased the 8-processor running time of the Traveling Salesman Problem by 30%, while the Floating-Home-Node protocol decreased that of Barnes-Hut by 56%.

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

Acknowledgments

Frans Kaashoek worked at least as hard as I did to complete this thesis. He helped me to find a topic, and participated in almost every part of the thesis process. During the last weeks of this thesis, Frans went out of his way to make sure I completed this thesis by the deadline. Thank you Frans for all you have done.

Kirk Johnson wrote CRL version 1.0, and without it, none of the work presented in this thesis could have been performed. Most of the figures and tables in this thesis are modified versions of those from Kirk's thesis. Without Kirk's permission to use them, I would have spent days just learning how to create them. Kirk was also very prompt in answering my CRL questions.

Thanks to the whole PDOS gang (Max, Greg, Dawson, Debby, Emmett, Josh, Anthony, David, Eddie, Hector, Robert, Tom, Wilson, and Joe) for providing such stimulating discussions on topics ranging from the sandboxing of code to the policies of nuclear disarmament.

A special thanks to Anthony for setting me up on my Thinkpad so that I could work on my thesis whenever an idea popped into my head. Josh, I really appreciate all the assistance you provided in formatting my thesis and finally in helping me overcome the last obstacle to my degree by printing my thesis and turning it in for me.

To Alex Schäffer, Willy Zwaenepoel, Bart Sinclair, and my other professors from Rice University, thank you for introducing me to research and giving me a great academic background.

I would like to thank the Office of Naval Research for awarding me with a fellowship to attend graduate school.

To my closest friends for the past two years, Chandanbhai, Dhara, Rakhi, Srikant, Kanchanbhai, Hemantibhabi, and the whole HSC, thank you for your support and encouragement in finishing this thesis.

Thank you, God. I know this thesis and the results would not have been possible without one of your miracles!

Mom, Dad, and Puneet, you have always been there for me. Whenever I needed your advice or I just needed to talk to somebody, I could always call you up and have things straightened out. I love you!

To my dearest Prabha, this thesis has been a part of our marriage since day one. Rather than spending many hours with you, I have been working to finish this thesis. Still, you have been very understanding and helped me by taking care of the rest of my life. Now the thesis is complete.

To Mom and Dad.

Contents

1 Introduction **19**

1.1 Contributions of This Thesis . . . 19

1.2 Background of Distributed Shared Memory . . . 20

1.3 Thesis Outline . . . 23

2 CRL Overview **25**

2.1 Description of CRL . . . 25

2.2 Goals of CRL . . . 26

2.3 Programming with CRL . . . 26

3 CRL Applications **29**

3.1 Traveling Salesman Problem . . . 29

3.2 Blocked LU . . . 29

3.3 Water-Nsquared . . . 30

3.4 Barnes-Hut . . . 30

4 An Implementation of CRL on the SP/2 **33**

4.1 The Message Passing Library . . . 33

4.1.1 Overview of MPL Functions . . . 34

4.1.2 Shortcomings of MPL . . . 35

4.2 CRL Region Version Numbers . . . 37

4.2.1 MsgRelease Messages . . . 37

4.2.2 Multiple Read-Only Copies . . . 38

4.3 SP/2 Specific Optimizations . . . 40

4.3.1 Nonurgent Messages . . . 40

4.3.2 Two Message Data Transfer . . . 41

4.3.3 Simultaneous Data Transfers of a Region . . . 43

4.4 CRL-SP/2 Performance . . . 43

4.4.1 MPL Round Trip Operations, Events and Base Operations 44
4.4.2 Base CRL Operations 45
4.4.3 Application Performance 47

5 Three-Message-Invalidation Protocol 51
5.1 Protocol Description 51
5.2 Why the Protocol Decreases Execution Time 53
5.3 Protocol Implementation in CRL 54
5.3.1 Transferring Version Information 55
5.3.2 Three-Message-Invalidation Completion 55
5.3.3 Handling Flushes 55
5.4 Performance Results 56
5.4.1 Three-Message-Invalidation Benchmark 56
5.4.2 Base CRL Operations 56
5.4.3 Application Performance 57

6 Floating-Home-Node Protocol 59
6.1 Why the Protocol Decreases Execution Time 59
6.2 Protocol Implementation in CRL 60
6.3 Messages in Floating-Home-Node Protocol 61
6.3.1 Simultaneous Become-Home Requests 61
6.3.2 Number of Forwarded Messages 63
6.4 Performance Results 63
6.4.1 Floating-Home-Node Benchmark 63
6.4.2 Base CRL Operations 64
6.4.3 Barnes-Hut Performance 64

7 Related Work 67

8 Conclusion 71

A Raw Experimental Data 73
A.1 IBM Message Passing Library (MPL) versus Message Passing Interface (MPI) 73
A.2 CRL Operation Latencies 75
A.3 CRL and MPL Event Counts 79
A.3.1 CRL-SP/2 Application Events 79
A.3.2 Three-Message-Invalidation Application Events 84
A.3.3 Floating-Home-Node Application Events 89

B Implementation Details of the Three-Message-Invalidate Protocol **95**

B.1 Protocol States and Events 95

B.2 Home-Side State Machine 97

B.3 Remote-Side State Machine 116

C Implementation Details of the Floating-Home-Node Protocol **129**

C.1 Protocol States and Events 129

C.2 Home-Side State Machine 131

C.3 Remote-Side State Machine 155

List of Figures

4-1 Messages sent in the original (4 message) data transfer protocol. 41
4-2 Messages sent in the optimized (2 message) data transfer protocol. 42

5-1 Messages sent in the original (4 message) invalidation protocol. 52
5-2 Messages sent in the optimized (3 message) invalidation protocol. 53

6-1 Messages sent during a become home request. 62

B-1 HomeExclusive: state transition diagram. 100
B-2 HomeExclusiveRip: state transition diagram. 101
B-3 HomeExclusiveWip: state transition diagram. 102
B-4 HomeShared: state transition diagram. 103
B-5 HomeSharedRip: state transition diagram. 104
B-6 HomeIip: state transition diagram. 105
B-7 HomeIipSpecial: state transition diagram. 106
B-8 HomeInvalid: state transition diagram. 107
B-9 HomeThreeWayIipSpecial: state transition diagram. 108
B-10 RemoteInvalid: state transition diagram. 117
B-11 RemoteInvalidReq: state transition diagram. 118
B-12 RemoteShared: state transition diagram. 119
B-13 RemoteSharedReq: state transition diagram. 120
B-14 RemoteSharedRip: state transition diagram. 121
B-15 RemoteModified: state transition diagram. 122
B-16 RemoteModifiedRip: state transition diagram. 123
B-17 RemoteModifiedWip: state transition diagram. 124

C-1 HomeExclusive: state transition diagram. 135
C-2 HomeExclusiveRip: state transition diagram. 136
C-3 HomeExclusiveWip: state transition diagram. 137

C-4 HomeShared: state transition diagram. 138
C-5 HomeSharedRip: state transition diagram. 139
C-6 HomeIip: state transition diagram. 140
C-7 HomeIipSpecial: state transition diagram. 141
C-8 HomeInvalid: state transition diagram. 142
C-9 HomeThreeWayIipSpecial: state transition diagram. 143
C-10 HomeBecomingRemote: state transition diagram. 144
C-11 RemoteInvalid: state transition diagram. 157
C-12 RemoteInvalidReq: state transition diagram. 158
C-13 RemoteShared: state transition diagram. 159
C-14 RemoteSharedReq: state transition diagram. 160
C-15 RemoteSharedRip: state transition diagram. 161
C-16 RemoteModified: state transition diagram. 162
C-17 RemoteModifiedRip: state transition diagram. 163
C-18 RemoteModifiedWip: state transition diagram. 164
C-19 RemoteBecomeHomeReq: state transition diagram. 165

List of Tables

4.1 Time to perform MPL round trip operations. 44

4.2 Time to perform base MPL events. 45

4.3 Measured CRL latencies, nodes polling, partial list. 45

4.4 Measured CRL latencies, nodes not polling, partial list. 47

4.5 Measured application execution time using CRL-SP/2. 48

4.6 Calculated application speedup using CRL-SP/2. 48

4.7 Expensive events per second using CRL-SP/2 on 8 processors. 48

5.1 Measured CRL latencies for Three-Message-Invalidation, nodes polling, partial list. . 57

5.2 Measured CRL latencies for Three-Message-Invalidation, nodes not polling, partial list. 57

5.3 Measured application execution time using Three-Message-Invalidation. 57

5.4 Calculated application speedup using Three-Message-Invalidation. 58

6.1 Comparison of measured Barnes-Hut execution times for each CRL-SP/2 version. . . 64

6.2 Comparison of calculated Barnes-Hut speedups for each CRL-SP/2 version. 65

A.1 Switch performance using MPL (user space, application space to application space) . 73

A.2 Switch performance using MPL (udp/IP, application space to application space) . . 74

A.3 Preliminary switch performance using MPI (user space, application space to application space) 74

A.4 Preliminary switch performance using MPI (udp/IP, application space to application space) 74

A.5 Events measured by latency microbenchmark. 76

A.6 CRL-SP/2 latencies with nodes polling 77

A.7 CRL-SP/2 latencies with nodes not polling 77

A.8 Three-Message-Invalidation latencies with nodes polling 78

A.9 Three-Message-Invalidation latencies with nodes not polling 78

A.10 CRL and MPL event counts for Barnes-Hut on CRL-SP/2. 80

A.11 CRL and MPL event counts for Block LU on CRL-SP/2. 81

A.12 CRL and MPL event counts for TSP on CRL-SP/2. 82
A.13 CRL and MPL event counts for Water on CRL-SP/2. 83
A.14 CRL and MPL event counts for Barnes-Hut on Three-Message-Invalidation. 85
A.15 CRL and MPL event counts for Block LU on Three-Message-Invalidation. 86
A.16 CRL and MPL event counts for TSP on Three-Message-Invalidation. 87
A.17 CRL and MPL event counts for Water on Three-Message-Invalidation. 88
A.18 CRL and MPL event counts for Barnes-Hut on Floating-Home-Node. 90
A.19 CRL and MPL event counts for Blocked LU on Floating-Home-Node. 91
A.20 CRL and MPL event counts for TSP on Floating-Home-Node. 92
A.21 CRL and MPL event counts for Water on Floating-Home-Node. 93

B.1 CRL home-side protocol states. 96
B.2 CRL remote-side protocol states. 96
B.3 CRL call events. 97
B.4 CRL home-to-remote protocol messages. 97
B.5 CRL remote-to-home protocol messages. 97
B.6 CRL remote-to-requesting protocol messages. 98
B.7 HomeExclusive: protocol events and actions. 109
B.8 HomeExclusiveRip: protocol events and actions. 109
B.9 HomeExclusiveWip: protocol events and actions. 110
B.10 HomeShared: protocol events and actions. 110
B.11 HomeSharedRip: protocol events and actions. 111
B.12 HomeIip: protocol events and actions. 111
B.13 HomeIipSpecial: protocol events and actions. 112
B.14 HomeInvalid: protocol events and actions. 113
B.15 HomeInvalid: protocol events and actions (continued). 114
B.16 HomeThreeWayIipSpecial: protocol events and actions. 115
B.17 RemoteInvalid: protocol events and actions. 125
B.18 RemoteInvalidReq: protocol events and actions. 125
B.19 RemoteShared: protocol events and actions. 126
B.20 RemoteSharedReq: protocol events and actions. 126
B.21 RemoteSharedRip: protocol events and actions. 127
B.22 RemoteModified: protocol events and actions. 127
B.23 RemoteModifiedRip: protocol events and actions. 128
B.24 RemoteModifiedWip: protocol events and actions. 128

C.1 CRL home-side protocol states. 130

C.2 CRL remote-side protocol states. 130
C.3 CRL call events. 131
C.4 CRL home-to-remote protocol messages. 131
C.5 CRL remote-to-home protocol messages. 131
C.6 CRL remote-to-requesting protocol messages. 132
C.7 CRL original-home protocol messages. 132
C.8 HomeExclusive: protocol events and actions. 145
C.9 HomeExclusiveRip: protocol events and actions. 146
C.10 HomeExclusiveWip: protocol events and actions. 146
C.11 HomeShared: protocol events and actions. 147
C.12 HomeShared: protocol events and actions (continued). 148
C.13 HomeSharedRip: protocol events and actions. 148
C.14 HomeIip: protocol events and actions. 149
C.15 HomeIipSpecial: protocol events and actions. 150
C.16 HomeInvalid: protocol events and actions. 151
C.17 HomeInvalid: protocol events and actions (continued). 152
C.18 HomeThreeWayIipSpecial: protocol events and actions. 153
C.19 HomeBecomingRemote: protocol events and actions. 154
C.20 RemoteInvalid: protocol events and actions. 166
C.21 RemoteInvalidReq: protocol events and actions. 167
C.22 RemoteShared: protocol events and actions. 168
C.23 RemoteSharedReq: protocol events and actions. 169
C.24 RemoteSharedRip: protocol events and actions. 169
C.25 RemoteModified: protocol events and actions. 170
C.26 RemoteModifiedRip: protocol events and actions. 170
C.27 RemoteModifiedWip: protocol events and actions. 171
C.28 RemoteBecomeHomeReq: protocol events and actions. 171

Chapter 1

Introduction

We all would like our applications to run more quickly. One way to achieve this goal is to simply buy a faster machine. However, the previous progress made in the hardware area will not continue forever because we are reaching physical limits such as the speed of light, and such limitations will hinder future progress in this area.

Instead of executing the application sequentially on a single machine, another solution is to run it on multiple processors in parallel by simultaneously performing a fraction of the total computation on each node, and then communicating to synchronize and/or combine the results of each part of the computation. If little communication is necessary to perform the computation, or the movement of data among processors is regular, then the communication can easily be performed by passing messages among nodes. This form of communication is commonly known as Message Passing. However, in complex systems with a large amount of communication or with communication that cannot be determined before runtime, it is much easier to use a Distributed Shared Memory (DSM) system.

During the past ten years, a lot of research has been performed to find ways to improve the speedup of parallel programs, and to improve their ease of use [29]. In particular, DSM systems and many of the tradeoffs involved in their implementations have been actively examined. This thesis discusses the design, implementation, and evaluation of a C Region Library port to the IBM SP/2. In addition, it discusses the design, implementation, and evaluation of two optimizations to the original CRL protocol, a Three-Message-Invalidation protocol [26] and a Floating-Home-Node protocol [35].

1.1 Contributions of This Thesis

The C Region Library (CRL) [21] is an all-software region-based DSM system that provides applications with the ability to specialize the communication protocols used for synchronization, and transfer of shared regions.

This thesis has a number of contributions. The first is a design, implementation, and evaluation of CRL version 1.0 for the IBM SP/2. The evaluation specifically looks at how the message passing performance of the SP/2 affects the speedup of CRL applications. Measurements show that the message passing latency and the overhead of the IBM SP/2 is too great to allow for acceptable application speedups. Optimizations, such as the two that follow, can significantly improve application running times on the SP/2 implementation of CRL (CRL-SP/2).

The second contribution is an improvement in the performance of CRL-SP/2 by designing and implementing a Three-Message-Invalidation protocol, which is a well-known optimization and enhancement [26]. Evaluation of the Three-Message-Invalidation protocol shows that it reduces the number of messages in an invalidation. This enhancement reduces the number of messages sent by up to 25% and thus improves application running times significantly on CRL-SP/2. The results show that for some applications, this optimization reduces execution time by 30%.

The third is a design and implementation of a Floating-Home-Node protocol, and an evaluation of the potential benefits of using this protocol. The Floating-Home-Node protocol provides the ability to change the node responsible for the coherence of a region's data. A similar type of protocol where data migrates is used in COMA machines [35]. The idea of examining such a protocol on CRL appeared in [18]. The results in this thesis show that the Floating-Home-Node protocol decreases the execution time of Barnes-Hut by over 50%.

1.2 Background of Distributed Shared Memory

This section describes why DSM is interesting and why it is worth exploring the possibilities of improving it. There are some tradeoffs involved that help determine when message passing should be used in parallelizing a program, and when DSM should be used.

An example where message passing is a clear win is the multiplication of dense matrices represented by two-dimensional arrays. If the problem is to multiply matrices A and B to produce matrix C, then matrix C can be equally partitioned into rectangles, and each rectangle can be assigned to a node participating in the computation. If the assigning of partitions is done statically by node number, then the node(s) containing the elements of matrices A and B can directly send to each node exactly that data that the node needs from A and B. After the multiplication is complete, then each node can send exactly those elements of C that it calculated to the destination node(s) where the result is to be kept.

However, many programs written today are much too complicated to precisely determine which processor has a valid copy of a particular piece of data. Additionally, the bookkeeping involved in irregular computations, such as multiplying two general sparse matrices, is too great a burden to put on the programmer. Of course the processors could be allocated equally sized regions of the

sparse matrix product, but then the processors may not each do an equal amount of work and the improvements from parallelizing the computation will be diminished.

Another good example of an irregular computation is a graph traversal where the values at a node of the graph are updated to the value of functions of the values at the nodes adjacent to it. In order to distribute this algorithm over a network of workstations, the programmer would have to implement a mechanism for nodes to get the necessary data at the appropriate time. This would involve nodes knowing who to ask for the data, and having the ability to have exclusive access to the data during writes while at the same time allowing multiple readers during reads for efficiency. Implementing these structures and protocols is very complicated and very time consuming in terms of debugging time.

Another possible solution to this complex data management is to have a centralized server for the shared data that would take care of all the bookkeeping. The problem here is that every node must go through this server to access data and the server becomes a bottleneck. Normally, this server would then be a more powerful machine, when compared with the client nodes. When this happens, the system model has basically become a database. However in many networks, it is common to have many machines of equivalent power. In such systems, it makes more sense to distribute the server onto each client, and thus each machine is both a server and a client.

When the shared data is distributed in such fashion over all the clients, and the data can be accessed in a way similar to local memory, then we have a DSM system. DSM systems handle all of the data bookkeeping so that applications are able to access the data almost as if they are accessing local data in their local memory. It is essentially an abstraction of a global shared memory. DSM implementations that are mostly software are implemented over some message passing mechanism [3, 21, 22, 33] while ones that are mostly hardware are usually implemented by using a fixed mapping from global address to node number, and then storing the information at that node [2, 25].

In DSM systems, applications are not required to keep track of which processor has a valid copy of the data, where the data is located locally in memory, and which processors are allowed to read from/write to the data. Because the DSM system takes care of all this information and moreover provides a coherency mechanism to keep the data consistent, it is widely accepted that it is much easier to write an application in DSM style than it is to write one in message passing style. The DSM paradigm saves programming and debugging time.

But if DSM is so great, why isn't everyone using this programming methodology? One reason is that many programmers still do not have a choice. Implementations of DSM that use specialized hardware cannot be used in a heterogeneous environment, and they tend to be expensive.

Most modern operating systems such as Windows/NT, SunOS, Solaris, and Ultrix come readily available with message passing libraries. However, if one wants to use DSM, the DSM library needs to be installed and added separately, at some inconvenience to the programmer. The most difficult

part about the installation is actually finding source or object files for the DSM system. And then, after finding the source, it may have to be ported to the appropriate network, CPU hardware, and OS combination.

TreadMarks [22] is one example of a DSM system that was written on standard UNIX systems, such as SunOS and Ultrix, at user-level. Because it is at user-level, TreadMarks does not require any modifications to the operating system. TreadMarks has a few differences though from CRL. Data coherency is done on a virtual memory page granularity, which is a fixed size set by the operating system virtual memory or hardware (set to 4 kbytes on many UNIX systems). If a shared data item is smaller than this size and a data access miss occurs on the shared data item, then data for other items may be sent in addition to the desired data, increasing access miss latency and using up bandwidth. If a shared data item is larger than this size and a data access miss occurs on the shared data item, then multiple data access misses will occur and the total overhead paid will be larger than if only one access miss occurred. Another difference is that since TreadMarks uses expensive operating system traps for maintaining data coherency, a lot of overhead is paid for each data access miss.

The availability of message passing libraries that are available free of charge for a wide range of operating systems and hardware platforms is much greater than the availability of such DSM libraries. For example, PVM [36] and MPI [31] are two message passing libraries that are in wide spread use on various operating systems and hardware platforms. PVM itself has thousands of users [32]. Also, PVM applications executing on different platforms can transfer messages to each other, and any necessary data conversion involving byte order or data-type size is performed transparently to the PVM user. So PVM is extremely useful and easy to use in a heterogeneous environment.

Another reason why DSM is not in widespread use, and possibly the reason why implementations of DSM are difficult to find, is that DSM applications tend to be less efficient than optimized message passing applications. In papers comparing the execution time of message passing programs to that of DSM programs, message passing programs usually perform better [30, 28, 4].

One observation should be made explicitly clear: For a given platform, the absolute optimal performance attainable by an all-software DSM system is inferior to that attainable by a message passing system for the simple reason that the software DSM implementation will have to use the underlying message passing system for communication among the nodes in the system.

There are some cases when DSM performs better than message passing though. For example in Kevin Lew's master's thesis [27], for a particular problem size where there was little memory contention, DSM performed better than message passing on the Alewife machine [1]. In this case the better performance of the DSM application was attributed to the fact that the hardware platform had hardware support for DSM when only a few nodes shared the data. In this scenario, when a request arrived at a node for shared data, the request could be handled by a specialized processor in the node,

in parallel with the normal executing thread. However, if a request arrived via message passing, then the executing thread would have to be interrupted for the request to be handled and some overhead would be incurred. When more nodes shared the data, the message passing application performed better than the DSM application, since in this case, DSM requests were also handled in software.

Even though implementations of DSM in software are less efficient than those in hardware, software has some advantages over hardware. The foremost reason is that hardware is HARDware; it is not malleable to the specific needs of an application, and cannot be ported. One can port software DSMs to newer platforms quickly [21]. And once the initial development costs have been paid, the incremental costs for updates and porting of a software DSM system is minimal. The costs for a hardware system are quite high, and when the specialized hardware has been developed, it can only be used on a particular CPU platform. Thus, software can also support heterogeneous environments much more easily.

Most importantly, a software DSM system's protocols can be altered and optimized to the needs of a particular application running on top of it [21, 9, 13, 5] without adding much complexity to the DSM system. For example, it would be very difficult to add the Three-Message-Invalidation and Floating-Home-Node optimizations to an existing hardware DSM system. In a software DSM system, fixed policies such as prioritizing shared memory requests from nodes and determining which regions should and should not be cached can be specialized with a few changes to the application running on top of the DSM. One could even specialize to adaptable techniques that would examine access patterns in the system and calculate how to set the policies for the system.

1.3 Thesis Outline

This thesis contains seven more chapters followed by three appendices. Chapter 2 contains a brief overview of CRL and its usage. Chapter 3 contains a brief description of the applications used to evaluate the performance of CRL. Chapter 4 describes the CRL port to the IBM SP/2 and contains an analysis of CRL application speedup on the IBM SP/2. Chapter 5 provides a description and analysis of the Three-Message-Invalidation protocol implemented for CRL. Chapter 6 contains a description and analysis of CRL's Floating-Home-Node protocol implementation. A description of work related to this thesis appears in Chapter 7. Chapter 8 is the conclusion of this thesis. Appendix A contains the Raw Data used in the analysis. Appendix B describes the Three-Message-Invalidation protocol implementation in great detail, including state diagrams and pseudocode. Appendix C describes the Floating-Home-Node protocol implementation in great detail, including state diagrams and pseudocode.

Chapter 2

CRL Overview

This chapter provides a brief overview of CRL, the C Region Library. Most of the following text is taken directly from the CRL User Documentation [20]. For more information on CRL, consult [19, 21, 18].

2.1 Description of CRL

The C Region Library (CRL) is an all-software distributed shared memory (DSM) system intended for use on message-passing multicomputers and distributed systems. Parallel applications built on top of CRL share data through regions. Each region is an arbitrarily sized, contiguous area of memory. The programmer defines regions, and includes annotations to delimit accesses to those regions. Regions are cached in the local memories of processors; cached copies are kept consistent using a directory-based coherence protocol. Because coherence is provided at the granularity of regions instead of memory pages, cache lines, or some other arbitrarily chosen fixed-size unit, CRL avoids the concomitant problems of false sharing for coherence units that are too large or inefficient use of bandwidth for coherence units that are too small.

Three key features distinguish CRL from other software DSM systems. First, CRL is system and language-independent. Providing CRL functionally in programming languages other than C should require little work. Second, CRL is portable. By employing a region-based approach, CRL is implemented entirely as a library and requires no functionality from the underlying hardware, compiler, or operating system beyond that necessary to send and receive messages. Third, CRL is efficient. Very little software overhead is interposed between applications and the underlying message-passing mechanisms. While these features have occurred in isolation or in tandem in other software DSM systems, CRL is the first software DSM system to provide all three in a simple, coherent package.

Because shared address space or shared memory programming environments like CRL provide

a uniform model for accessing all shared data, whether local or remote, they are relatively easy to use. In contrast, message-passing environments burden programmers with the task of orchestrating all interprocessor 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 relatively simple communication patterns), the task can be far more difficult for large, complex applications, particularly those in which data is shared at a fine granularity or according to irregular, dynamic communication patterns.

In spite of this fact, message passing environments such as PVM [36, 10, 32] and MPI [31] are often the de facto standards for programming multicomputers and networks of workstations. We believe that this is primarily due to the fact that these systems require no special hardware, compiler, or operating system support, thus enabling them to run entirely at user level on unmodified, "stock" systems. Because CRL also requires minimal support from the underlying system, it should be equally portable and easy to run on different platforms. As such, we believe that CRL should serve as an excellent vehicle for applications requiring more expressive programming environments than those provided by PVM or MPI.

2.2 Goals of CRL

Several major goals guided the development of CRL. 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 what functionality was 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) are worthy of study, but can suffer a performance penalty because of inefficient interfaces for accessing such features. 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.

2.3 Programming with CRL

Applications using CRL can affect interprocessor communication and synchronization either through a relatively typical set of global operations (barrier, broadcast, reduce) or through operations on regions.

A region is an arbitrarily sized, contiguous area of memory identified by a unique region identifier. A region identifier is a portable and stable name for a region; in order to access a region, a processor node must know the region's region identifier. New regions can be created dynamically with a function call similar to `malloc()`, and can later be removed when they are no longer needed. Blocks of memory comprising distinct regions (those with different region identifiers) are nonoverlapping.

The operations on the regions are reading and writing. All accesses to shared region data must be delimited by CRL function calls. These calls are used to indicate to CRL exactly which region data is being accessed so that consistency for that region can be performed. The functions are:

- **`rgn_start_read`**, which is used before reading data.
- **`rgn_end_read`**, which is used after reading data.
- **`rgn_start_write`**, which is used before writing data.
- **`rgn_end_write`**, which is used after writing data.

If access has been granted for writing data, then the application also has permission to read the data. Each of the above functions takes a region data pointer as the sole argument. Also, reads and writes to shared data may be grouped such that only one start and end function call needs to be performed for the whole group.

Chapter 3

CRL Applications

This chapter briefly describes the applications used to measure the performance of CRL-SP/2. The applications are the Traveling Salesman Problem (TSP), Blocked LU (LU), Water-Nsquared (Water), and Barnes-Hut (Barnes). LU, Water, and Barnes are taken from the SPLASH-2 parallel application suite [38] and their descriptions are taken directly from [38]. Some of the text regarding the communication and computation granularity of LU, Water, and Barnes was taken directly from [18].

3.1 Traveling Salesman Problem

The Traveling Salesman Problem solves the NP-complete problem of finding the shortest cyclic path going through every node of a graph. In this implementation, one master node creates a job queue of possible solutions, and the rest of the nodes become slaves. While the job queue still contains jobs, the slaves continue to grab the next possible solution off of the queue and to explore that possible solution. A shared counter is used to keep track of the next available job. This shared counter causes 99% of the communication in TSP and is the bottleneck for TSP.

TSP is an interesting application to measure because its computation granularity is relatively course-grained, but at the same time TSP requires low latency access to the shared counter. The results for TSP presented in this thesis are obtained for a problem size of 14 cities.

3.2 Blocked LU

The LU kernel factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The dense $n \times n$ matrix A is divided into an $N \times N$ array of $B \times B$ blocks ($n = NB$) to exploit *temporal* locality on submatrix elements. To reduce communication, block ownership is assigned using a 2-D scatter decomposition, with a block being updated by the processor that owns

it. Elements within a block are allocated contiguously to improve spatial locality benefits, and blocks are allocated locally to processors that own them.

After the home node writes to a region, all the requests for that region are Shared Requests. So no invalidations are ever sent in LU, and all requests can be responded to immediately. LU's computation granularity is fairly large, so LU is very close to a best-case application.

The results for LU presented in this thesis are for a 1000x1000 matrix using 20x20 blocks.

3.3 Water-Nsquared

This application evaluates forces and potentials that occur over time in a system of water molecules. The forces and potentials are computed using an $O(n^2)$ algorithm, and a predictor-corrector method is used to integrate the motion of the water molecules over time. A process updates a local copy of the particle accelerations as it computes them, and accumulates into the shared copy once at the end.

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). The computation granularity of Water is much smaller than that of LU, so it provides a more challenging workload for CRL. The results for Water presented in this thesis are for a problem size of 512 molecules.

3.4 Barnes-Hut

The Barnes application simulates the interaction of a system of bodies (galaxies or particles, for example) in three dimensions over a number of time-steps, using the Barnes-Hut hierarchical N-body method. It represents the computational domain as an octree with leaves containing information on each body, and internal nodes representing space cells. Most of the time is spent in partial traversals of the octree (one traversal per body) to compute the forces on individual bodies. The communication patterns are dependent on the particle distribution and are quite unstructured. No attempt is made at intelligent distribution of body data in main memory, since distribution is difficult at page granularity and not very important to performance.

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.

Barnes-Hut's computation granularity is even more fine-grained than Water's, providing even more of a challenge for CRL. Moreover, the problem size used for this thesis (16384 bodies) causes the CRL Unmapped Region Cache to overflow (cache size is 1024 regions for this thesis). The overflowing cache results in much more communication caused by region flushes and start operation misses.

Chapter 4

An Implementation of CRL on the SP/2

The RS/6000 Scalable POWERparallel Systems 2 (SP/2) is IBM's general purpose scalable supercomputer [14]. The SP/2 connects up to 512 RS/6000 processors using IBM's low-latency (39.2 microseconds on a 66MHz processor) and high-bandwidth (35.6 megabytes per second point-to-point on a 66MHz processor) SP Switch [8].

The SP/2 implementation of CRL (from now on called CRL-SP/2) uses IBM's Message Passing Library (MPL) [17], which is part of the IBM AIX Parallel Environment [15]. Another message passing library option available on the SP/2 is the Message Passing Interface (MPI) [31]. However, the performance of MPL is superior to MPI [8] as can be seen in appendix Section A.1. Thus MPL is used in lieu of MPI.

The following sections contain a brief description of MPL's functions and some of MPL's shortcomings, a description of the changes made to the assigning of region version numbers, a description of the SP/2-specific optimizations for the implementation of CRL, and a performance evaluation of CRL-SP/2.

4.1 The Message Passing Library

To better understand the specifics of CRL-SP/2, one needs to understand the precise semantics of the key MPL functions used in the implementation. The following subsections contain a short description of key MPL functions, and then explain how the semantics of those functions did not match with what was desired.

The Message Passing Library (MPL) is IBM's message passing library for the SP/2. This library

includes functions that support point-to-point communication, creation and modification of task groups, and collective communication within task groups.

MPI is another message passing library available on the IBM SP/2. However, on the version of the IBM AIX Parallel Environment available for this research, the MPI library is written using the MPL library, so the MPL library functions have higher data bandwidth and lower message latency.

On the SP/2 mailing lists, a message was sent indicating that a newer version of the MPL library and AIX operating system were available and that the combination of both improved the performance of applications using MPL. Unfortunately, the system used for this research is using AIX version 3.2 and does not have the newer versions installed. It would be interesting to examine the performance of CRL and the CRL applications on these newer versions and to analyze if the implementation tradeoffs differ between the older versions and the newer versions.

4.1.1 Overview of MPL Functions

This section contains a brief description of the key MPL functions. For a more in-depth description of MPL, consult the IBM AIX Parallel Environment Parallel Programming Subroutine Reference [16]. The key functions are **mpc_send**, **mpc_recv**, **mpc_rcvncall**, **mpc_status**, and **mpc_wait**.

```
int mpc_send(const void *outmsg, size_t msglen, int dest, int type,
                              int *msgid);
```

Asynchronous send.
outmsg specifies the output buffer.
msglen specifies the buffer length in bytes.
dest specifies the destination node for the message.
type specifies the application defined type for the message. The type can be used to determine what to do with the message.
after **mpc_send** has completed, **msgid** contains an identifier that can be used to query for the current status of the send via **mpc_status**, or can be used to postpone further execution until the send has completed via **mpc_wait**.

```
int mpc_recv(void *inmsg, size_t msglen, int *source, int *type, int *msgid);
```

Asynchronous receive for a message matching a particular source and a particular type.
inmsg specifies the input buffer.
msglen specifies the buffer length in bytes.
source specifies which node the message should come from. **source** may also have the wildcard value **DONTCARE**, in which case a message will be accepted from any node. After the **mpc_recv** call completes, **source** will contain the actual sending node identifier.
type specifies which application defined type the incoming message should have. **type** may also have the wildcard value **DONTCARE**, in which case a message of any type will be accepted. After the **mpc_recv** call completes, **type** will contain the actual type of the message.
after **mpc_send** has completed, **msgid** contains an identifier that can be used to query for the current status of the receive via **mpc_status**, or can be used to postpone further execution until the receive has completed via **mpc_wait**.

```
int mpc_rcvncall(void *inmsg, size_t msglen, int *source, int *type,
                              int *msgid, void (func*)());
```

Asynchronous receive for a message matching a particular source and a particular type. When the message has arrived, the specified handler is called.

`inmsg` specifies the input buffer.

`msglen` specifies the buffer length in bytes.

`source` specifies which node the message should come from. `source` may also have the wildcard value `DONTCARE`, in which case a message will be accepted from any node. After the `mpc_recv` call completes, `source` will contain the actual sending node identifier.

`type` specifies which application defined type the incoming message should have. `type` may also have the wildcard value `DONTCARE`, in which case a message of any type will be accepted. After the `mpc_recv` call completes, `type` will contain the actual type of the message.

`func` specifies which function to call after the message has arrived. The function is passed the message identifier of the message.

after `mpc_send` has completed, `msgid` contains an identifier that can be used to query for the current status of the receive via `mpc_status`, or can be used to postpone further execution until the receive has completed via `mpc_wait`.

```
int mpc_status(int msgid);
```

Query for message status.

`msgid` specifies the message for which the query is being performed.

The return code specifies whether the message has been sent/received, or is still pending.

```
int mpc_wait(int *msgid, size_t *nbytes);
```

Postpone further execution until message operation is complete.

`msgid` specifies the message for which the wait is being performed. `msgid` may also have the wildcard value `DONTCARE`, in which case execution is postponed until any asynchronous message operation completes. After the `mpc_wait` call completes, `msgid` will contain the actual message identifier for the message that completed.

after `mpc_wait` has completed, if the message waited upon was a receive, then `nbytes` contains the number of bytes received in the message.

4.1.2 Shortcomings of MPL

MPL has a few shortcomings that provided obstacles and challenges for the implementation of CRL-SP/2. This section describes the following three shortcomings and explains what was done to overcome the shortcomings:

1. When a message arrives and a message handler is called, the message handler's only argument is a message identifier

2. No polling function exists that both accepts wildcards and is nonblocking

3. Every MPL function is uninterruptible, and thus messages arriving during MPL function calls get blocked

Argument to Message Handlers

`mpc_rcvncall`, briefly described in Section 4.1.1, is the only function available for associating a handler with a particular message type. When a message of the correct type arrives, the handler

executes. The only argument given to the handler is the message identifier of the message. This message identifier matches the return value of the original `mpc_rcvncall` function call.

Only having the message identifier makes the job of the message handler more difficult since the message handler must somehow determine where the message buffer is stored, based solely on the message identifier. Many message passing systems, including Berkeley's Active Message Library [37], provide for a user-settable one word argument to the message handler. This one word argument may not sound like a better idea then the message identifier, but there are a couple of important differences:

- The message identifier is set by the system, but the one word argument is assigned by the application
- The message identifier does not provide any direct information about where the message is stored or what kind of message has arrived. On the other hand, the one word argument can be used to store the pointer to an application structure containing all the desired information.

A similar problem exists with `mpc_recv` and `mpc_send`. When one of those operations is complete and an `mpc_wait` is called to wait for the next completed send or receive, only the message identifier of the completed operation is given to the application.

The solution to this problem is to create a linked list hash table with the message id as the hash, and a message information structure as an entry. This table contains entries for data sends, data receives, barrier messages, and reduction messages. Luckily the solution is simple, but unfortunately, machine cycles need to be used for a table lookup that would not be needed if an extra argument could be provided to the application when the asynchronous message operation completed.

Polling Function Semantics

MPL does not provide any polling function that is both nonblocking and that accepts wildcards. Such a polling function is particularly useful when a node is waiting for the next event to occur, such as when the node is waiting for a response to its request. The functions that are closest to these semantics in functionality are `mpc_status` and `mpc_wait`. The problem with `mpc_status` is that a nonwildcard message identifier must be provided, so only a specific message can be checked. `mpc_wait` does accept a wildcard as the message identifier, however, `mpc_wait` blocks and does not return until a message is complete.

Another shortcoming in the semantics is that if a message is returned as complete by either `mpc_status` or `mpc_wait`, then the handler corresponding to that message is not called. For example, if a node is waiting for a response, then it will poll with `mpc_wait` until the response arrives. In the meantime, if another message arrives, `mpc_wait` will only return the message identifier of the arrived

message and from that identifier, the arrived message's handler must be determined. So a hash table lookup on the message identifier is performed to determine which message handler to call.

Every MPL function Is Uninterruptible

An interrupt will never be called during the execution of an MPL function. So if a message arrives during an MPL function call, the message blocks until the MPL function completes. In most cases, the correctness of the application is not affected by delayed message delivery.

However, there is one case that can cause applications using CRL to hang. If a node calls any of MPL's synchronization functions, then any requests from remote nodes that arrive during the function execution are not handled. Since the remote node never gets a response to its request, the requesting node never reaches the synchronization point.

Instead of using MPL's blocking send and receive functions, the asynchronous send and receive functions are used, with some added complications as described previously. Also, MPL's synchronization functions are reimplemented to make CRL's synchronization functions interruptible.

4.2 CRL Region Version Numbers

In order to fix a bug and allow CRL-SP/2 to send multiple read-only copies of a region's data, the situations in which a region's version number is incremented has been changed. The bug occurs when two particular messages become out-of-order and cause nodes to use old region data. To support simultaneous data transfers (Section 4.3.3), it is necessary that the transfer arguments sent to each node, including the region version number, are identical.

4.2.1 MsgRelease Messages

Remote nodes use MsgRelease messages as a response to Read Invalidate messages. MsgRelease means that the remote node, which had an exclusive modified copy of the data, has kept a read-only copy of the data for itself and has sent the modified data with the MsgRelease. In the original version of CRL, the version number of the data at the remote node does not change throughout this process.

One bug, which occurs very rarely, is caused by messages becoming out of order. Suppose that the remote node sends a MsgFlush right after sending the MsgRelease and that the MsgFlush reaches the home node before the MsgRelease does. The information contained in that MsgFlush is identical to a MsgFlush that could have been sent before the Read Invalidate was received by the remote node.

The home node has no way of telling whether the remote node sent the MsgFlush before or after it received the Read Invalidate.

- If the MsgFlush was sent before the Read Invalidate was received, then the home node will be able to end the invalidation process and proceed.

- If the MsgFlush was sent after the Read Invalidate was received, then the home node will have to wait for the MsgRelease, which contains the data. Proceeding before the MsgRelease arrives may cause stale data to be used.

To fix this bug, a remote node increments its region data version number by one when it sends a MsgRelease. When a MsgFlush arrives at the home, the home can compare the version stamp on the MsgFlush with what the home node had assigned earlier to the remote node. If the version numbers match, then no MsgRelease was sent by the remote node. If the MsgFlush version number is one greater, then a MsgRelease was sent but has not arrived yet and the home node must wait for the MsgRelease to arrive.

4.2.2 Multiple Read-Only Copies

In the original CRL implementation, the version number for a particular region is incremented by one every time a request for a region is handled. This increment policy guarantees that each copy of the data received by a remote node has a unique region version number, and thus remote nodes can detect and appropriately handle out-of-order invalidate messages [18].

In CRL-SP/2, a region's version number is not incremented for Shared-Requests handled when other nodes are already sharing that particular region. It follows that if multiple nodes all have valid shared copies of a region, then all nodes have the same version number for the region.

The reason for this change is not because of a change in protocol, but because of implementation details in CRL-SP/2 discussed in Section 4.3.3. Section 5.3.1 contains a discussion on how this change further assists in the implementation of the Three-Message-Invalidate protocol.

One concern with this change is whether remote nodes are still able to appropriately handle out-of-order invalidate messages. For a discussion of out-of-order invalidate messages in the original CRL, consult [18].

It is important that each node identify which invalidates are old and can be ignored, which invalidates refer to the current version and must be processed, and which invalidates refer to a requested version that has not arrived yet and must be saved. A brief explanation of how the identification is performed follows.

If a remote node is currently not making a request on a region, and an invalidate arrives, then there are two possibilities:

1. The invalidation's version number is less than the region data's version number. In this case, the invalidate is old and can be ignored.

2. The invalidation's version number is equal to the region data's version number. In this case, the invalidate pertains to the most recent version of the data owned by the remote node. The invalidate is handled immediately, unless a region operation on the remote node conflicts with

the invalidation (such as in the case where a read-invalidation arrives during a write operation) and delays the invalidation until the operation is complete.

Usually, when a node requests a region's data, the version number assigned to the region data is greater than the version number assigned to any previous copy of the data owned by that node. It is true except in the case where a node requests a read-only copy, flushes (and thus self-invalidates) the copy, and then again requests a read-only copy. If no writable copy request arrives at the home node between these two read requests, then both the data copies may have the same version number.

At first, it seems that the node cannot determine whether the invalidate refers to the first copy and thus ignore the invalidate, or to the second copy and thus handle the invalidate. However, if the invalidate refers to the first copy, then the invalidate can only be for a write request (a read request does not cause invalidates to be sent to read-only copies of the data). Since write requests cause a region data's version number to increase, the version of the second read-only copy would have to be greater than that of the first read-only copy. Therefore the invalidate does not refer to the first read-only copy, and it must refer to the second read-only copy. Of course, the invalidation cannot refer to a future read-only request since the request has not been made yet.

The other case that needs to be taken care of is when an outstanding request has been sent by the remote node, but the response has not arrived yet. This situation will be broken up into two more cases: whether or not the remote node has a valid shared copy of the data when the invalidate arrives.

First, assume that the remote node does not have a valid copy of the data. If the invalidation version is less than the most recent data version, then the remote node just ignores the invalidation. Otherwise, it stores the invalidation and waits for the home node to send its response. On arrival of the response, the remote node compares the region version number contained in the response, to the region version number contained in the invalidation. If they are equal, then the invalidation corresponds to the most recent request and the remote node handles the invalidation after its operation completes. In this case, a future version number cannot appear since a future request has not been made.

Next, assume that the remote node does have a valid copy of the data. This case can only occur if the node sent a request to the home node indicating that it wished to modify the region data. Again, if the invalidation version is less than the most recent data version, then the remote node just ignores the invalidation. If the invalidation version is equal to the valid region data version, then the invalidation must correspond to the current version for two reasons:

1. As shown before, this invalidation message cannot corresponded to a previous copy of the data
2. The invalidation could not have been sent by the home after it sent the response since the acknowledgment to a write request causes the region data version to increase by one.

If the invalidation version is greater than the valid region data version, then the invalidation must have been sent by the home node after it had sent the modify acknowledgment. So the invalidation is saved until after the remote node receives the acknowledgment and completes the operation.

4.3 SP/2 Specific Optimizations

A number of optimizations were added specifically to the SP/2 communication section of CRL:

- Nonurgent messages, which reduce the overhead of synchronization operations and decrease the overall running time
- Two message data transfer, which half the number of messages required to transfer data and reduce the overhead of transferring data
- Simultaneous data transfers of a region, which allow simultaneous data sends to occur and reduce the response time for data requests

Some of these optimizations may also be implementable on other platforms, but because the underlying communication and synchronization for each platform is different and is specialized in CRL, these were solely implemented for the SP/2 implementation. The following sections describe the SP/2 specific optimizations in detail.

4.3.1 Nonurgent Messages

Some messages in CRL may not be crucial to the performance in the sense that they need not be handled immediately in all situations. For example, when a node reaches a barrier, it sends "reached barrier" messages. If the node receiving the message has not reached the barrier yet, then the immediate handling of that message is not necessary. The message can instead be handled when the receiving node reaches the barrier, or when the receiving node is polling for some other message.

Actually, if interrupts caused by message arrivals have a particularly expensive overhead, such as 130 microseconds on the IBM SP/2, then it is highly desirable that nonurgent messages do not interrupt the processor. The effect of this optimization is that nodes spend less time processing interrupts and thus the overall running time is reduced.

In CRL-SP/2, handlers are not setup with `mpc_rcvncall` for any synchronization operations. Instead, receive buffers for synchronization operations are setup with `mpc_recv` to prevent an interrupt from occurring when synchronization messages arrive. This change results in significant decreases in execution time. In particular, the execution time of the water application [38] decreases by 15%.

4.3.2 Two Message Data Transfer

Because of the SP/2's very expensive interrupt overhead (130 microseconds) and high message latency (55 microseconds), each message involved in data transfers dramatically increases the response time for the data request. It was of utmost importance to reduce the total number of messages required to send data from one node to another, thus reducing the overall running time.

When data needs to be transferred from one node to another, some sort of handshake protocol may be needed. On the MIT Alewife machine [1], it is possible to write directly to an address on a remote node using Alewife's builtin data transfer functions.

However on the SP/2, there is no such mechanism. The only way to transfer data is by using message passing. The normal mechanism for transferring data, and the original CRL's handshake protocol for transferring data from the home node to the remote node involved four messages (see Figure 4-1):

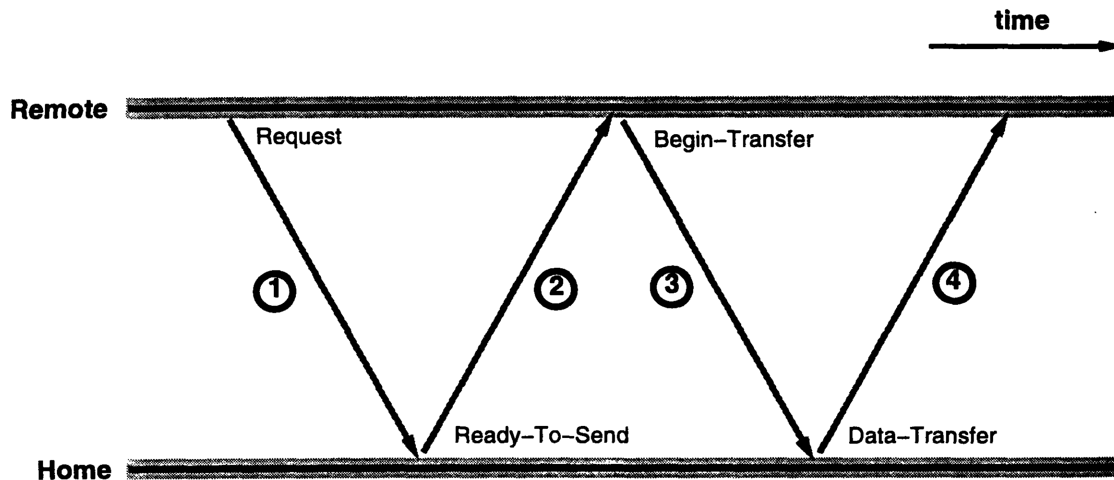

Figure 4-1: Messages sent in the original (4 message) data transfer protocol.

1. Request (such as Shared Request or Exclusive Request) from remote node to home node for data.
2. A Ready-To-Send message from the home node to the remote node containing response type and region version number information.
3. A Begin-Transfer message from the remote node to the home node containing information about which virtual path to use to send the data. For example, on the CM-5 and TCP, the virtual path is the port number, and on the SP/2, it is the message type.
4. The Data-Transfer from the home node to the remote node.

This protocol is not just limited to cases where the remote node requests data from the home node. It is also used when the home node sends an invalidate message to a remote node, and the remote

node needs to send data back to the home node. For flushes, only three messages are needed since the Ready-To-Send message is implied by the original Flush message from the remote node.

The key insight to this optimization is that before the requesting node sends a request, it can determine whether it will receive data or not with the home node's response. If the requesting node is going to receive data, then the node performs the necessary setup operations for its own side before sending the request. Then as part of the initial request message, the requesting node sends the virtual path information, and one of the messages has been removed.

In order to remove another message, it is necessary to combine the Ready-To-Send message and the Data-Transfer message. The method for implementing this combination message depends on how data transfer is performed on the architecture platform. On the SP/2, the only way to transfer messages is via a general mechanism that accepts any size buffer.

Thus the protocol in CRL-SP/2 only involves two messages (see Figure 4-2):

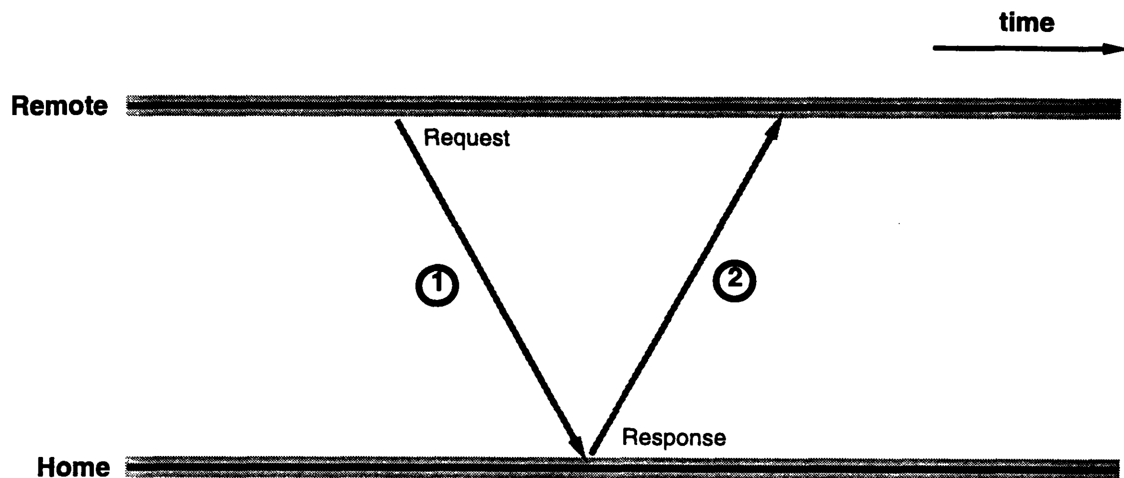

Figure 4-2: Messages sent in the optimized (2 message) data transfer protocol.

1. Request (such as Shared Request or Exclusive Request) from remote node to home node for data. Message includes information regarding the virtual path that the data should take.

2. Response (such as Shared Acknowledge Data or Exclusive Acknowledge Data) from home node to remote node with data.

This protocol works particularly well on the SP/2 since the virtual path information does not have to be sent with the request. Instead, message types are used to set the virtual data paths. For example, the message type of a message containing data for region number 1054 is 1054. On the requesting side, a handler and buffer is setup for message type 1054, and the message is handled appropriately.

On a benchmark where a region's data was ping-ponging among nodes, this optimization doubled the throughput of the number of operations.

This optimization is not fundamentally specific to the SP/2, but the need to improve the data transfer rate in the other implementations was not as necessary for reasonable performance.

4.3.3 Simultaneous Data Transfers of a Region

If multiple nodes request a read-only copy of a region's data, it is desirable that all of the data transfers occur simultaneously to all of the requesting nodes. Simultaneous transfers will reduce the latency of data requests.

As described in Section 4.3.2, when data is sent, the message arguments corresponding to the data are sent with the data. Unfortunately, this combining causes problems when multiple data sends of the same region occur simultaneously. This situation may occur, for example, if the home node receives multiple Shared-Requests, and then sends data to satisfy the requests.

The problem is that the metadata portion of the region's data, which contains the arguments for the data send, cannot be overwritten with the second send's arguments until the first send completes.

Two easy solutions to this problem are to wait until the first send completes before proceeding with the second send, or to copy the data into a temporary buffer before sending it. Of course, both of these methods are undesirable as they significantly increase the amount of time to respond to a request.

Fortunately, the only time data sends need to occur simultaneously is when multiple remote nodes request a read-only copy of the region data. The arguments contained in the response message are "Shared Acknowledge with Data", the version number, and the actual data. Because of the change in how version numbers are incremented, all of these arguments are identical for the simultaneous data transfers.

4.4 CRL-SP/2 Performance

This section displays the results of numerous CRL experiments performed on the IBM SP/2. In addition, the section contains an analysis of why the applications perform as poorly as they do on the SP/2. First, measurements are performed on the MPL round trip operations, and on MPL events and base operations. Next, the base CRL operations are examined. Finally, execution times of the applications described in Chapter 3 are measured.

The SP/2 experimental platform was a twelve-node machine using IBM AIX version 3.2. Each processing node was a Model 590 RS/6000 processor, which is a wide node running at 66MHz. Two of the twelve nodes were reserved for development, so timing measurements could only be performed on ten of the nodes. All tests were executed with interrupts on (via setting the `MP_CSS_INTERRUPT` environment variable to `yes`). The communication switch was used in dedicated (user space) mode, which is the quicker of the SP/2's two communication modes.

Round-trip time with receiver using:	μsec
mpc_wait	124
mpc_status	251
mpc_rcvncall	255

Table 4.1: Time to perform MPL round trip operations. For each measurement, the receiver of the first message uses a different method to receive the message.

Each application was executed four times and the execution time presented is an average of those four runs. Sometimes one of the four execution times was more than 10% off of the other three times. In those cases, the experiment was run once more to get a time closer to the other three. One explanation for variances in execution time is that the tests were run on an AIX system, where other processes were also running on the system. When measurements were taken, other user processes were not executing, but system processes may have been active.

4.4.1 MPL Round Trip Operations, Events and Base Operations

To understand why CRL's operations perform as they do, it is necessary to examine the primitive or base MPL operations and see how they perform. Micro-benchmarks were written and executed on the SP/2. The results appear in Tables 4.1 and 4.2.

Table 4.1 deserves a bit of explanation. Suppose there are two nodes: a requesting node which sends a request, and a responding node which responds to that request and sends a response to the requesting node. In every test, the requesting node sends a request and waits for a response. The responding node has three methods by which it may receive the request: `mpc_wait`, `mpc_status`, and `mpc_rcvncall`, all of which are described in Section 4.1.1.

In the `mpc_wait` case, the responding node calls `mpc_wait` and waits for a message to arrive. In CRL-SP/2, a node uses `mpc_wait` only when it is waiting for a response to its request. `mpc_status` is used by continuously checking to see if a particular message has arrived. This method is not used to receive messages in CRL-SP/2 because it has a much higher overhead than the `mpc_wait` method, and CRL never needs to check if a particular message has arrived. In the final case, the responding node calls `mpc_rcvncall` and then enters a busy loop. When a message arrives, an interrupt occurs and a message handler is called. CRL nodes use this method to receive requests when they are not waiting for their own responses.

The `mpc_wait` test also measures the smallest possible round-trip time using MPL. So the base application-space to application-space latency of a message is roughly 60 microseconds.

As can be seen from the times indicated in the table, the round-trip time to send a request and get a response is greatly affected by what the responding node is doing when the request arrives.

event/operation	μsec
mpc_wait call	7.4
mpc_status call	11.6
mpc_send call	20.5
mpc_recv call	39.3
mpc_rcvncall call	41.3
disable/enable interrupts pair	3.16
interrupt overhead	131

Table 4.2: Time to perform base MPL events.

Event		16-byte region		512-byte region	16384-byte region
		cycles	μsec	μsec	μsec
Start read	hit	76	1.15	1.15	1.15
End read		84	1.27	1.30	1.29
Start read	miss, no invalidations	9563	145	180	748
Start write	miss, one invalidation	17681	268	305	879
Start write	miss, six invalidations	64878	983	1019	1671

Table 4.3: Measured CRL latencies when all nodes are polling (in microseconds). This table is a partial list. The complete table is in Table A.6.

In Section 4.4.3 the effect of interrupt processing overhead on application execution time will be examined.

The time to perform the base MPL events is in Table 4.2. These measurements will be used in the next section to make sure that the base CRL operation times are close to what is expected.

4.4.2 Base CRL Operations

This section examines the time to perform the base CRL operations on CRL-SP/2 by breaking up the operations into their component MPL operations. Two sets of measurements were taken. In the first, every node is polling while messages are arriving, so there is no interrupt overhead on any message. In the second, no node is polling while messages are arriving, so there is an interrupt overhead on every message. The idea is to get a best case and worst case scenario.

Table 4.3 contains several measurements obtained while the nodes are polling, meaning that none of the CRL messages cause interrupts upon arrival. Each node is polling at a barrier. Table A.6 contains a complete set of the measurements.

The "Start read hit" entry measures the amount of time it takes to start a read operation when the node has a valid cached copy, and the "End read" measures the amount of time to end the read operation. As is to be expected, if the node already has a valid copy of the data, the amount of time to start a read is independent of the region size. Both of these operations are relatively inexpensive when compared to the read and write operations performed when there is no valid cached copy (i.e. a miss), so they will not be examined any further.

When a node begins a read operation on a noncached region, it

1. Sets up an asynchronous receive (via `mpc_recv`) for the data.

2. Sends a message to the home node.

3. Waits for a response.

When the home node receives a read request and it has valid data, it

1. Examines a list for duplicate requesters.

2. Sends the data to the requesting node.

3. Inserts information about the requester into a list.

Therefore, the time for the "Start read miss" is the `mpc_rcvncall` setup time plus the round-trip time for an `mpc_wait`. This sum results in a total of 165 μsec, which is approximately the measured amount of 145 μsec.

When a node begins a write operation on a noncached region, it performs the same operations. However, the home node now needs to send invalidate messages to any nodes that have valid cached copies, and then needs to wait for invalidate acknowledgments before sending the data to the requesting node.

1. Examine a list for duplicate requesters.

2. Send invalidates to nodes with valid cached copies.

3. Wait for all invalidate acknowledgments.

4. Send the data to the requesting node.

5. Insert information about the requester into a list.

In practice, the home node does not block while waiting for invalidates. Normally, after sending the invalidates, the home node will continue with its own computation. When an invalidate acknowledgment arrives, the region's metadata is updated. When all of the invalidate acknowledgments have arrived, the data is sent to the requesting node.

The latency time for the "Start write miss, one invalidation" should therefore be about one `mpc_wait` roundtrip (124 μsec) more than the time for the "Start read miss", which is the case.

The arithmetic gets a little more complicated when six invalidations must be performed. Since all of the invalidates are sent before any waiting is done for the invalidate acknowledgments, it seems that some of the roundtrips would overlap and thus the overhead would be less than six times the `mpc_wait` roundtrip time. However, there does not appear to be any savings for sending multiple invalidations.

Event		16-byte region		512-byte region	16384-byte region
		cycles	μsec	μsec	μsec
Start read	hit	78	1.18	1.09	1.10
End read		81	1.22	1.22	1.26
Start read	miss, no invalidations	21945	333	357	896
Start write	miss, one invalidation	55546	842	858	1401
Start write	miss, six invalidations	72369	1097	1097	1646

Table 4.4: Measured CRL latencies when no node is polling (in microseconds). This table is a partial list. The complete table is in Table A.7.

When sending data, the time difference among the columns is roughly constant, i.e. column 2 - column 1 = 36 μsec, column 3 - column 1 = 620 μsec. That roughly comes out to a bandwidth of 13.8 MB/sec when sending 512 bytes, and 26.4 MB/sec when sending 16 kbytes. The reason for greater bandwidth as more data is sent is that there is an initial startup overhead to send the data. As the amount of data gets larger, this overhead becomes less significant to the overall cost of the operation. As more data is sent in the message, a limit of 35.6 MB is approached.

The previous measurements are set so that all of the nodes are polling and the arrival of messages do not cause an interrupt. However, in applications, nodes are normally performing some computation when messages arrive, and so an interrupt occurs. Table 4.4 contains measurements obtained while the nodes are busy looping and thus not polling. The operations in this table are identical to those in Table 4.3. A complete set of nonpolling measurements appears in Table A.7.

As expected, the time for "Start/End read hit" operations does not change with what remote nodes are doing since there is no communication during these operations.

The "Start read miss, no invalidations" operation has become much more expensive because of the interrupt processing overhead. The expected time for the operation is roughly an `mpc_rcvncall` setup time and an `mpc_rcvncall` roundtrip call plus some miscellaneous overhead, which totals 296 μsec, without the search overhead. This sum is close to the 333 μsec measured.

The "Start write miss, one invalidation" operation is also much more expensive than it was in Table 4.3 (about 575 μsec more). Now this operation includes the overhead of three interrupts, which accounts for 393 μsec.

Surprisingly, the "Start write miss, six invalidations" is only 110 μsec more expensive than it was in Table 4.3. Here the invalidate round-trip messages are overlapping in time. The total time should be approximately two round-trip `mpc_rcvncalls` plus five interrupts, which totals to 1165 μsec, which is close to the measured 1097 μsec.

4.4.3 Application Performance

This section examines the execution time and speedup with the various applications described in Chapter 3. Some analysis is performed to get an idea of why the application execution times are

	Blocked LU (1000, 20) sec	Water (512 molecules) sec	Barnes-Hut (16384 bodies) sec	TSP (14 cities) sec
1 proc	10.0	2.60	38.0	54.0
2 procs	6.28	1.84	51.5	54.0
4 procs	3.84	1.28	36.1	24.5
8 procs	2.53	0.94	22.2	14.2

Table 4.5: Measured application execution time using CRL-SP/2 (in seconds).

	Blocked LU (1000, 20)	Water (512 molecules)	Barnes-Hut (16384 bodies)	TSP (14 cities)
2 procs	1.59	1.41	0.74	1.00
4 procs	2.60	2.03	1.05	2.20
8 procs	3.95	2.77	1.71	3.80

Table 4.6: Calculated application speedup using CRL-SP/2 (with 1 processor as the base execution time).

what they are. Table 4.5 contains the execution time of the applications, running on 1, 2, 4, and 8 processors. Table 4.6 contains the speedup calculations for those measurements. The speedups are calculated with respect to the one processor execution time.

The application speedups displayed in Table 4.6 are not even near the ideal of speedup equal to the number of processors. Much of the problem is with the overhead of MPL function calls and interrupts. Appendix Section A.3.1 has tables containing CRL and MPL event counts for each application.

Even though TSP and Blocked LU speedups are not near perfect, their speedup values are still respectable considering the high communication latency and interrupt costs of the SP/2. The reason why Water and Barnes-Hut do not perform as well can be seen in Table 4.7. The "computation time" is taken as the time for the 1 processor version divided by 8, giving the ideal running time for 8 processors if no messages are sent. Tables A.10 through A.13 contain the event count information.

Looking at one of the entries in Table 4.7 does not give conclusive evidence of communication overhead being the problem, however, by looking at all three expensive events and their respective rate of occurrences in all four applications results in a high degree of confidence. For example, the number of interrupts handled per computation second of Barnes-Hut is 4748. So 4748 * 130μs =

	Blocked LU (1000, 20)	Water (512 molecules)	Barnes-Hut (16384 bodies)	TSP (14 cities)
protocol messages	1019	7156	8908	1090
start operation misses	1034	2212	5715	273
interrupts handled	937	1150	4748	544

Table 4.7: The number of expensive CRL/MPL events per computation second using CRL-SP/2 on 8 processors.

617ms or approximately 60% of the time is spent in the operating system as interrupt overhead. It is clear that Barnes-Hut is not performing well because its sharing pattern is too fine grained. Water also has a large number of protocol messages, but does not cause nearly as many interrupts, which is why its speedup is better than Barnes-Hut's.

Both TSP and Blocked LU have a similar amount of overhead, but Blocked LU's is slightly better. Examining both applications and the types of messages sent during execution reveals why the difference occurs. In TSP, almost all of the communication involves starting a write operation on a shared counter. Moreover, this shared counter is never at the home node and exactly one invalidation must be sent by the home node to handle the request. In Blocked LU, each message is either a map or read request that the home node responds to without sending any invalidations. So these requests have a lower response latency, and also the data for the responses may be sent in parallel since multiple nodes are allowed to read simultaneously.

The application speedups presented in this section are not very impressive, but the next two chapters describe two enhancements that improve the speedups by reducing communication among the CRL nodes.

Chapter 5

Three-Message-Invalidation Protocol

As indicated previously, the primary reason for poor application performance on CRL-SP/2 is the high overhead in the message passing functions and interrupt handling routines. The Three-Message-Invalidation protocol attempts to reduce the number of messages sent to perform a CRL operation, and thus decrease the overall running time of the application.

The Three-Message-Invalidation protocol is not a novel protocol. It has been implemented in other DSM systems as well. Other implementations will be discussed further in the Related Work chapter (Chapter 7).

The following sections describe the Three-Message-Invalidation protocol, explain why this protocol decreases application running time, describes the protocol implementation in CRL, and compares application performance using the Three-Message-Invalidate protocol versus using the original CRL-SP/2.

5.1 Protocol Description

In a system where a region has a home node, the most straightforward implementation of a start operation request/acknowledgment pair that involves an invalidation of remote region copies consists of the following four messages, as shown in Figure 5-1:

1. Start Operation Request from the requesting node to the home node.
2. Invalidation Request from the home node to the owner node.
3. Invalidation Acknowledgment from the owner node to the home node.
4. Start Operation Acknowledgment from the home node to the requesting node.

Of course, if multiple owners exist for a region, such as in the case of multiple simultaneous readers of the region, then additional invalidation request and acknowledgment pairs (messages 2

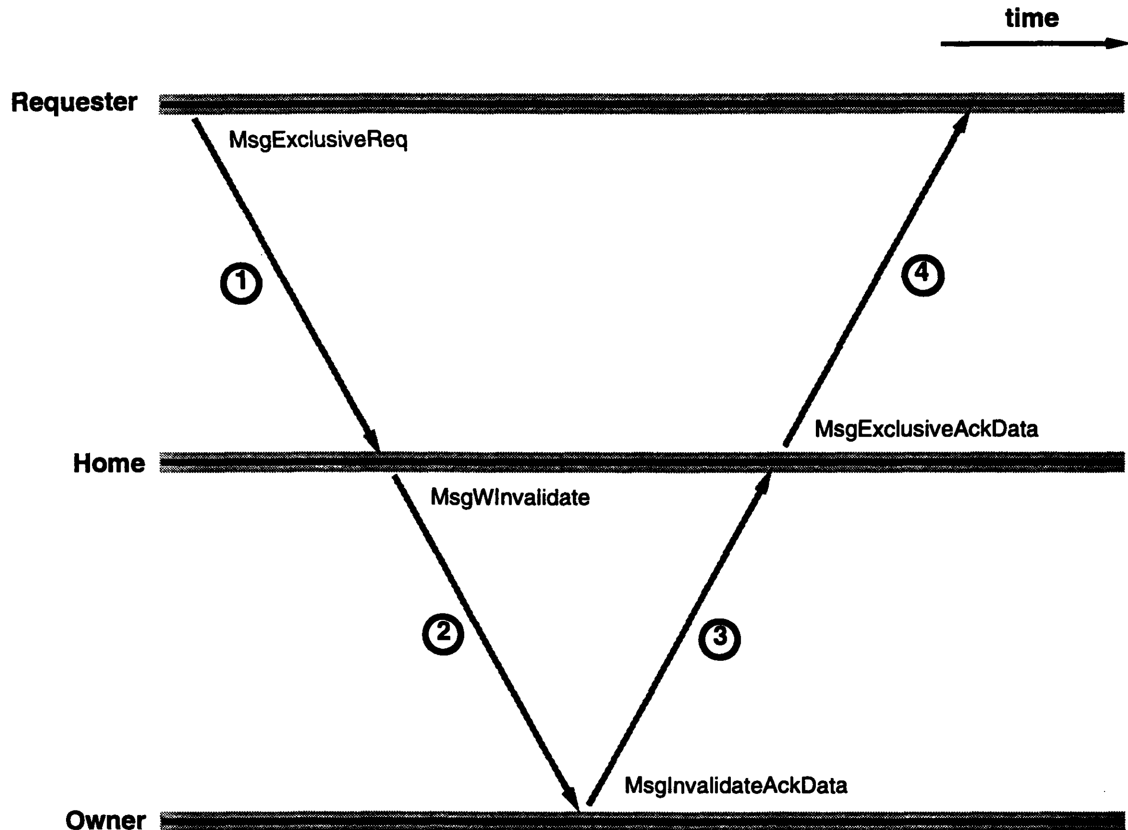

Figure 5-1: Messages sent in the original (4 message) invalidation protocol.

and 3 in Figure 5-1) are involved in the process. Handling multiple simultaneous readers is a trivial extension, and all the benefits obtained from the single remote owner case also hold for the multiple remote owner case.

In this protocol, the latency from when the start operation request is sent to when the start operation acknowledgment is received is four messages.

The messages sent in a Three-Message-Invalidation appear in Figure 5-2 and are listed below:

1. Start Operation Request from the requesting node to the home node.

2. Invalidation Request from the home node to the owner node.

3. Invalidation Acknowledgment from the owner node to the requesting node.

The primary difference is that the owner node sends the invalidation acknowledgment directly to the requesting node, rather than to the home node.

For an in-depth explanation of the implementation of the Three-Message-Invalidation coherence protocol, including state diagrams and pseudocode, please turn to Appendix Chapter B.

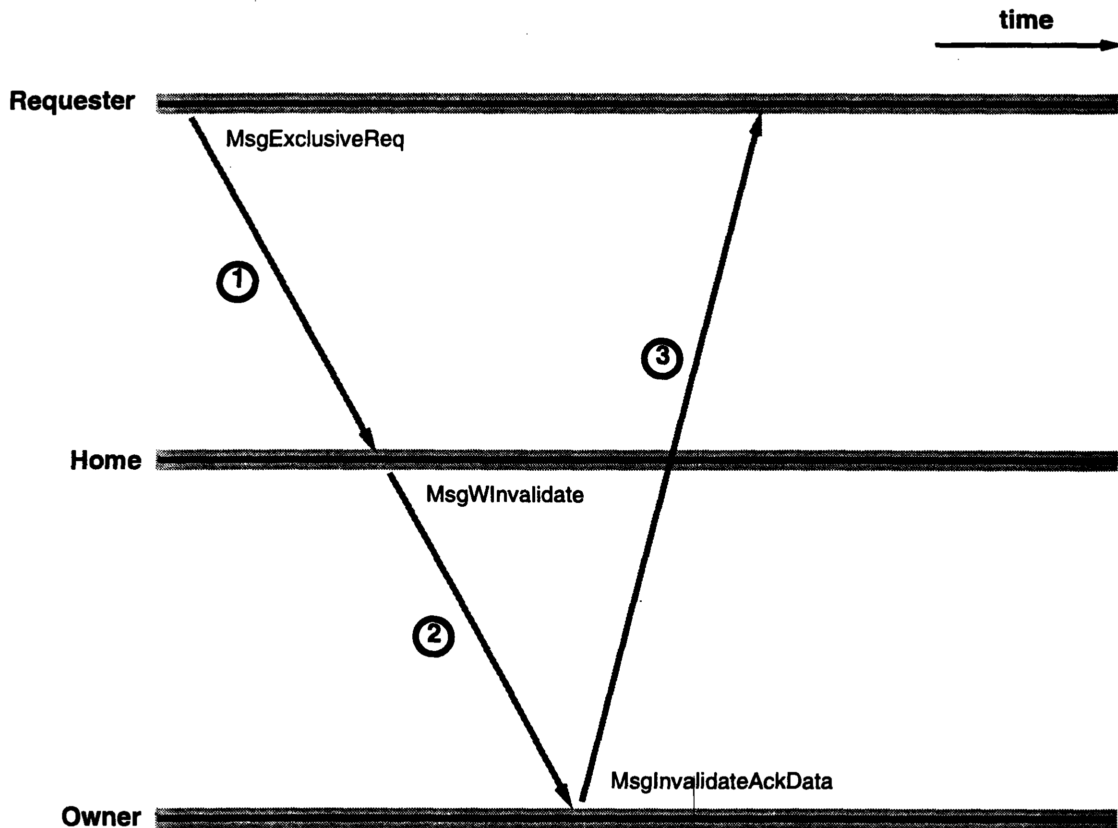

Figure 5-2: Messages sent in the optimized (3 message) invalidation protocol.

5.2 Why the Protocol Decreases Execution Time

At first, since three messages are sent instead of the previous four messages, it seems that the message latency will decrease from four messages to three messages, and thus this optimization will provide a 25% performance improvement. However, there are reasons why a greater performance improvement may occur.

One reason is that the three message invalidation wins when transferring ownership of large regions of data. Only one data transfer is in the critical path, whereas the four message version has two data transfers: the data must first be sent by the current owner to the home node and then forwarded by the home node to the requesting node.

Another reason is that if interrupts are used to signal the receiving of a message when a node is not waiting for a message, then reducing messages sent to nodes not expecting a message can result in significant performance improvement. In the common case, the number of interrupts is reduced from three to two. For example, during a request operation where the region is owned by a third node, the home node is just forwarding the invalidate and acknowledgment messages and, in general, the thread of computation on the home node is busy. When the request arrives at the home node, the processor has to interrupt the computation thread and pay some overhead. As measured in Section

4.4.1, the overhead incurred on the SP/2 is approximately 130 μsec. When the Write-Invalidate sent by the home node arrives at the remote node that owns a copy of the region, again it is most likely that the thread on that node is busy and a second interrupt is incurred. In the four message invalidate, the response to the invalidate message, which is sent back to the home node, causes another interrupt. However, this is the interrupt that does not occur in the Three-Message-Invalidation. The final message, which is a message to the requesting node. is the message that the requesting node will be expecting, and moreover, the requesting node will not be performing any computation since the node will be blocked on the operation that caused the initial request to be sent. On many machines, such as the CM-5 and SP/2, the implementation of blocking the node can be to turn off interrupts and poll for messages. Then when the message arrives, a much smaller overhead is incurred.

The improvements described above deal with the case where only one remote node needs to receive an invalidation message. However, if many nodes are sharing a readable copy, then it may be necessary for the home node to send an invalidation message to each of these sharing nodes, and then for each sharing node to respond to the home node, which would most likely be performing some computation as each message arrives. So as more nodes are sharing the region, the greater the improvement in response time for the initial request.

Even if interrupts are turned off, and only polling is used to receive messages, the Three-Message-Invalidation prevents a possible delay at the home node. For example, when the invalidate acknowledgment message arrives at the home node, it will have to wait for the home node to poll the network interface before it is handled. Again, the response time has a greater improvement if more nodes need to be sent an invalidation message, however, the improvement will not be as dramatic as that for interrupt-based message delivery.

Of course, the message latency, which is about 60μs on the SP/2 as measured in Section 4.4.1, is also saved by not having to wait for an extra message.

Theoretically, it seems that this new protocol would always be superior to the original protocol. However, there are some applications, such as Blocked LU, and Barnes-Hut that rarely take advantage of the Three-Message-Invalidation. Such applications may suffer a slight increase in execution time (1%) since these applications will still need to pay overhead for updating the Three-Message-Invalidation data structures when messages are sent.

5.3 Protocol Implementation in CRL

This section discusses several specific changes to CRL-SP/2 to create the Three-Message-Invalidate protocol. First, the method used to transfer version information will be discussed. Next, an explanation of how the Three-Message-Invalidation protocol completes is given. Then the method used to handle flushes sent during a Three-Message-Invalidation will be discussed.

5.3.1 Transferring Version Information

As seen in Section 5.1, a region's home node does not respond directly to operation requests during a Three-Message-Invalidation. Instead, it responds indirectly via other remote nodes, so the home node also sends region information (such as the version number) via this indirect path.

For portability, especially with Active-Messages [37], each protocol message is limited to a size of five words. Three words are needed to specify the remote message handler, the region, and the message type (such as Shared Request). The remaining two words are used as arguments for the message type's handler. When an invalidation message is sent, the two argument words are the invalidation version number and the node to which the invalidation acknowledgment should be sent, leaving no room for the requesting node's assigned region version number.

Because of the version number change described in Section 4.2, every node receiving an invalidate message can calculate the version number for the requesting node. When a node receives an invalidate message, it therefore can send the requesting node's assigned region version number without receiving that version number from the home node.

5.3.2 Three-Message-Invalidation Completion

In the original invalidation protocol, while an invalidation is in progress, the home node queues all requests for the region until all invalidation acknowledgments have arrived. In the Three-Message-Invalidation protocol, if a request asks for write access and invalidates are sent, then the home node does not receive any invalidation acknowledgments. So as far as the home node is concerned, the whole Write Request process is over after sending all of the invalidations. If a request asks for read access, then the home still receives a message from the current region data owner because the current region data owner will send the data to both the requesting node and the home node. Thus, a Read Request is still over after the home node receives a message from the current owner.

One concern is that before the requesting node receives a response from the current owner(s), the home node may receive another request and send an invalidation to the previous requester. This case is handled without any extra coding though, since this case is identical to what would happen with out-of-order acknowledgment/invalidate messages in the original CRL protocol [18].

5.3.3 Handling Flushes

If a flush arrives at a home node during a Three-Message-Invalidation, it is possible that the remote node sent the flush before receiving the invalidate message. In this case, an invalidate acknowledgment must be sent to the requesting node on behalf of the flushing node. According to the previous section, it is possible for multiple Three-Message-Invalidations to occur on the same region. Somehow the

home node needs to match the flush message with the corresponding Three-Message-Invalidation and send the invalidate acknowledgment to the appropriate requester.

This problem is solved with a requesting structure list. When a write request arrives, the home node first sends out the write invalidate message(s), and then stores information about the requester into a data structure, which is then put into a list. When a flush arrives, the list is searched for the structure corresponding to the flush's version number and an invalidate acknowledgment is then sent to the correct requesting node on behalf of the flushing node.

5.4 Performance Results

This section compares the execution time of the sample applications running with the Three-Message-Invalidation to CRL-SP/2 without the Three-Message-Invalidation protocol. First, the results of a benchmark are provided. Next, the time for base CRL operations using the Three-Message-Invalidation are compared with those using CRL-SP/2 without the Three-Message-Invalidation protocol. Then the application results with an analysis are given.

5.4.1 Three-Message-Invalidation Benchmark

The idea of the Three-Message-Invalidation benchmark is to determine the maximum amount of execution time improvement obtainable by an application. The benchmark that is perfect for this optimization is one where many nodes are each taking turns writing to a large region. This benchmark requires the large region to ping-pong among the nodes, making the data transfer the bottleneck.

The benchmark is run with a region of size 65536 bytes and node 0 is the home node for this region. Nodes 1 to 7 are each in a loop that writes to the region and then performs a computation not related to the region. The average time for a write operation to start and complete is measured on each of the nodes. The time using CRL-SP/2 is 32.1ms while the time using the Three-Message-Invalidate is 16.2ms, resulting in an overall execution time savings of nearly 50%.

5.4.2 Base CRL Operations

A partial set of results for the CRL operations appear in Table 5.1 and Table 5.2. The complete results are in Tables A.8 and A.9, respectively.

When nodes are polling, the latency from CRL-SP/2 without Three-Message-Invalidation decreases by as much as 40%. When nodes are busy doing some computation and interrupts must be used for message delivery, the latency decreases by as much as 30%. The primary reasons for execution time savings when the home node must send invalidations is that in Three-Message-Invalidations, the data is sent at most once per request, fewer messages are sent, and fewer interrupts are caused when nodes are busy.

Event		16-byte region		512-byte region	16384-byte region
		cycles	μsec	μsec	μsec
Start read	hit	72	1.09	1.07	1.04
End read		87	1.32	1.24	1.36
Start read	miss, no invalidations	9913	150	182	745
Start write	miss, one invalidation	13860	210	239	833
Start write	miss, six invalidations	40201	609	635	985

Table 5.1: Measured CRL latencies when all nodes are polling (in microseconds). This table is a partial list. The complete table is in Table A.8.

Event		16-byte region		512-byte region	16384-byte region
		cycles	μsec	μsec	μsec
Start read	hit	71	1.08	1.07	1.08
End read		84	1.27	1.23	1.19
Start read	miss, no invalidations	22136	335	363	847
Start write	miss, one invalidation	39402	597	626	1166
Start write	miss, six invalidations	66211	1003	1027	1393

Table 5.2: Measured CRL latencies when no node is polling (in microseconds). This table is a partial list. The complete table is in Table A.9.

5.4.3 Application Performance

This section examines the execution time and speedup of the various applications described in Chapter 3 using the Three-Message-Invalidate protocol. In addition, an execution time comparison is done with the results obtained using CRL-SP/2 without the Three-Message-Invalidation protocol. Table 5.3 contains the execution time of the applications, running on 1, 2, 4, and 8 processors. Table 5.4 contains the speedup calculations for those measurements. The speedups are calculated with respect to the one processor execution time for CRL-SP/2 without the Three-Message-Invalidation protocol.

The Three-Message-Invalidation execution times of both Blocked LU and Barnes-Hut are almost identical to those for CRL-SP/2 without the Three-Message-Invalidation protocol. The reason can be seen in the CRL and MPL event Tables A.15 and A.14, respectively.

Blocked LU does not take any advantage of Three-Message-Invalidation. In this application, the prominent region access pattern is for the home node to write to the region, and then for other nodes to read from that region. Thus the Blocked LU times are not changed significantly.

	Blocked LU (1000, 20) sec	Water (512 molecules) sec	Barnes-Hut (16384 bodies) sec	TSP (14 cities) sec
1 proc	10.0	2.58	40.0	54.0
2 procs	6.31	1.84	50.2	54.0
4 procs	3.87	1.23	36.3	20.8
8 procs	2.55	0.88	22.1	9.53

Table 5.3: Measured application execution time using Three-Message-Invalidation (in seconds).

	Blocked LU (1000, 20)	Water (512 molecules)	Barnes-Hut (16384 bodies)	TSP (14 cities)
1 proc	1.00	1.01	0.95	1.00
2 procs	1.58	1.41	0.76	1.00
4 procs	2.58	2.11	1.05	2.60
8 procs	3.92	2.95	1.72	5.67

Table 5.4: Calculated application speedup using Three-Message-Invalidation (with 1 processor CRL-SP/2 as the base execution time).

The Barnes-Hut times did not change because it only uses Three-Message-Invalidation to a small extent. On 4 processors, only 0.9% of all start operation misses use the Three-Message-Invalidation. On 8 processors, Three-Message-Invalidation is used in only 2.8% of all start operation misses. Thus start operations involving invalidations are not the bottleneck for Barnes-Hut.

Water uses Three-Message-Invalidation to a larger extent and thus decreases its 8 processor running time by 6.4%. When using 4 processors, it uses Three-Message-Invalidation in 25% of start operation misses, and when using 8 processors, Three-Message-Invalidation is used 35% of the time (see Table A.17).

TSP has the largest running time improvement because it uses Three-Message-Invalidation almost 100% of the time (see Table A.16). Its 4 processor running time decreases by over 15%, and its 8 processor running time decreases by over 30%. TSP has a counter that is used as an index into a job queue. A node increments the counter every time it takes a job from the job queue. The ping-ponging of this counter is the bottleneck for TSP, and Three-Message-Invalidation is able to reduce the invalidation latency.

Chapter 6

Floating-Home-Node Protocol

Many times, it is possible for an application programmer to determine which static set of nodes will access a particular region. In this case, the programmer can choose a particular node as the home node. However, in other cases, the set of nodes accessing a region changes in time, or it is not possible to determine the set before execution. In these cases a Floating-Home-Node protocol can be used to change the location of a region's home node.

The following sections explain why the Floating-Home-Node protocol decreases application running time, describe the protocol implementation in CRL, and compare application performance using the Floating-Home-Node protocol versus using the Three-Message-Invalidation protocol (since the Floating-Home-Node protocol changes were made using Three-Message-Invalidation as a base).

6.1 Why the Protocol Decreases Execution Time

In CRL, the home node of a region is the node that provides read/write access for that region. For example, if a node does not have valid data for a particular region, the node sends a request to the home node for the data. Also, the home node is responsible for sending out invalidations when necessary. Thus the home node also needs to keep track of which other nodes have valid copies of the data.

If only one particular node is accessing the data, then the choice of the home node is not important, since after the first access, the data will be located on the accessing node. This case happens infrequently though, since if only one node is accessing the data, then the region does not even have to be allocated as shared.

However, if multiple nodes are reading and writing to a region, then the home node choice can have a great effect on the performance of the system, and thus is extremely important. For instance, if two nodes, 0 and 1, are reading and writing to a region, and neither is the home node, then a write request from node 0 when node 1 had a modified copy would generate the following messages:

1. Node 0 $\Rightarrow$ Home node, requesting exclusive access
2. Home node $\Rightarrow$ Node 1, requesting that an invalidate acknowledgment with data be sent to node 0
3. Node 1 $\Rightarrow$ Node 0, acknowledging Node 0's access request and allowing Node 0 to continue.

If Node 0 were the home node, then the following messages would be sent:

1. Node 0 $\Rightarrow$ Node 1, requesting that an invalidate acknowledgment with data be sent to node 0
2. Node 1 $\Rightarrow$ Node 0, acknowledging Node 0's access request and allowing Node 0 to continue.

This change in the home node reduces the number of messages by a factor of 3. Moreover, it is very likely that in the first case, the home node is performing some other computation and is not expecting the message. In a polling system, processing of the request message would be delayed until the home node polled. If the system was using interrupts, then the additional message would result in wasted CPU time for processing the interrupt (which takes 130 microseconds on our SP/2). In addition, it is clear that if Node 1 were the home node, then the same benefits would be achieved.

The original version of CRL defined a region's home node as the node where the region is first created via `rgn_create`. In some cases, the access pattern for a region may be obvious and clear from an application's code, and the home node can be chosen statically and independent of the application's input.

However, sometimes the access pattern depends on the application's input. For example, the distribution for a matrix multiplication would depend on the size of the matrices, and the number of processors involved in the computation. Still, in many of these cases it is possible to determine an expression that calculates the best home node for a region. The region can then be assigned to that node at program initialization.

But there are also many applications for which better performance can be achieved. For instance, in parallel FASTLINK [11], many iterations of a function evaluation are performed. During a single execution, sometimes all of the nodes are working toward computing an iteration, and other times the nodes are split into two groups that are each computing an iteration. When all nodes are working together, it may make sense to assign a region to a particular home node. However, when the nodes are split up into two groups, a different home node may be appropriate. In this case, the application would benefit from a Floating-Home-Node protocol.

6.2 Protocol Implementation in CRL

This section talks about several specific changes made to the Three-Message-Invalidation code to create the Floating-Home-Node protocol. First the messages involved in the protocol are shown.

Then the method for handling simultaneous Become-Home requests is explained. Finally, an upper bound on the number of extra messages sent because a region's home node changes is calculated.

6.3 Messages in Floating-Home-Node Protocol

This section shows which messages are sent in the Floating-Home-Node protocol. First, to become the home node for a region, a node calls rgn_become_home. The messages sent during a Become-Home request are very similar to the ones sent during a Write request, as displayed in Figure 6-1:

1. Requester sends Become-Home request to Original Home
2. Original Home node forwards request to Real Home
3. Real Home sends Write Invalidate(s) with New-Home-Information to nodes with a valid data copy. Also, the Real Home node forwards any queued requests to the Requester.
4. nodes with a valid copy send Invalidate Acknowledgments to the Requester
5. Requester sends Become-Home-Done to Original Home
6. Original Home sends Become-Home-Done to Real Home.

The two new messages in this protocol are Become-Home-Done and New-Home-Information. (There is also a Negative Acknowledgment message, but that will be discussed in Section 6.3.1.) The Become-Home-Done message is used to indicate that the Become-Home request is complete and data-structure cleanup can be done on the Original Home node and on the previous Home node.

New-Home-Information is used to transmit information about the new home node, including the node number and address of the region on the new home node. During the Become-Home request, this message is combined with a Write-Invalidate to update the information on all of the nodes that currently own a valid copy of the data.

Also, when a previous home node receives a request, the request is forwarded to the Original Home node, which then reforwards it to the Real Home node. When the Real Home node receives such a forwarded message, it sends a New-Home-Information message to the requesting node.

For an in-depth explanation of the implementation of the Floating-Home-Node coherence protocol, including state diagrams and pseudocode, please turn to Appendix Chapter C.

6.3.1 Simultaneous Become-Home Requests

When a Become-Home request arrives at the Original Home and begins, no other Become-Home request on the same region can begin until the previous one has completed. If a Become-Home request does arrive when another one is in progress, there are two ways to handle the second request.

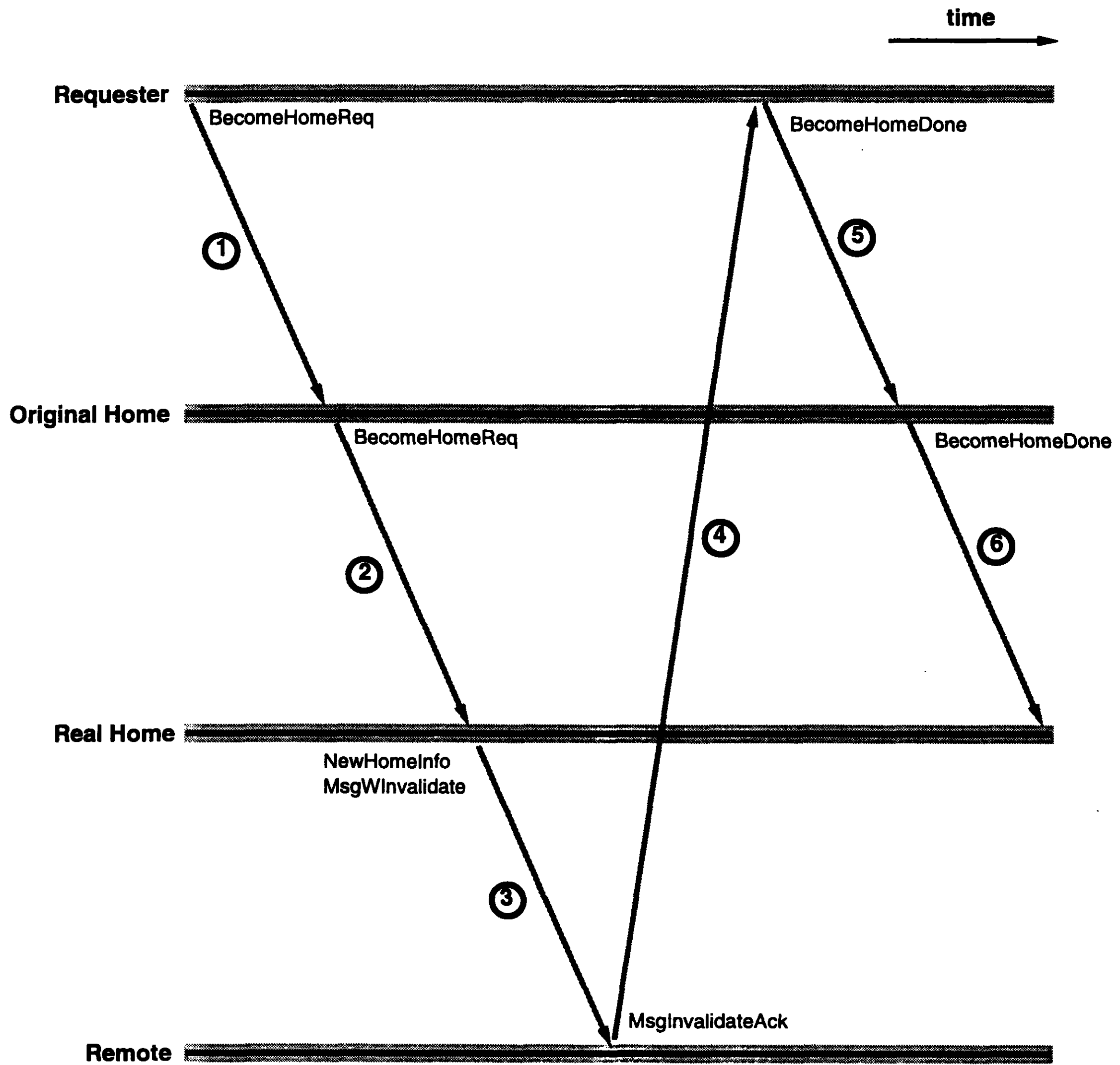

Figure 6-1: Messages sent during a become home request.

The first is to queue the request, and the second is to send a Negative-Acknowledgment message back to the second requester.

The second option was chosen for several reasons. First, the first option would further complicate the region data structures since a queue would need to be added to the Original Home node for every region. Second, the common case for Become-Home requests is that they occur rarely. An application should not be changing the home node after every operation. In fact, a Negative-Acknowledgment was never sent for the Barnes-Hut application evaluated for this protocol (see Table A.18). Third, by sending a Negative-Acknowledgment back to the requester, the requester may decided not to become the home node, as part of a heuristic.

6.3.2 Number of Forwarded Messages

One concern about moving a home node is that many requests may need to be forwarded from previous home nodes to the new home node, increasing the latency for a response. There is actually an upper bound of one forwarded request per node per Become-Home request. The reason a particular node will not have two requests forwarded is that the first forwarded request will cause New-Home-Information to be sent to the node, and the second request will go to the correct home node.

Also, if a node does not have any information regarding the region, then the node does not have any messages forwarded. When this node calls rgn_map to get region information from the original home node and maps the region, the original home node responds with information about the real home node. So when this node sends a request, the request automatically goes to the real home node and the message does not need to be forwarded.

6.4 Performance Results

This section compares the execution time of Barnes-Hut running with the Floating-Home-Node protocol to that running with just the Three-Message-Invalidation protocol. First, the results of a benchmark are provided. Next, the execution times of the base CRL operations are briefly discussed. Then the Barnes-Hut results with an analysis are given.

6.4.1 Floating-Home-Node Benchmark

The Floating-Home-Node benchmark is similar to the one used in Section 5.4.1 for Three-Message-Invalidation. A region also is being written to by a group of nodes. However, for this benchmark, the region has a size of 4 bytes, only 2 nodes are writing to the region, and a rgn_flush is performed after every write operation. At the beginning of the benchmark, the region is created on Node 0, and then Node 1 calls rgn_become_home on the region.

This new benchmark is run on both CRL-SP/2 with the Three-Message-Invalidation protocol and CRL-SP/2 with the Floating-Home-Node protocol. When running with the Three-Message-Invalidation protocol, the rgn_become_home call becomes a null call. The average time for a write operation to start and complete is measured on each of the nodes. The time using Three-Message-Invalidation is 1.41ms while the time using the Floating-Home-Node is 0.82ms, resulting in an overall execution time savings of nearly 42%.

	Barnes-Hut (16384 bodies)		
	Original CRL-SP/2 sec	Three-Message-Invalidate sec	Floating-Home-Node sec
1 proc	38.0	40.0	42.5
2 procs	51.5	50.2	20.3
4 procs	36.1	36.3	14.6
8 procs	22.2	22.1	9.68

Table 6.1: Comparison of measured Barnes-Hut execution times for each CRL-SP/2 version (in seconds).

6.4.2 Base CRL Operations

The base CRL operations are measured exactly like they were for CRL-SP/2 and for CRL-SP/2 with the Three-Message-Invalidation protocol. All of the operation times are within 2% of the corresponding Three-Message-Invalidation times in Section 5.4.2.

6.4.3 Barnes-Hut Performance

This section focuses on the performance of Barnes-Hut using the Floating-Home-Node protocol. Other applications are not discussed here since they were not modified to use the rgn_become_home call. Blocked LU and TSP are unlikely to benefit by any modifications since the initial placement of regions is the best possible. Water may benefit by modifications, but because of time constraints, the application is not examined thoroughly enough for regions that could benefit by the rgn_become_home call. The execution times of all applications other than Barnes-Hut are remeasured with the Floating-Home-Node protocol, and each time changes by at most 2% of its Three-Message-Invalidation measurement.

Barnes-Hut was modified by changing the home node for some of the bodies in the simulation. In the original Barnes-Hut, after each body's physical location in the simulation is updated, the bodies are repartitioned among the nodes. As a high-level explanation to the algorithm, the bodies are ordered linearly according to some algorithm involving their respective 3-dimensional positions in the system. Then the linear list of bodies is partitioned into n lists of consecutive bodies such that each partition involves roughly the same amount of "work", as measured in the program. After the repartitioning, the modified Barnes-Hut calls rgn_become_home on every body-region assigned to it.

Table 6.1 contains the execution times for Barnes-Hut, running on 1, 2, 4, and 8 processors for each type of CRL library running on the SP/2. Table 6.2 contains the speedup calculations for those measurements. The speedups were calculated with respect to the one processor execution time for the original CRL-SP/2.

As stated in Section 5.4.3, the Three-Message-Invalidation Barnes-Hut running times are close to

	Barnes-Hut (16384 bodies)		
	Original CRL-SP/2 sec	**Three-Message-Invalidate** sec	**Floating-Home-Node** sec
1 proc	1.00	0.95	0.89
2 procs	0.74	0.76	1.87
4 procs	1.05	1.05	2.69
8 procs	1.71	1.72	3.93

Table 6.2: Comparison of calculated Barnes-Hut speedups for each CRL-SP/2 version (with 1 processor CRL-SP/2 as the base execution time).

the original CRL-SP/2 Barnes-Hut running times. However there is over a factor of two improvement from Three-Message-Invalidation to Floating-Home-Node.

The event counts for Barnes-Hut with Three-Message-Invalidation appear in Table A.14 and those for Barnes-Hut with Floating-Home-Node appear in Table A.18. For a given problem size, the total number of rgn_map, rgn_start_read, and rgn_start_write calls, summed up over every node, is the same (actually within 1% because of some constant overhead). What the Floating-Home-Node allows the application programmer to do is significantly decrease the number of operation misses, which cause messages to be sent to other nodes.

One concern is that perhaps the regions are changing their respective home nodes on every iteration of the simulation. However, examination of the event count tables shows otherwise. For example, on the 8 processor execution, each processor has roughly 2048 bodies in its partition, yet fewer than 20 rgn_become_home calls on these regions results in a new home for a region (i.e., the other 2028 regions already have the current node as their home). Thus there is good body locality in Barnes-Hut that can and is being exploited.

The good locality is one of the primary reasons for why misses are so low. Another reason is that previously, most of the regions used by the nodes were remote and all of these regions would not fit in CRL's region cache, which is set to 1024 regions for these experiments. When the cache becomes full, flushes of regions in the cache are performed in order to make space for other regions. There are two pieces of evidence that the cache elements were being flushed. First, the number of rgn_flush calls dropped dramatically from the Three-Message-Invalidation measurement to the Floating-Home-Node measurement, indicating that the cache was not filling up nearly as much (note that the only time Barnes-Hut flushes is when the cache is full). Second, the number of Three-Message-Invalidation opportunities is much lower than the operation misses in Barnes-Hut, meaning that most of the misses involve regions that the home node has a valid copy of. Since the home node has the only valid copy, the node modifying the region is not able to keep the region in its cache.

One interesting comparison is the execution time of the 1 processor simulation (42.5 sec) to that of the 2 processor simulation (20.3 sec). Here it seems that there is a speedup greater than two, which should not be possible. In the two processor simulation, the number of operations performed

by each node is half the number that is performed in the one processor simulation, as can be seen in Table A.18. However, four times as many lookup iterations (45 million) into the region hash table are performed in the one processor version because almost twice as many regions exist in its region hash table. The lookups may be the reason why the one processor execution time is more than twice that of the two processor execution time.

From the measurements performed, it is clear that the Floating-Home-Node protocol has the ability to improve the execution time of some applications.

Chapter 7

Related Work

As described in Section 1.2, much research has been performed in the area of Distributed Shared Memory. This section focuses on those research projects that have similarities with topics in this thesis.

TreadMarks [22] is a mostly-software DSM system available on many standard UNIX operating systems including SunOS, Ultrix, and AIX. This DSM is mostly-software (instead of all-software) because although it is written at user-level and does not require any operating system modifications, it does require some virtual memory support from the operating system. A version of TreadMarks does exist for the IBM SP/2. However, performance measurements for that version could not be found and thus a direct comparison could not be made between it and CRL-SP/2.

TreadMarks uses a three message protocol for obtaining locks. A Three-Message-Invalidation and Floating-Home-Node protocol is not useful for data in TreadMarks because in TreadMarks, data does not have a home node. The home node is not required since the data consistency model in TreadMarks is release consistency, a weaker consistency model than sequential consistency that allows multiple readers and writers. Instead, data is updated by examining processor time stamps at each synchronization point and then sending out data requests when a processor accesses data modified by other processors.

TreadMarks uses release consistency in order to reduce the number of messages sent in the system. This goal was the same reason why the Three-Message-Invalidation and Floating-Home-Node protocols were examined in this thesis.

Beng-Hong Lim also ported CRL to the IBM SP/2. However, instead of using MPL for communication, he used an Active Message library written for the IBM SP/2 (SP-2 AM) [7, 6] which bypassed MPL and is layered directly on top of the SP/2's network adapter. His measurements, reported in a person communication, were the same or slightly better than the ones obtained in this thesis. There are two reasons why Lim's results were better. First, SP-2 AM did not support inter-

rupt driven message delivery, and polling had to be used. The cost of handling an interrupt (130μs) was much greater than the cost of an unsuccessful Active Message poll (1.3μs). Second, SP-2 AM reduced the latency of small messages by 40%.

The Stanford DASH machine [25, 26] is a shared-memory multiprocessor with hardware cache coherence. The coherence protocol has an optimization identical to the Three-Message-Invalidation called Request Forwarding. This optimization forwards remote requests from the home cluster to a third cluster that currently caches the requested memory block. The third cluster then responds directly to the requesting cluster. It was determined that Request Forwarding could reduced the average cache miss latency by 12% for real applications on their hardware platform. This thesis showed that Three-Message-Invalidation could also significantly reduce cache miss latencies and overall application execution time in an all-software DSM system.

The Stanford FLASH multiprocessor [24] was created after the Stanford DASH machine. [24] states that the main difference between DASH's and FLASH's protocols is that in DASH each cluster collects its own invalidation acknowledgments, whereas in FLASH invalidation acknowledgments are collected at the home node, that is, the node where the directory data is stored for that block. It seems that for some reason, the Request Forwarding has been removed, but unfortunately, no reason is given for the removal in the paper.

With respect to application specific optimizations, FLASH has a programmable protocol processor for each node in the system. Thus, the Three-Message-Invalidate and the Floating-Home-Node protocol should be relatively easily implementable on FLASH.

Falsafi et al. [9] describes some application specific protocol optimizations that can be implemented on Blizzard to improve application performance. Barnes-Hut is the only application that is examined in both this thesis and [9]. None of the optimizations mentioned in their research are similar to the Three-Message-Invalidation or the Floating-Home-Node protocol. These two optimizations complement the work done by Falsafi et al.

Much research has been done in Cache-Only Memory Architectures (COMA) [23, 12, 34, 35]. In COMAs, each processor contains a portion of the address space, however, the partition is not fixed. Instead, the address space of a processor is like another level of cache. So COMA is similar to the Floating-Home-Node idea that data is not bound to any particular processing node. Data in COMA machines do not have home nodes. Thus a particular processing node is never responsible for keeping a particular data item coherent. In reality, some COMA machines do have home nodes for data items, but these nodes are only responsible for keeping a valid copy of the data if no other node has a valid copy in its cache.

The Kendall Square Research KSR1 system [23, 34] is an example of a COMA machine. The KSR1 processors are grouped into a hierarchy. 32 processing nodes are grouped together into a ring called Ring:0. 32 Ring:0s grouped together make a Ring:1, and 32 Ring:1s grouped together make a

Ring:2. Instead of sending a request directly to a particular node, requesters broadcast the request on their local Ring:0. If the request cannot be satisfied by any of these nodes, then the request is broadcasted on Ring:1. If the request cannot be satisfied by any of these nodes, then the request is broadcast on Ring:2, where the request must be satisfied since some node in the system must have valid data to satisfy the request. Because nodes with valid copies of data respond directly to requests, a Three-Message-Invalidation protocol is not useful in COMA architectures.

Chapter 8

Conclusion

This thesis discussed the design, implementation, and evaluation of a C Region Library port to the IBM SP/2. In addition, it discussed the design, implementation, and evaluation of two optimizations to the original CRL protocol, a Three-Message-Invalidation protocol, and a Floating-Home-Node protocol.

The IBM SP/2 was the platform used for this thesis. The interrupt overhead and message latency on the SP/2 were very large at 130μs, and 60μs, respectively. The cost of these two message passing operation components on the SP/2 resulted in less than ideal speedup for the applications tested. Speedups on 8 processors ranged from 1.71 for Barnes-Hut to 3.95 for Blocked LU.

The Three-Message-Invalidate protocol helps those applications that have a large number of invalidate messages sent out to remote nodes in order to satisfy another remote node's request. This is done by reducing the number of messages required for a one-node invalidation from four messages to three messages. Improvements ranged from no improvement to a 30% reduction in application running time on 8 processors. Specifically, Blocked LU, which does not use invalidations, and Barnes-Hut, which uses invalidations for fewer than 3% of all requests, did not improve at all. Water-Nsquared, which uses invalidations for 35% of all requests, had a 6% reduction in running time. Finally, TSP, which uses invalidations almost 100% of the time, had a 30% reduction in running time.

The Floating-Home-Node protocol provides the ability to move a region's home node to another node, i.e. it provides the ability to change the node responsible for maintaining the coherency of the region's data. The Floating-Home-Node protocol was shown to be very effective when used on appropriate regions. It is best used by regions that are accessed by a set of nodes, where the set changes during the execution of the application. Barnes-Hut took advantage of this protocol to reduce its 8 processor running time by 56%. Specifically, the protocol was used to change the home node

for a simulation body whenever the responsibility for calculating that body's attributes was assigned to another node.

Appendix A

Raw Experimental Data

This chapter contains the raw data used in the analysis in Chapters 4, 5, and 6. The sections compare MPL and MPI on the IBM SP/2, display CRL operation latencies, and display CRL and MPL event counts for each application.

A.1 IBM Message Passing Library (MPL) versus Message Passing Interface (MPI)

This table was obtained from [8] and compares the performance of MPL and MPI.

Notes:

1. Exchange bandwidth is defined as a simultaneous send and receive data exchange between nodes.
2. Megabytes are defined as 10**6 bytes.

Node Type	Latency	Pt to Pt BW	Exchange BW
66 MHz "Thin Node"	40.0 Microseconds	35.4 MB/Sec.	41.2 MB/Sec.
66 MHz "Thin Node 2"	39.0 Microseconds	35.7 MB/Sec.	48.3 MB/Sec.
66 MHz "Wide Node"	39.2 Microseconds	35.6 MB/Sec.	48.3 MB/Sec.

Table A.1: Switch performance using MPL (user space, application space to application space)

Node Type	Latency	Pt to Pt BW	Exchange BW
66 MHz "Thin Node"	312.1 Microseconds	9.9 MB/Sec.	12.9 MB/Sec.
66 MHz "Thin Node 2"	270.4 Microseconds	12.0 MB/Sec.	15.8 MB/Sec.
66 MHz "Wide Node"	268.8 Microseconds	12.1 MB/Sec.	15.6 MB/Sec.

Table A.2: Switch performance using MPL (udp/IP, application space to application space)

Node Type	Latency	Pt to Pt BW	Exchange BW
66 MHz "Wide Node"	less than 45 Microseconds	greater than 35 MB/Sec.	greater than 47 MB/Sec

Table A.3: Preliminary switch performance using MPI (user space, application space to application space)

Node Type	Latency	Pt to Pt BW	Exchange BW
66 MHz "Wide Node"	less than 300 Microseconds	greater than 12 MB/Sec.	greater than 16 MB/Sec.

Table A.4: Preliminary switch performance using MPI (udp/IP, application space to application space)

A.2 CRL Operation Latencies

Kirk Johnson wrote a simple microbenchmark to measure the cost of various CRL events. A description of the microbenchmark from [18] follows:

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.

Table A.5 (taken directly from [18]) describes the 26 different types of events measured by the microbenchmark. This microbenchmark is executed twice, once with nonrequesting nodes polling for messages, and once with nonrequesting nodes not polling for messages. The difference is that in the second case, the arrival of messages causes an interrupt with an overhead of approximately 130μs. Table A.6 contains the CRL-SP/2 latencies when the nonrequesting nodes are polling for messages. Table A.7 contains the CRL-SP/2 latencies when the nonrequesting nodes are not polling for messages. Table A.8 contains the Three-Message-Invalidation latencies when the nonrequesting nodes are polling for messages. Table A.9 contains the Three-Message-Invalidation latencies when the nonrequesting nodes are not polling for messages.

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 A.5: Events measured by latency microbenchmark.

Event		Region Size (bytes) 16 cycles	16 μsec	512 μsec	16384 μsec
Map	miss	10263	156	157	164
Map	hit [a]	68	1.03	0.91	1.06
Map	hit [b]	45	0.68	0.63	0.69
Unmap	[c]	12	0.19	0.29	0.25
Unmap	[d]	40	0.61	0.65	0.64
Start read	miss, 0 copies	9563	145	180	748
Start read	miss, 1 copies	9583	145	179	744
Start read	miss, 2 copies	9629	146	182	749
Start read	miss, 3 copies	9775	148	182	777
Start read	miss, 4 copies	9629	146	180	750
Start read	miss, 5 copies	9643	146	182	806
Start read	miss, 6 copies	9616	146	181	842
Start read	hit [e]	76	1.15	1.15	1.15
Start read	hit [f]	80	1.21	1.17	1.21
End read	[g]	86	1.30	1.25	1.28
End read	[h]	84	1.26	1.30	1.29
Start write	miss, 0 inv	9478	144	179	749
Start write	miss, 1 inv	17681	268	305	879
Start write	miss, 2 inv	28783	436	458	1023
Start write	miss, 3 inv	37970	575	612	1196
Start write	miss, 4 inv	49500	750	790	1335
Start write	miss, 5 inv	58364	884	913	1525
Start write	miss, 6 inv	64878	983	1019	1671
Start write	modify	9471	144	145	142
Start write	hit	81	1.22	1.22	1.07
End write		83	1.25	1.24	1.11

Table A.6: CRL-SP/2 latencies with nodes polling

Event		Region Size (bytes) 16 cycles	16 μsec	512 μsec	16384 μsec
Map	miss	20618	312	306	313
Map	hit [a]	62	0.93	0.95	0.95
Map	hit [b]	43	0.66	0.70	0.68
Unmap	[c]	17	0.25	0.23	0.28
Unmap	[d]	41	0.63	0.63	0.65
Start read	miss, 0 copies	21945	333	357	896
Start read	miss, 1 copies	22242	337	361	900
Start read	miss, 2 copies	22328	338	359	904
Start read	miss, 3 copies	22202	336	358	905
Start read	miss, 4 copies	22169	336	360	902
Start read	miss, 5 copies	22163	336	362	907
Start read	miss, 6 copies	22189	336	360	903
Start read	hit [e]	78	1.18	1.09	1.10
Start read	hit [f]	71	1.07	1.07	1.11
End read	[g]	77	1.17	1.16	1.20
End read	[h]	81	1.22	1.22	1.26
Start write	miss, 0 inv	22024	334	359	900
Start write	miss, 1 inv	55546	842	858	1401
Start write	miss, 2 inv	58483	886	902	1441
Start write	miss, 3 inv	61552	933	942	1487
Start write	miss, 4 inv	64449	977	990	1531
Start write	miss, 5 inv	67848	1028	1042	1587
Start write	miss, 6 inv	72369	1097	1097	1646
Start write	modify	21648	328	324	327
Start write	hit	71	1.08	1.08	1.09
End write		79	1.20	1.16	1.24

Table A.7: CRL-SP/2 latencies with nodes not polling

Event		Region Size (bytes) 16 cycles	16 μsec	512 μsec	16384 μsec
Map	miss	10256	155	155	159
Map	hit [a]	61	0.92	0.87	0.93
Map	hit [b]	44	0.66	0.65	0.63
Unmap	[c]	17	0.27	0.31	0.24
Unmap	[d]	43	0.66	0.65	0.69
Start read	miss, 0 copies	9913	150	182	745
Start read	miss, 1 copies	9689	147	183	741
Start read	miss, 2 copies	9808	149	182	747
Start read	miss, 3 copies	9847	149	182	775
Start read	miss, 4 copies	9728	147	184	750
Start read	miss, 5 copies	9788	148	184	798
Start read	miss, 6 copies	9847	149	182	836
Start read	hit [e]	72	1.09	1.07	1.04
Start read	hit [f]	82	1.24	1.20	1.08
End read	[g]	82	1.24	1.26	1.17
End read	[h]	87	1.32	1.24	1.36
Start write	miss, 0 inv	9689	147	182	748
Start write	miss, 1 inv	13860	210	239	833
Start write	miss, 2 inv	20665	313	296	834
Start write	miss, 3 inv	27337	414	476	891
Start write	miss, 4 inv	32248	489	542	922
Start write	miss, 5 inv	36135	548	592	952
Start write	miss, 6 inv	40201	609	635	985
Start write	modify	9821	149	150	149
Start write	hit	71	1.07	1.21	1.04
End write		80	1.21	1.23	1.15

Table A.8: Three-Message-Invalidation latencies with nodes polling

Event		Region Size (bytes) 16 cycles	16 μsec	512 μsec	16384 μsec
Map	miss	20368	309	307	314
Map	hit [a]	63	0.95	0.93	0.96
Map	hit [b]	45	0.68	0.69	0.70
Unmap	[c]	16	0.24	0.26	0.28
Unmap	[d]	41	0.62	0.65	0.64
Start read	miss, 0 copies	22136	335	363	847
Start read	miss, 1 copies	22645	343	366	901
Start read	miss, 2 copies	22493	341	366	903
Start read	miss, 3 copies	22526	341	370	903
Start read	miss, 4 copies	22255	337	366	902
Start read	miss, 5 copies	22506	341	366	903
Start read	miss, 6 copies	22447	340	366	904
Start read	hit [e]	71	1.08	1.07	1.08
Start read	hit [f]	73	1.10	1.09	1.09
End read	[g]	78	1.18	1.16	1.18
End read	[h]	84	1.27	1.23	1.19
Start write	miss, 0 inv	22143	336	366	901
Start write	miss, 1 inv	39402	597	626	1166
Start write	miss, 2 inv	44068	668	702	1209
Start write	miss, 3 inv	52034	788	795	1264
Start write	miss, 4 inv	58047	880	876	1317
Start write	miss, 5 inv	62746	951	945	1353
Start write	miss, 6 inv	66211	1003	1027	1393
Start write	modify	21965	333	325	329
Start write	hit	73	1.11	1.13	1.09
End write		76	1.15	1.16	1.18

Table A.9: Three-Message-Invalidation latencies with nodes not polling

A.3 CRL and MPL Event Counts

This section contains the raw data obtained with an instrumented version of the CRL library created by Kirk Johnson.

The following, with parts taken from [18], explains the contents of each table:

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. The fourth section of the table indicates how many times `rgn_flush` was called, and how many times `rgn_become_home` was called, if applicable.

The fifth section of the table shows counts for the MsgRgnInfoReq and MsgRgnInfoAck messages required when calls to `rgn_map` cannot be satisfied locally. The sixth 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. The seventh section of the table provides counts for protocol messages. The eighth section of the table shows the total number of region table lookups performed, and how many iterations of the search loop were performed. The ninth section of the table provides counts for the MPL functions. Finally, the tenth section of the table provides counts for interrupt events, and Three-Message-Invalidate opportunities, if applicable.

The following sections contain tables for application execution on CRL-SP/2, Three-Message-Invalidation, and Floating-Home-Node.

A.3.1 CRL-SP/2 Application Events

This section contains CRL and MPL event counts for each application running on CRL-SP/2. Table A.10 contains the counts for Barnes-Hut. Table A.11 contains the counts for Blocked LU. Table A.12 contains the counts for TSP. Table A.13 contains the counts for Water.

Event	1 proc	2 procs	4 procs	8 procs
Map	4744745	2372382	1186207	593118
(remote)	0	694469	663045	412968
(miss)	0	23141	20355	12536
(unmap)	4744745	2372382	1186207	593118
Start read	4650895	2325457	1162745	581388
(remote)	0	679903	652153	406534
(miss)	0	14793	14230	9072
(block)	0	0	7	12
(end)	4650895	2325457	1162745	581388
Start write	128276	64165	66238	72846
(remote)	0	18711	48290	65138
(miss)	0	12476	9402	5539
(block)	0	0	1	0
(end)	128276	64165	66238	72846
Flush	0	23141	20355	12536
MsgRgnInfoReq	0	23141	20355	12536
(ack)	0	23141	20355	12536
total protocol msgs	0	77614	67814	42316
(queued)	0	4194	3560	2053
MsgSharedReq	0	14754	14151	9017
MsgSharedAckData	0	14754	14151	9017
MsgExclusiveReq	0	8426	6331	3719
MsgExclusiveAckData	0	8426	6331	3719
MsgModifyReq	0	3985	2992	1773
MsgModifyAck	0	3984	2990	1771
MsgModifyAckData	0	1	2	3
MsgRInvalidate	0	39	179	268
MsgWInvalidate	0	65	203	292
MsgInvalidateAck	0	66	195	274
MsgInvalidateAckData	0	40	190	289
MsgRelease	0	0	0	0
MsgFlush	0	10705	10970	6977
MsgFlushData	0	12371	9133	5201
Region table lookup	4744745	2395604	1206929	606205
(iters)	22628546	5218946	1738467	633974
`mpc_send`	0	136350	117780	72733
`mpc_recv`	0	84	126	147
`mpc_rcvncall`	0	136268	117657	72589
`mpc_status`	0	145116	123909	76108
`mpc_wait`	0	127596	111669	69378
`mpc_lockrnc`	186	247256	214983	131864
interrupt handler	0	48425	39022	22554

Table A.10: CRL and MPL event counts for Barnes-Hut on CRL-SP/2; all values are per-processor averages computed over four consecutive runs.

Event	1 proc	2 procs	4 procs	8 procs
Map	84576	42314	21489	10770
(remote)	0	10288	5476	4044
(miss)	0	638	650	656
(unmap)	84576	42314	21489	10770
Start read	41701	20876	10757	5392
(remote)	0	10288	5463	4025
(miss)	0	638	638	637
(block)	0	0	0	0
(end)	41701	20876	10757	5392
Start write	42925	21463	10731	5366
(remote)	0	0	0	0
(miss)	0	0	0	0
(block)	0	0	0	0
(end)	42925	21463	10731	5366
Flush	0	0	0	0
MsgRgnInfoReq	0	638	650	656
(ack)	0	638	650	656
total protocol msgs	0	1275	1275	1274
(queued)	0	12	10	5
MsgSharedReq	0	638	638	637
MsgSharedAckData	0	638	638	637
MsgExclusiveReq	0	0	0	0
MsgExclusiveAckData	0	0	0	0
MsgModifyReq	0	0	0	0
MsgModifyAck	0	0	0	0
MsgModifyAckData	0	0	0	0
MsgRInvalidate	0	0	0	0
MsgWInvalidate	0	0	0	0
MsgInvalidateAck	0	0	0	0
MsgInvalidateAckData	0	0	0	0
MsgRelease	0	0	0	0
MsgFlush	0	0	0	0
MsgFlushData	0	0	0	0
Region table lookup	84576	42950	22139	11425
(iters)	84576	49877	26391	10968
`mpc_send`	0	2844	3016	3100
`mpc_recv`	0	297	446	520
`mpc_rcvncall`	0	2550	2575	2586
`mpc_status`	0	2702	2802	2851
`mpc_wait`	0	3009	3264	3389
`mpc_lockrnc`	630	4479	4525	4534
interrupt handler	0	1272	1196	1172

Table A.11: CRL and MPL event counts for Blocked LU on CRL-SP/2; all values are per-processor averages computed over four consecutive runs.

Event	1 proc	2 procs	4 procs	8 procs
Map	0	0	0	0
(remote)	0	0	0	0
(miss)	0	0	0	0
(unmap)	0	0	0	0
Start read	34323	10865615	1710160	376379
(remote)	0	23545	17504	14923
(miss)	0	3	7	9
(block)	0	0	0	0
(end)	34323	10865615	1710160	376379
Start write	17180	8590	4297	2150
(remote)	0	8589	4296	2150
(miss)	0	3	3387	1833
(block)	0	0	0	0
(end)	17180	8590	4297	2150
Flush	0	0	0	0
MsgRgnInfoReq	0	0	0	0
(ack)	0	0	0	0
total protocol msgs	0	11	13564	7355
(queued)	0	2	3401	1843
MsgSharedReq	0	3	7	8
MsgSharedAckData	0	3	7	8
MsgExclusiveReq	0	2	3386	1832
MsgExclusiveAckData	0	2	3386	1832
MsgModifyReq	0	1	1	1
MsgModifyAck	0	1	1	0
MsgModifyAckData	0	0	0	0
MsgRInvalidate	0	1	2	1
MsgWInvalidate	0	1	3387	1835
MsgInvalidateAck	0	1	3	5
MsgInvalidateAckData	0	1	3386	1832
MsgRelease	0	0	0	0
MsgFlush	0	0	0	0
MsgFlushData	0	0	0	0
Region table lookup	0	1	3389	1837
(iters)	0	1	3389	1837
mpc_send	0	153	13777	7604
mpc_recv	0	145	218	254
mpc_rcvncall	0	11	13564	7355
mpc_status	0	87	17062	9330
mpc_wait	0	243	10526	5928
mpc_lockrnc	326	343	12124	6428
interrupt handler	0	5	6777	3673

Table A.12: CRL and MPL event counts for TSP on CRL-SP/2; all values are per-processor averages computed over four consecutive runs.

Event	1 proc	2 procs	4 procs	8 procs
Map	0	0	0	0
(remote)	0	0	0	0
(miss)	0	0	0	0
(unmap)	0	0	0	0
Start read	263682	131842	65922	32962
(remote)	0	32769	24578	14338
(miss)	0	383	387	341
(block)	0	0	0	0
(end)	263682	131842	65922	32962
Start write	5640	3078	1669	965
(remote)	0	258	259	260
(miss)	0	637	482	379
(block)	0	0	0	1
(end)	5640	3078	1669	965
Flush	0	0	0	0
MsgRgnInfoReq	0	0	0	0
(ack)	0	0	0	0
total protocol msgs	0	2040	2367	2326
(queued)	0	75	45	26
MsgSharedReq	0	381	385	340
MsgSharedAckData	0	381	385	340
MsgExclusiveReq	0	2	155	223
MsgExclusiveAckData	0	2	155	223
MsgModifyReq	0	256	103	35
MsgModifyAck	0	256	103	35
MsgModifyAckData	0	0	0	1
MsgRInvalidate	0	3	66	36
MsgWInvalidate	0	380	475	529
MsgInvalidateAck	0	125	283	307
MsgInvalidateAckData	0	257	258	259
MsgRelease	0	0	0	0
MsgFlush	0	0	0	0
MsgFlushData	0	0	0	0
Region table lookup	0	382	540	564
(iters)	0	385	540	564
`mpc_send`	0	2117	2483	2462
`mpc_recv`	0	79	119	138
`mpc_rcvncall`	0	2040	2367	2326
`mpc_status`	0	2081	2456	2443
`mpc_wait`	0	2165	2527	2501
`mpc_lockrnc`	176	2362	1982	1666
interrupt handler	0	510	448	375

Table A.13: CRL and MPL event counts for Water on CRL-SP/2; all values are per-processor averages computed over four consecutive runs.

A.3.2 Three-Message-Invalidation Application Events

This section contains CRL and MPL event counts for each application running on Three-Message-Invalidation. Table A.14 contains the counts for Barnes-Hut. Table A.15 contains the counts for Blocked LU. Table A.16 contains the counts for TSP. Table A.17 contains the counts for Water.

Event	1 proc	2 procs	4 procs	8 procs
Map	4744745	2372382	1186207	593118
(remote)	0	694499	663189	412959
(miss)	0	23134	20341	12520
(unmap)	4744745	2372382	1186207	593118
Start read	4650895	2325457	1162745	581388
(remote)	0	679925	652297	406525
(miss)	0	14789	14213	9052
(block)	0	0	4	6
(end)	4650895	2325457	1162745	581388
Start write	128276	64138	65323	72754
(remote)	0	18692	47374	65042
(miss)	0	12449	9401	5538
(block)	0	0	0	0
(end)	128276	64138	65323	72754
Flush	0	23134	20341	12520
MsgRgnInfoReq	0	23134	20341	12520
(ack)	0	23134	20341	12520
total protocol msgs	0	77568	67642	42045
(queued)	0	3913	3429	1987
MsgSharedReq	0	14746	14133	8995
MsgSharedAckData	0	14746	14037	8786
MsgExclusiveReq	0	8426	6331	3719
MsgExclusiveAckData	0	8426	6296	3655
MsgModifyReq	0	3986	2990	1770
MsgModifyAck	0	3985	2914	1640
MsgModifyAckData	0	1	1	0
MsgRInvalidate	0	43	176	267
MsgWInvalidate	0	37	201	288
MsgInvalidateAck	0	37	164	216
MsgInvalidateAckData	0	43	313	553
MsgRelease	0	0	0	0
MsgFlush	0	10725	10955	6961
MsgFlushData	0	12369	9134	5199
Region table lookup	4744745	2395577	1206909	606184
(iters)	22628546	5218660	1737465	634110
mpc_send	0	136288	117581	72429
mpc_recv	0	84	126	147
mpc_rcvncall	0	136206	117457	72284
mpc_status	0	145359	123797	75852
mpc_wait	0	127229	111382	69025
mpc_lockrnc	186	246629	214637	131605
interrupt handler	0	48482	38929	22553
three message inv	0	0	207	402

Table A.14: CRL and MPL event counts for Barnes-Hut on Three-Message-Invalidation; all values are per-processor averages computed over four consecutive runs.

Event	1 proc	2 procs	4 procs	8 procs
Map	84576	42314	21489	10770
(remote)	0	10288	5476	4044
(miss)	0	638	650	656
(unmap)	84576	42314	21489	10770
Start read	41701	20876	10757	5392
(remote)	0	10288	5463	4025
(miss)	0	638	638	637
(block)	0	0	0	0
(end)	41701	20876	10757	5392
Start write	42925	21463	10731	5366
(remote)	0	0	0	0
(miss)	0	0	0	0
(block)	0	0	0	0
(end)	42925	21463	10731	5366
Flush	0	0	0	0
MsgRgnInfoReq	0	638	650	656
(ack)	0	638	650	656
total protocol msgs	0	1275	1275	1274
(queued)	0	12	18	7
MsgSharedReq	0	638	638	637
MsgSharedAckData	0	638	638	637
MsgExclusiveReq	0	0	0	0
MsgExclusiveAckData	0	0	0	0
MsgModifyReq	0	0	0	0
MsgModifyAck	0	0	0	0
MsgModifyAckData	0	0	0	0
MsgRInvalidate	0	0	0	0
MsgWInvalidate	0	0	0	0
MsgInvalidateAck	0	0	0	0
MsgInvalidateAckData	0	0	0	0
MsgRelease	0	0	0	0
MsgFlush	0	0	0	0
MsgFlushData	0	0	0	0
Region table lookup	84576	42951	22139	11426
(iters)	84576	49877	26391	10967
mpc_send	0	2844	3016	3100
mpc_recv	0	297	446	520
mpc_rcvncall	0	2550	2575	2586
mpc_status	0	2702	2802	2851
mpc_wait	0	3009	3264	3389
mpc_lockrnc	630	4480	4537	4537
interrupt handler	0	1272	1199	1177
three message inv	0	0	0	0

Table A.15: CRL and MPL event counts for Blocked LU on Three-Message-Invalidation; all values are per-processor averages computed over four consecutive runs.

Event	1 proc	2 procs	4 procs	8 procs
Map	0	0	0	0
(remote)	0	0	0	0
(miss)	0	0	0	0
(unmap)	0	0	0	0
Start read	34323	10866296	1718964	336227
(remote)	0	23499	17396	14774
(miss)	0	3	7	8
(block)	0	0	0	0
(end)	34323	10866296	1718964	336227
Start write	17180	8590	4297	2150
(remote)	0	8589	4296	2150
(miss)	0	3	2381	1618
(block)	0	0	0	0
(end)	17180	8590	4297	2150
Flush	0	0	0	0
MsgRgnInfoReq	0	0	0	0
(ack)	0	0	0	0
total protocol msgs	0	11	7161	4880
(queued)	0	1	1209	689
MsgSharedReq	0	3	7	8
MsgSharedAckData	0	3	5	7
MsgExclusiveReq	0	2	2380	1618
MsgExclusiveAckData	0	2	1	0
MsgModifyReq	0	1	1	1
MsgModifyAck	0	1	0	0
MsgModifyAckData	0	0	0	0
MsgRInvalidate	0	1	2	1
MsgWInvalidate	0	1	2381	1621
MsgInvalidateAck	0	1	3	4
MsgInvalidateAckData	0	1	2382	1620
MsgRelease	0	0	0	0
MsgFlush	0	0	0	0
MsgFlushData	0	0	0	0
Region table lookup	0	1	2383	1622
(iters)	0	1	2383	1622
mpc_send	0	153	7374	5128
mpc_recv	0	145	218	254
mpc_rcvncall	0	11	7161	4880
mpc_status	0	87	7329	5028
mpc_wait	0	243	7507	5284
mpc_lockrnc	326	341	8222	6196
interrupt handler	0	5	4228	2325
three message inv	0	0	2381	1619

Table A.16: CRL and MPL event counts for TSP on Three-Message-Invalidation; all values are per-processor averages computed over four consecutive runs.

Event	1 proc	2 procs	4 procs	8 procs
Map	0	0	0	0
(remote)	0	0	0	0
(miss)	0	0	0	0
(unmap)	0	0	0	0
Start read	263682	131842	65922	32962
(remote)	0	32769	24578	14338
(miss)	0	383	390	371
(block)	0	0	0	0
(end)	263682	131842	65922	32962
Start write	5640	3078	1669	965
(remote)	0	258	259	260
(miss)	0	636	482	378
(block)	0	0	0	0
(end)	5640	3078	1669	965
Flush	0	0	0	0
MsgRgnInfoReq	0	0	0	0
(ack)	0	0	0	0
total protocol msgs	0	2039	2219	2219
(queued)	0	66	37	25
MsgSharedReq	0	381	388	370
MsgSharedAckData	0	381	324	311
MsgExclusiveReq	0	2	155	207
MsgExclusiveAckData	0	2	36	26
MsgModifyReq	0	256	103	51
MsgModifyAck	0	256	64	29
MsgModifyAckData	0	0	0	0
MsgRInvalidate	0	3	66	60
MsgWInvalidate	0	379	477	522
MsgInvalidateAck	0	125	231	268
MsgInvalidateAckData	0	257	376	375
MsgRelease	0	0	0	0
MsgFlush	0	0	0	0
MsgFlushData	0	0	0	0
Region table lookup	0	382	543	583
(iters)	0	385	543	583
mpc_send	0	2117	2335	2355
mpc_recv	0	79	119	138
mpc_rcvncall	0	2039	2219	2219
mpc_status	0	2080	2291	2299
mpc_wait	0	2164	2395	2430
mpc_lockrnc	176	2344	1981	1739
interrupt handler	0	509	409	327
three message inv	0	0	221	263

Table A.17: CRL and MPL event counts for Water on Three-Message-Invalidation; all values are per-processor averages computed over four consecutive runs.

A.3.3 Floating-Home-Node Application Events

This section contains CRL and MPL event counts for each application running on Floating-Home-Node. Table A.18 contains the counts for Barnes-Hut. Table A.19 contains the counts for Blocked LU. Table A.20 contains the counts for TSP. Table A.21 contains the counts for Water.

Event	1 proc	2 procs	4 procs	8 procs
Map	4744745	2372382	1186207	593118
(remote)	0	594899	593802	373587
(miss)	0	1441	3924	3468
(unmap)	4744745	2372382	1186207	593118
Start read	4650895	2325457	1162745	581388
(remote)	0	598866	596693	375217
(miss)	0	2214	4743	3952
(block)	0	0	2	4
(end)	4650895	2325457	1162745	581388
Start write	128276	64138	33162	17256
(remote)	0	141	1413	1459
(miss)	0	1346	1346	1093
(block)	0	0	0	0
(end)	128276	64138	33162	17256
Flush	0	1979	4428	3639
Become Home	16384	8193	4098	2051
MsgRgnInfoReq	0	1441	3924	3468
(ack)	0	1441	3924	3468
total protocol msgs	0	8018	15807	13401
(queued)	0	701	1347	934
MsgSharedReq	0	2197	4694	3917
MsgSharedAckData	0	2197	4689	3903
MsgExclusiveReq	0	11	41	33
MsgExclusiveAckData	0	11	40	30
MsgModifyReq	0	15	45	42
MsgModifyAck	0	13	35	22
MsgModifyAckData	0	1	1	1
MsgRInvalidate	0	17	54	49
MsgWInvalidate	0	1320	1408	1376
MsgInvalidateAck	0	1317	1395	1363
MsgInvalidateAckData	0	29	96	99
MsgRelease	0	0	1	0
MsgFlush	0	879	3260	2516
MsgFlushData	0	0	3	1
MsgBecomeHomeReq	0	9	23	20
MsgNewHomeInfoWInv	0	0	1	0
MsgNegativeAck	0	0	0	0
MsgBecomeHomeDone	0	0	13	16
forward to real home	0	0	12	16
Region table lookup	4744745	2375160	1191615	598039
(iters)	22628546	6181411	1875658	659096
`mpc_send`	0	10982	23782	20483
`mpc_recv`	0	84	126	147
`mpc_rcvncall`	0	10900	23658	20339
`mpc_status`	0	10943	23728	20422
`mpc_wait`	0	11033	23853	20565
`mpc_lockrnc`	186	18943	44343	37243
interrupt handler	0	5314	11184	8428
three message inv	0	0	16	37

Table A.18: CRL and MPL event counts for Barnes-Hut on Floating-Home-Node; all values are per-processor averages computed over four consecutive runs.

Event	1 proc	2 procs	4 procs	8 procs
Map	84576	42314	21489	10770
(remote)	0	10288	5476	4044
(miss)	0	638	650	656
(unmap)	84576	42314	21489	10770
Start read	41701	20876	10757	5392
(remote)	0	10288	5463	4025
(miss)	0	638	638	637
(block)	0	0	0	0
(end)	41701	20876	10757	5392
Start write	42925	21463	10731	5366
(remote)	0	0	0	0
(miss)	0	0	0	0
(block)	0	0	0	0
(end)	42925	21463	10731	5366
Flush	0	0	0	0
MsgRgnInfoReq	0	638	650	656
(ack)	0	638	650	656
total protocol msgs	0	1275	1275	1274
(queued)	0	9	14	5
MsgSharedReq	0	638	638	637
MsgSharedAckData	0	638	638	637
MsgExclusiveReq	0	0	0	0
MsgExclusiveAckData	0	0	0	0
MsgModifyReq	0	0	0	0
MsgModifyAck	0	0	0	0
MsgModifyAckData	0	0	0	0
MsgRInvalidate	0	0	0	0
MsgWInvalidate	0	0	0	0
MsgInvalidateAck	0	0	0	0
MsgInvalidateAckData	0	0	0	0
MsgRelease	0	0	0	0
MsgFlush	0	0	0	0
MsgFlushData	0	0	0	0
Region table lookup	84576	42951	22139	11426
(iters)	84576	49877	26391	10967
`mpc_send`	0	2844	3016	3100
`mpc_recv`	0	297	446	520
`mpc_rcvncall`	0	2550	2575	2586
`mpc_status`	0	2702	2802	2851
`mpc_wait`	0	3009	3264	3389
`mpc_lockrnc`	630	4473	4531	4534
interrupt handler	0	1272	1200	1176
three message inv	0	0	0	0

Table A.19: CRL and MPL event counts for Blocked LU on Floating-Home-Node; all values are per-processor averages computed over four consecutive runs.

Event	1 proc	2 procs	4 procs	8 procs
Map	0	0	0	0
(remote)	0	0	0	0
(miss)	0	0	0	0
(unmap)	0	0	0	0
Start read	34323	10019050	1721496	337345
(remote)	0	23074	17439	14961
(miss)	0	3	7	8
(block)	0	0	0	0
(end)	34323	10019050	1721496	337345
Start write	17180	8590	4297	2150
(remote)	0	8589	4296	2150
(miss)	0	3	2390	1652
(block)	0	0	0	0
(end)	17180	8590	4297	2150
Flush	0	0	0	0
MsgRgnInfoReq	0	0	0	0
(ack)	0	0	0	0
total protocol msgs	0	11	7189	4982
(queued)	0	2	1212	685
MsgSharedReq	0	3	7	8
MsgSharedAckData	0	3	5	7
MsgExclusiveReq	0	2	2389	1652
MsgExclusiveAckData	0	2	1	0
MsgModifyReq	0	1	1	0
MsgModifyAck	0	1	0	0
MsgModifyAckData	0	0	0	0
MsgRInvalidate	0	1	2	1
MsgWInvalidate	0	1	2390	1655
MsgInvalidateAck	0	1	3	4
MsgInvalidateAckData	0	1	2391	1653
MsgRelease	0	0	0	0
MsgFlush	0	0	0	0
MsgFlushData	0	0	0	0
Region table lookup	0	1	2392	1656
(iters)	0	1	2392	1656
`mpc_send`	0	153	7402	5230
`mpc_recv`	0	145	218	254
`mpc_rcvncall`	0	11	7189	4982
`mpc_status`	0	87	7319	5128
`mpc_wait`	0	243	7535	5385
`mpc_lockrnc`	326	343	8283	6352
interrupt handler	0	5	4246	2357
three message inv	0	0	2390	1652

Table A.20: CRL and MPL event counts for TSP on Floating-Home-Node; all values are per-processor averages computed over four consecutive runs.

Event	1 proc	2 procs	4 procs	8 procs
Map	0	0	0	0
(remote)	0	0	0	0
(miss)	0	0	0	0
(unmap)	0	0	0	0
Start read	263682	131842	65922	32962
(remote)	0	32769	24578	14338
(miss)	0	384	389	375
(block)	0	0	0	0
(end)	263682	131842	65922	32962
Start write	5640	3078	1669	965
(remote)	0	258	259	260
(miss)	0	638	482	379
(block)	0	0	0	0
(end)	5640	3078	1669	965
Flush	0	0	0	0
MsgRgnInfoReq	0	0	0	0
(ack)	0	0	0	0
total protocol msgs	0	2044	2217	2229
(queued)	0	70	43	26
MsgSharedReq	0	382	388	374
MsgSharedAckData	0	382	324	311
MsgExclusiveReq	0	2	153	202
MsgExclusiveAckData	0	2	38	29
MsgModifyReq	0	256	105	57
MsgModifyAck	0	256	64	28
MsgModifyAckData	0	0	0	0
MsgRInvalidate	0	2	66	64
MsgWInvalidate	0	381	475	519
MsgInvalidateAck	0	126	232	270
MsgInvalidateAckData	0	257	373	376
MsgRelease	0	0	0	0
MsgFlush	0	0	0	0
MsgFlushData	0	0	0	0
Region table lookup	0	383	541	582
(iters)	0	386	541	582
mpc_send	0	2122	2333	2364
mpc_recv	0	79	119	138
mpc_rcvncall	0	2044	2217	2229
mpc_status	0	2085	2291	2311
mpc_wait	0	2169	2392	2437
mpc_lockrnc	176	2357	1989	1750
interrupt handler	0	519	411	333
three message inv	0	0	220	264

Table A.21: CRL and MPL event counts for Water on Floating-Home-Node; all values are per-processor averages computed over four consecutive runs.

Appendix B

Implementation Details of the Three-Message-Invalidate Protocol

This appendix contains a more in-depth description of the Three-Message-Invalidate protocol than was given in Chapter 5. The first section describes the protocol states and events. The second and third sections describe the home- and remote-side protocol state machines, respectively. Since this protocol's base was CRL's original protocol, which was described in [18], annotations and markings appear in the state diagrams and pseudocode to indicate what was changed from this base. Readers who 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.

B.1 Protocol States and Events

The CRL Three-Message-Invalidate protocol uses a total of seventeen states, nine for the home-side state machine and eight for the remote-side state machine. The original CRL protocol also uses every one of these states, except for the new home state HomeThreeWaylipSpecial. These states are described in Tables B.1 and B.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 B.3 describes the five types of call events used in CRL. In the original CRL implementation, protocol messages in CRL were 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 were never exchanged between remote nodes. The Three-Message-Invalidate protocol requires the use of remote-to-remote messages in order to provide better performance. Precisely, these messages are remote-to-requesting messages since remote nodes

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 for the home node)
HomeIipSpecial	An invalidation of remote copies of the region is in progress (to obtain a shared copy for the home node)
HomeInvalid	A single remote node has a valid copy of the region
HomeThreeWayIipSpecial	An invalidation of remote copies of the region is in progress (to obtain a shared copy for a remote node)

Table B.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 B.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 B.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 B.4: CRL home-to-remote protocol messages.

send these messages to nodes requesting access, regardless of whether the requesting node is the home node or a remote node. Two of the previous remote-to-home messages have become remote-to-request messages.

Table B.4 describes the types of protocol messages sent from home nodes to remote nodes (six types of messages); Table B.5 describes those sent from remote nodes to home nodes (six types of messages). Table B.6 describes those sent from remote nodes to requesting nodes (two types of messages).

B.2 Home-Side State Machine

Figures B-1 through B-9 show the state transition diagrams for the nine 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).

Message	Description
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 B.5: CRL remote-to-home 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)

Table B.6: CRL remote-to-requesting protocol messages.

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. 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). Note that the HomeThreeWayIipSpecial state does not appear in every state transition diagram. In order to make the diagrams less cluttered, it has only been added to Figures B-8 and B-9, which are the only two diagrams which make use of the state.

Figures B-1 through B-9 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 B.7 through B.16. These tables provide pseudocode for the actions taken in response to different protocol events for each of the nine home-side states. Any events that are not listed for a particular state cause a protocol error if they occur (as in Figures B-1 through B-9).

Each of Tables B.7 through B.16 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 B.7 through B.16. 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).

requesting structure list: In the Three-Message-Invalidate protocol, the home-side metadata for a region contains a list of previous requesters for that region. The list is used to redirect flush messages that were sent by remote nodes before those nodes received an invalidate message. Operations supported on the requesting structure lists include insertion of a new structure (insert requesting structure), looking up a structure within the list (get requesting structure for flushed version) and deleting the whole list (delete requesting structure list).

As mentioned earlier, the Three-Message-Invalidate protocol was built using CRL's original protocol as the base. Similarly, the state diagrams and pseudocode from the original protocol were used as a base for this appendix, and changes were marked as follows:

- Additions to the state diagram are bolded. This includes both the event names, and the arrows.
- Additions to the pseudocode are boxed.
- Deletions are crossed out.

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

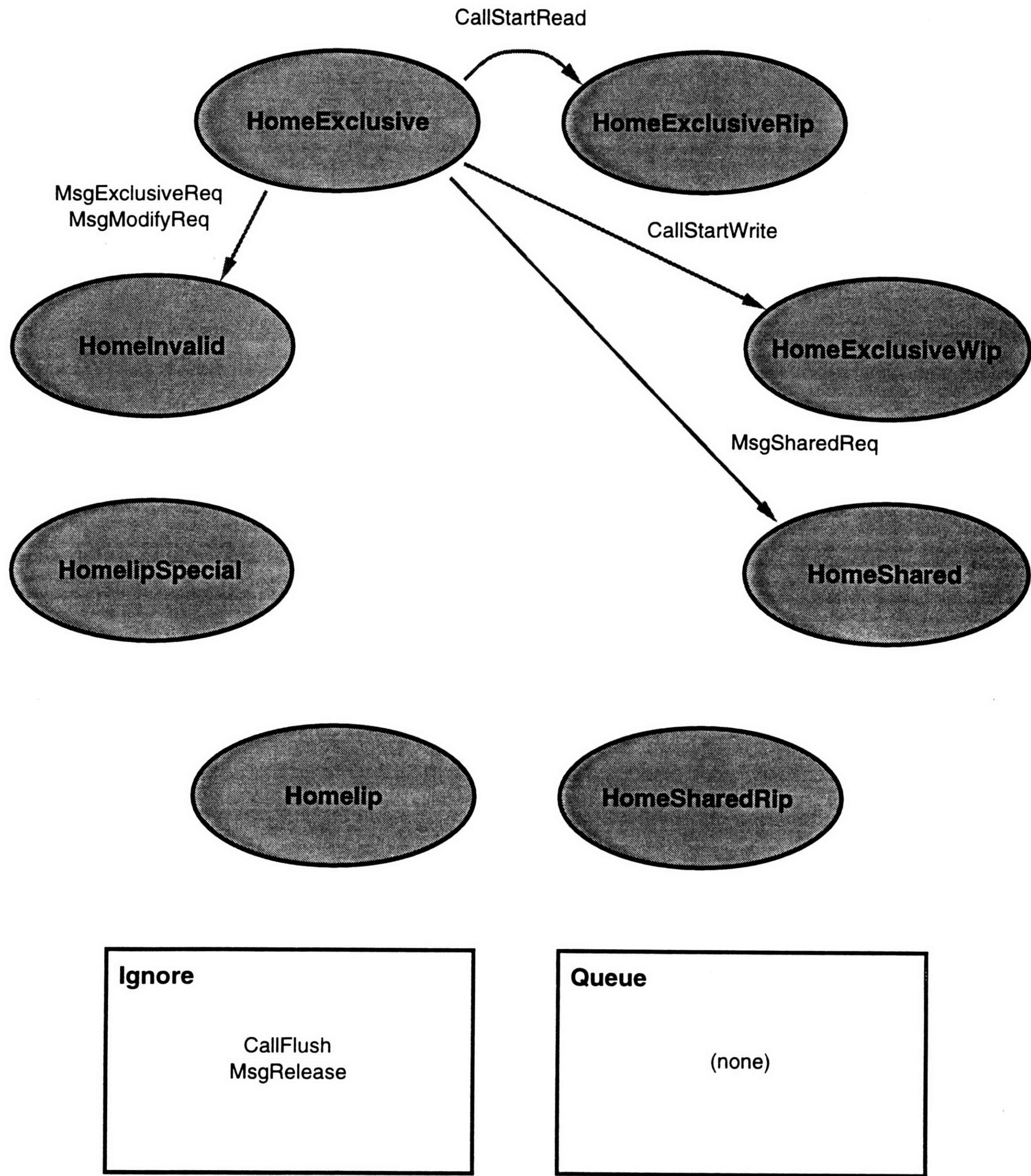

Figure B-1: HomeExclusive: state transition diagram.

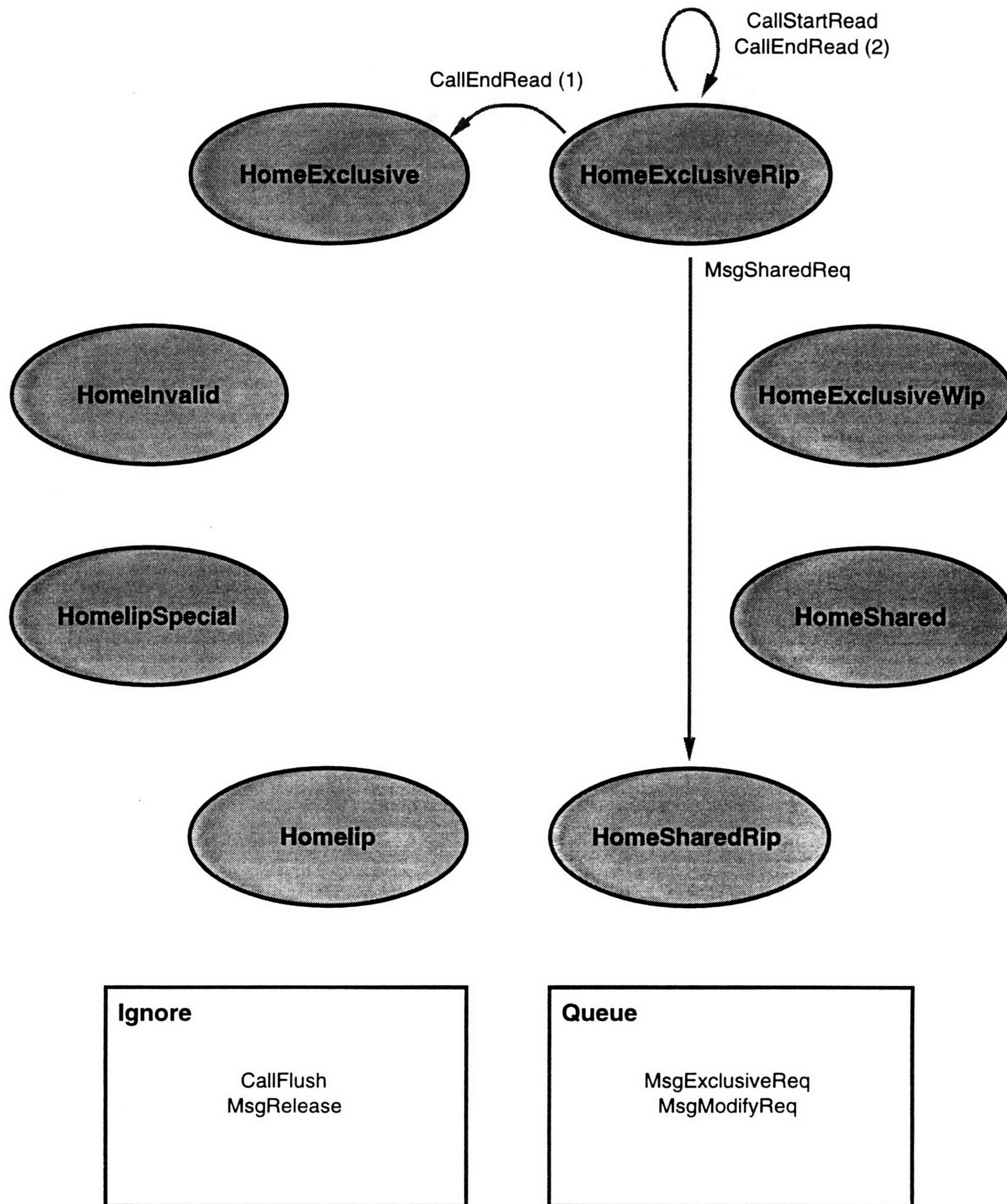

Figure B-2: HomeExclusiveRip: state transition diagram.

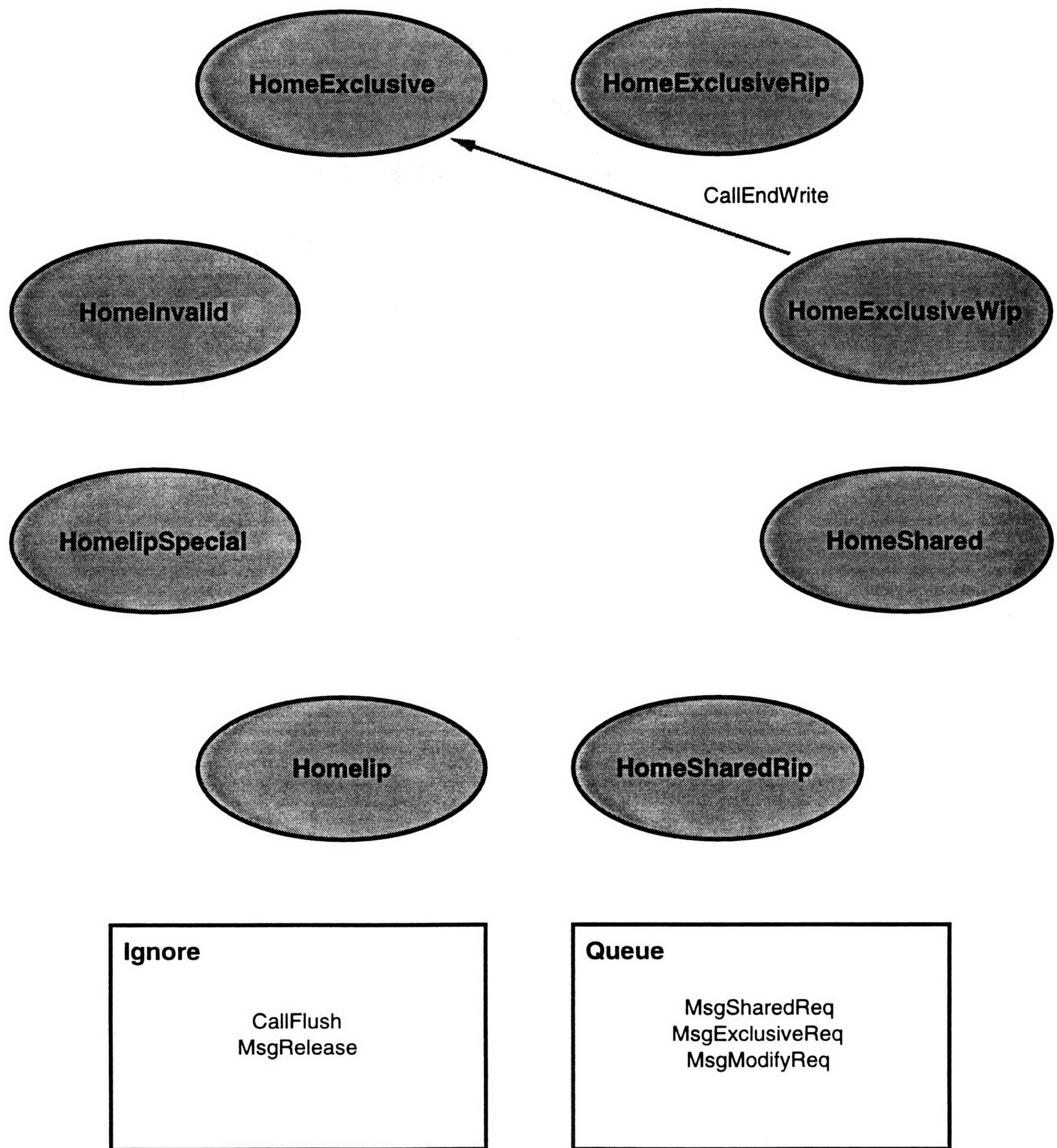

Figure B-3: HomeExclusiveWip: state transition diagram.

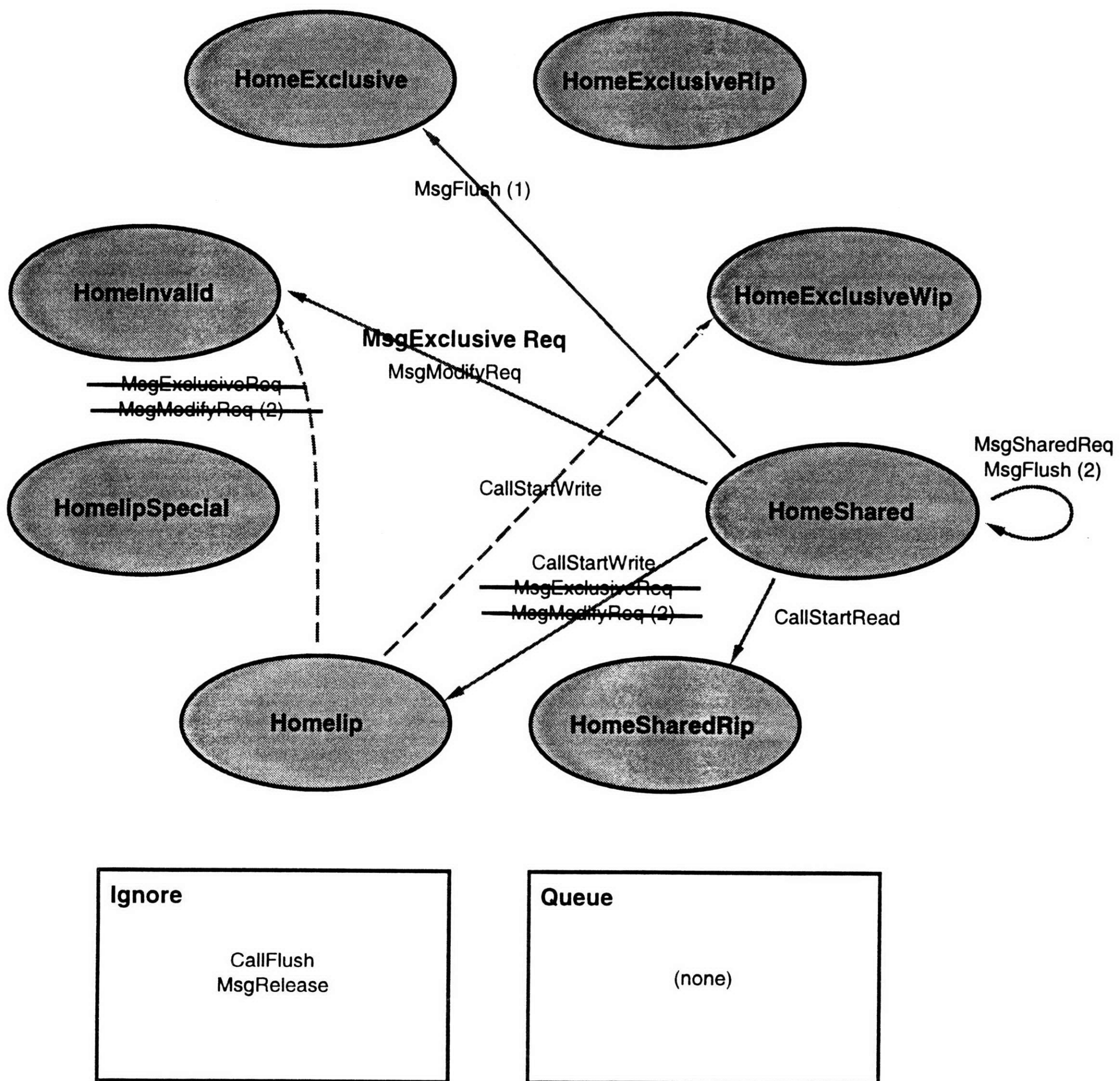

Figure B-4: HomeShared: state transition diagram.

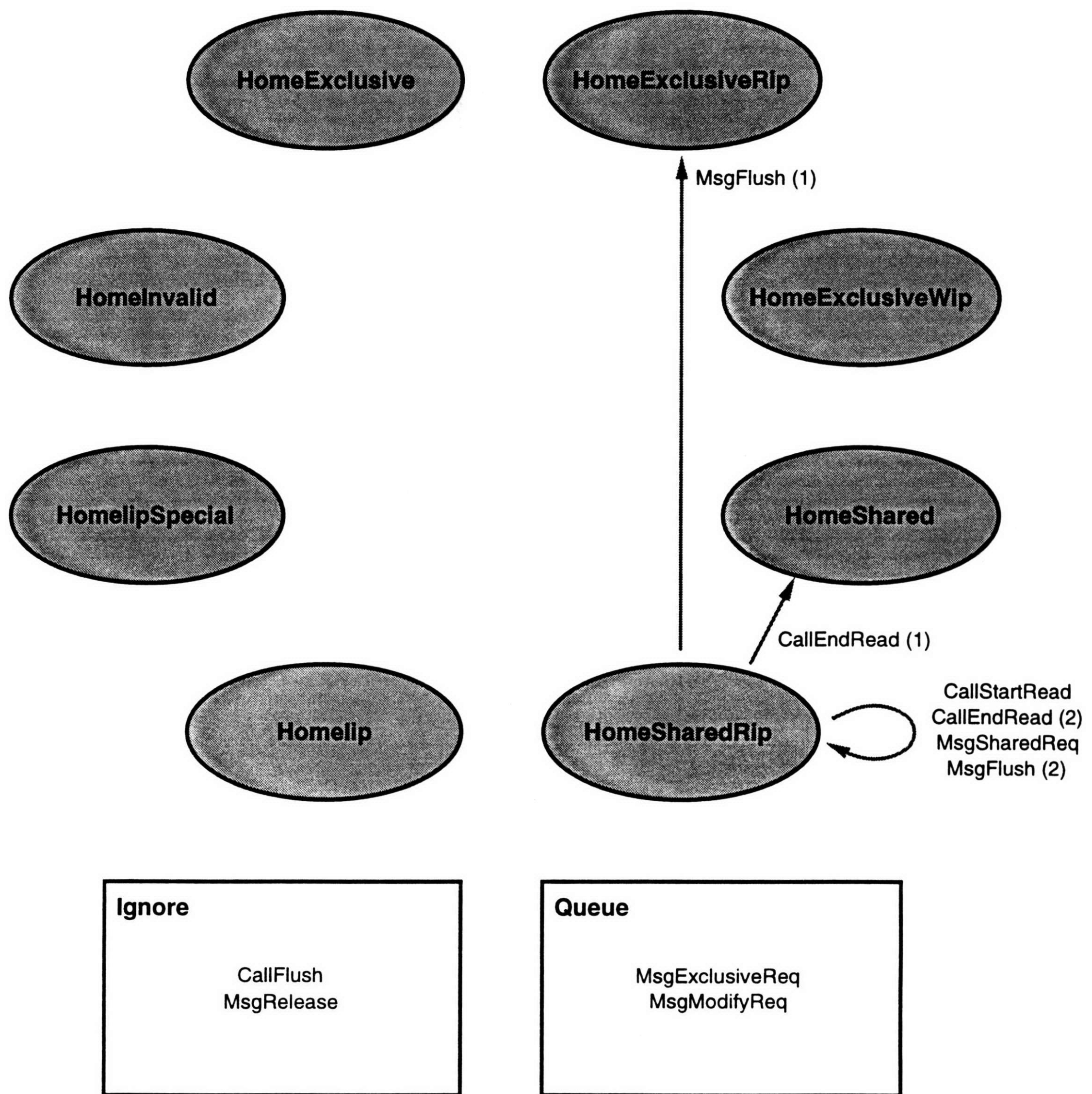

Figure B-5: HomeSharedRip: state transition diagram.

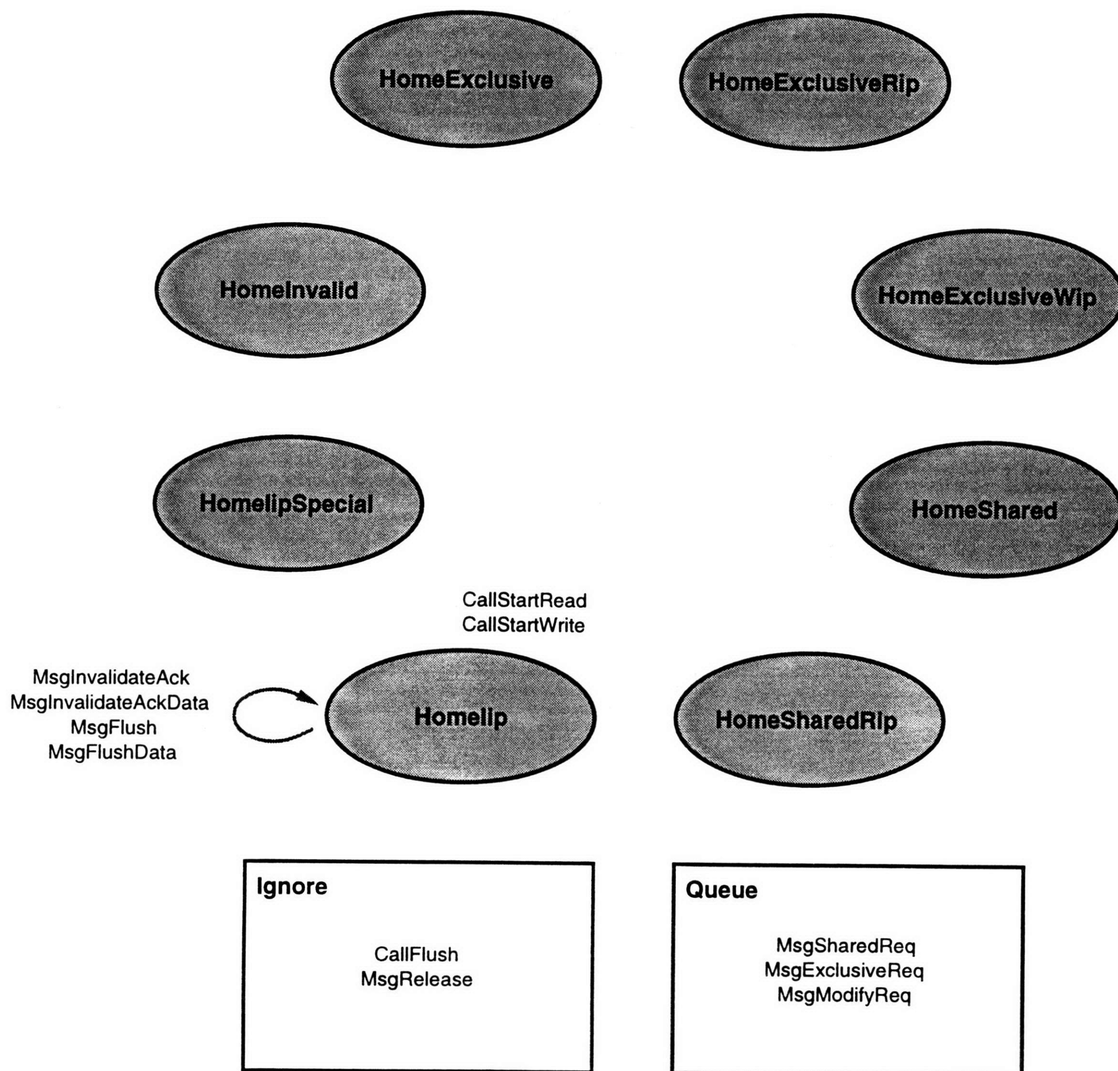

Figure B-6: HomeIip: state transition diagram.

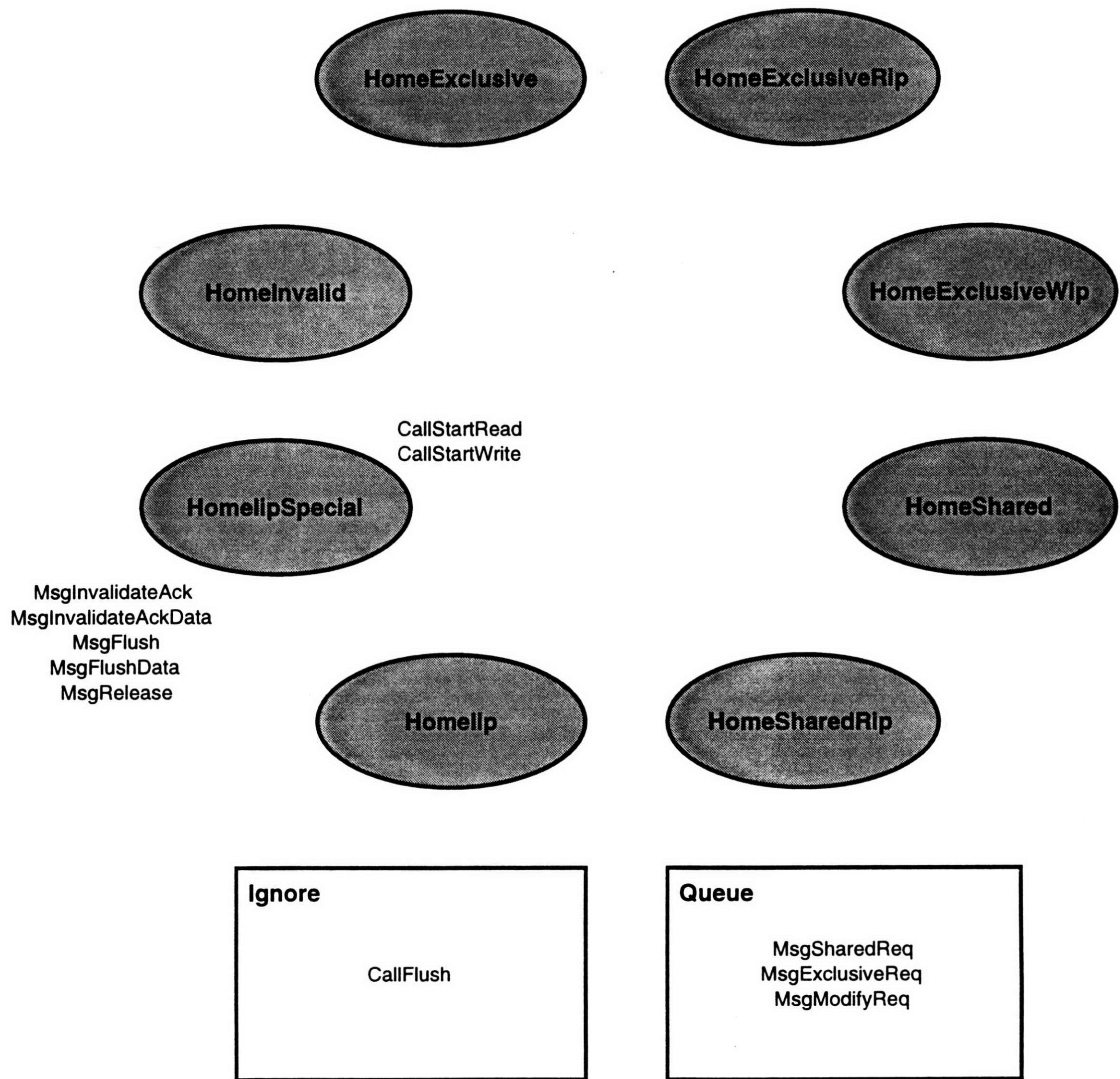

Figure B-7: HomeIipSpecial: state transition diagram.

Figure B-8: HomeInvalid: state transition diagram.

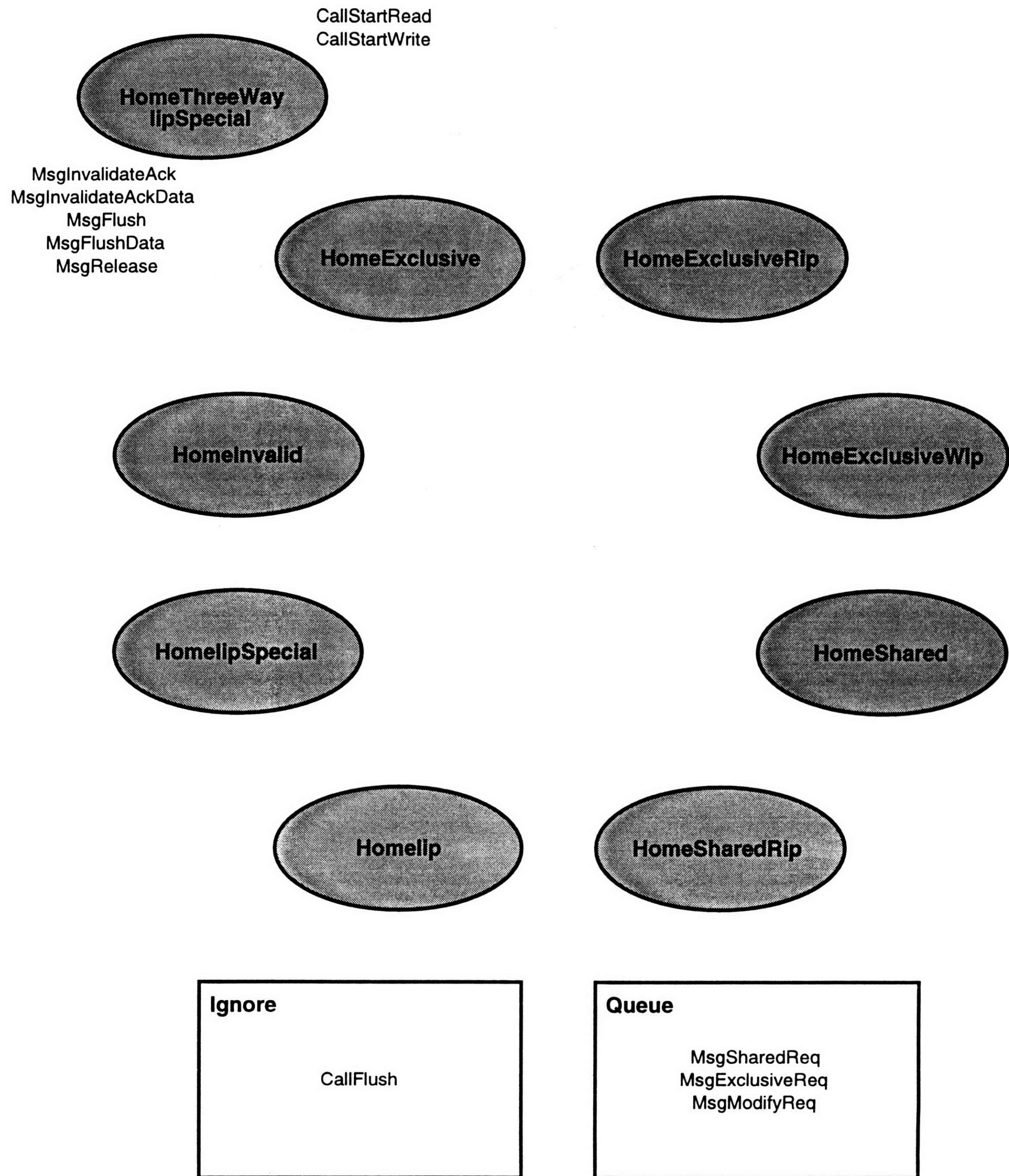

Figure B-9: HomeThreeWayIipSpecial: state transition diagram.

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 MsgModifyAckData insert pointer *state* = HomeInvalid
	MsgRelease	do nothing

Table B.7: **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 B.8: **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 B.9: 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* = Homelip 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* = Homelip~~ insert pointer insert requesting structure *state* = HomeInvalid
	~~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* = Homelip~~
		  insert pointer
		  insert requesting structure
		  state = HomeInvalid

	~~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 B.10: 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 B.11: **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
		num_ptrs -= 1
		if (*num_ptrs* == 0)
		  delete requesting structure list
		  invoke *tx_cont*

	MsgFlush,
	MsgFlushData
		~~*num_ptrs* -= 1~~
		~~if (*num_ptrs* == 0)~~
		~~invoke *tx_cont*~~
		if (flush for most recent version)
		  num_ptrs -= 1
		  if (*num_ptrs* == 0)
		    delete requesting structure list
		    invoke *tx_cont*
		else
		  get requesting structure for flushed version
		  if (flushing node was sending data)
		    send MsgInvalidateAckData to requester
		  else
		    send MsgInvalidateAck to requester

	MsgSharedReq, MsgExclusiveReq, MsgModifyReq	queue message
	MsgRelease	do nothing

Table B.12: **HomeIip**: protocol events and actions.

State	Event	Actions
HomelipSpecial	CallStartRead	wait until *state* != HomelipSpecial retry CallStartRead
	CallStartWrite	wait until *state* != HomelipSpecial retry CallStartWrite
	CallFlush	do nothing
	MsgInvalidateAck, MsgInvalidateAckData	delete requesting structure list invoke *tx_cont* with an arg of 0

	MsgFlush,
	MsgFlushData
		if (flush sent after a MsgRelease)
		rcvd_flush |= OWNERFLUSH
		else if (flush sent before a MsgRelease)
		delete requesting structure list
		invoke *tx_cont* with an arg of 0
		else
		get requesting structure for flushed version
		if (flushing node was sending data)
		send MsgInvalidateAckData to requester
		else
		send MsgInvalidateAck to requester

	MsgRelease	~~invoke *tx_cont* with an arg of 1~~ if (MsgRelease is for most recent version) delete requesting structure list invoke *tx_cont* with an arg of 1
	MsgSharedReq, MsgExclusiveReq, MsgModifyReq	queue message

Table B.13: HomelipSpecial: 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~~ *state* = HomeThreeWayIipSpecial

	cont:MsgSharedReq	~~send MsgSharedAckData~~
		~~insert pointer~~
		~~*state* = HomeShared~~
		if (*num_ptrs* > 0)
		  state = HomeShared
		else
		  state = HomeExclusive

	MsgExclusiveReq	send MsgWInvalidate to remote copy ~~*tx_cont* = cont:MsgExclusiveReq~~ ~~*state* = HomeIip~~ insert pointer insert requesting structure
	~~cont:MsgExclusiveReq~~	~~send MsgExclusiveAckData~~ ~~insert pointer~~ ~~*state* = HomeInvalid~~ ~~retry queued messages~~
	HomeInvalid: protocol events and actions continues in Table B.15	

Table B.14: HomeInvalid: protocol events and actions.

State	Event/Continuation	Actions
HomeInvalid	**HomeInvalid**: protocol events and actions continued from Table B.14	
	MsgModifyReq	send MsgWInvalidate to remote copy ~~*tx_cont* = cont:MsgModifyReq~~ ~~*state* = HomeIp~~ insert pointer insert requesting structure
	~~cont:MsgModifyReq~~	~~send MsgModifyAckData~~ ~~insert pointer~~ ~~*state* = HomeInvalid~~ ~~retry queued messages~~

	MsgFlush,
	MsgFlushData
		~~delete pointer~~
		~~*state* = HomeExclusive~~
		if (most recent requester flushed)
		  delete requesting structure list
		  delete pointer
		  state = HomeExclusive
		else
		  get requesting structure for flushed version
		  if (flushing node was sending data)
		    send MsgInvalidateAckData to requester
		  else
		    send MsgInvalidateAck to requester

	MsgRelease	do nothing

Table B.15: HomeInvalid: protocol events and actions (continued).

State	Event	Actions
HomeThreeWaylipSpecial	CallStartRead	wait until *state* != HomeThreeWaylipSpecial retry CallStartRead
	CallStartWrite	wait until *state* != HomeThreeWaylipSpecial retry CallStartWrite
	CallFlush	do nothing
	MsgInvalidateAck, MsgInvalidateAckData	if ((*rcvd_flush* & REQUESTERFLUSH) != 0) insert pointer delete requesting structure list invoke *tx_cont* with an arg of 0

	MsgFlush,
	MsgFlushData
		if (flush sent by previous owner after a MsgRelease)
		  rcvd_flush |= OWNERFLUSH
		else if (flush sent by requester)
		  rcvd_flush |= REQUESTERFLUSH
		else if (flush sent before a MsgRelease)
		  insert pointer
		  send MsgSharedAckData to requester
		  delete requesting structure list
		  invoke *tx_cont* with an arg of 0
		else
		  get requesting structure for flushed version
		  if (flushing node was sending data)
		    send MsgInvalidateAckData to requester
		  else
		    send MsgInvalidateAck to requester

	MsgRelease	if (MsgRelease is for most recent version)
		  if ((*rcvd_flush* & OWNERFLUSH) != 0)
		    delete pointer
		  if ((*rcvd_flush* & REQUESTERFLUSH) == 0)
		    insert pointer
		  delete requesting structure list
		  invoke *tx_cont* with an arg of 1

	MsgSharedReq, MsgExclusiveReq, MsgModifyReq	queue message

Table B.16: HomeThreeWaylipSpecial: protocol events and actions.

B.3 Remote-Side State Machine

Figures B-10 through B-17 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 B-1 through B-9), with one minor difference. 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), the **Queue** box is instead labeled **Set rcvd_inv flag**.

As was the case in Figures B-1 through B-9 (for the home-side state machine), Figures B-10 through B-17 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 pseudocode) can be found in Tables B.17 through B.24.

Each of Tables B.17 through B.24 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, three other components of the remote-side protocol metadata associated with each region are referenced in Tables B.17 through B.24. 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.

num_invalidate_acks: In the Three-Message-Invalidate protocol, this field is used to count the number of invalidate acknowledgment messages that will be arriving.

As mentioned earlier, the Three-Message-Invalidate protocol was built using CRL's original protocol as the base. Similarly, the state diagrams and pseudocode from the original protocol were used as a base for this appendix, and changes were marked as follows:

- Additions to the state diagram are bolded. This includes both the event names, and the arrows.
- Additions to the pseudocode are boxed.
- Deletions are crossed out.

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

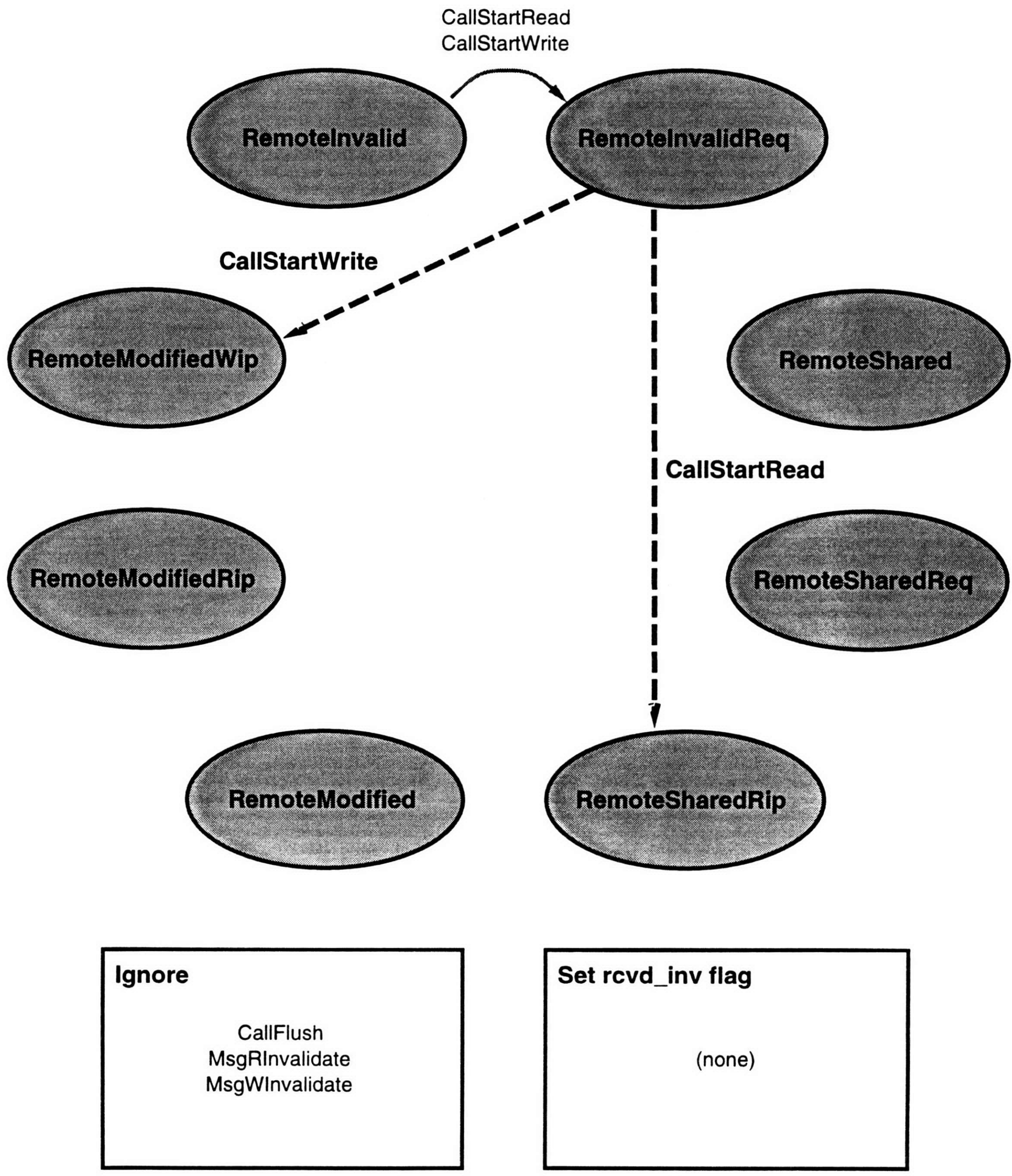

Figure B-10: RemoteInvalid: state transition diagram.

Figure B-11: RemoteInvalidReq: state transition diagram.

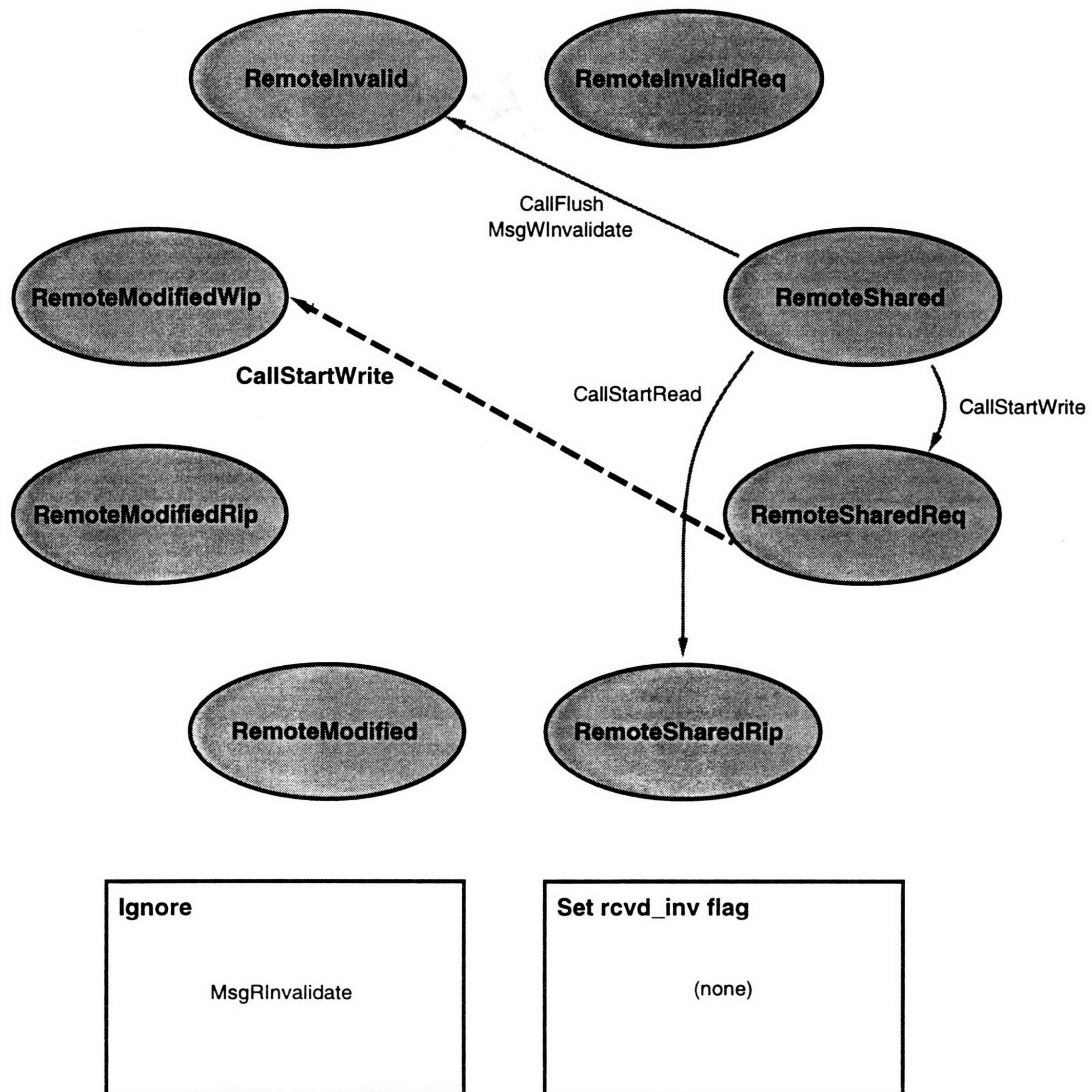

Figure B-12: RemoteShared: state transition diagram.

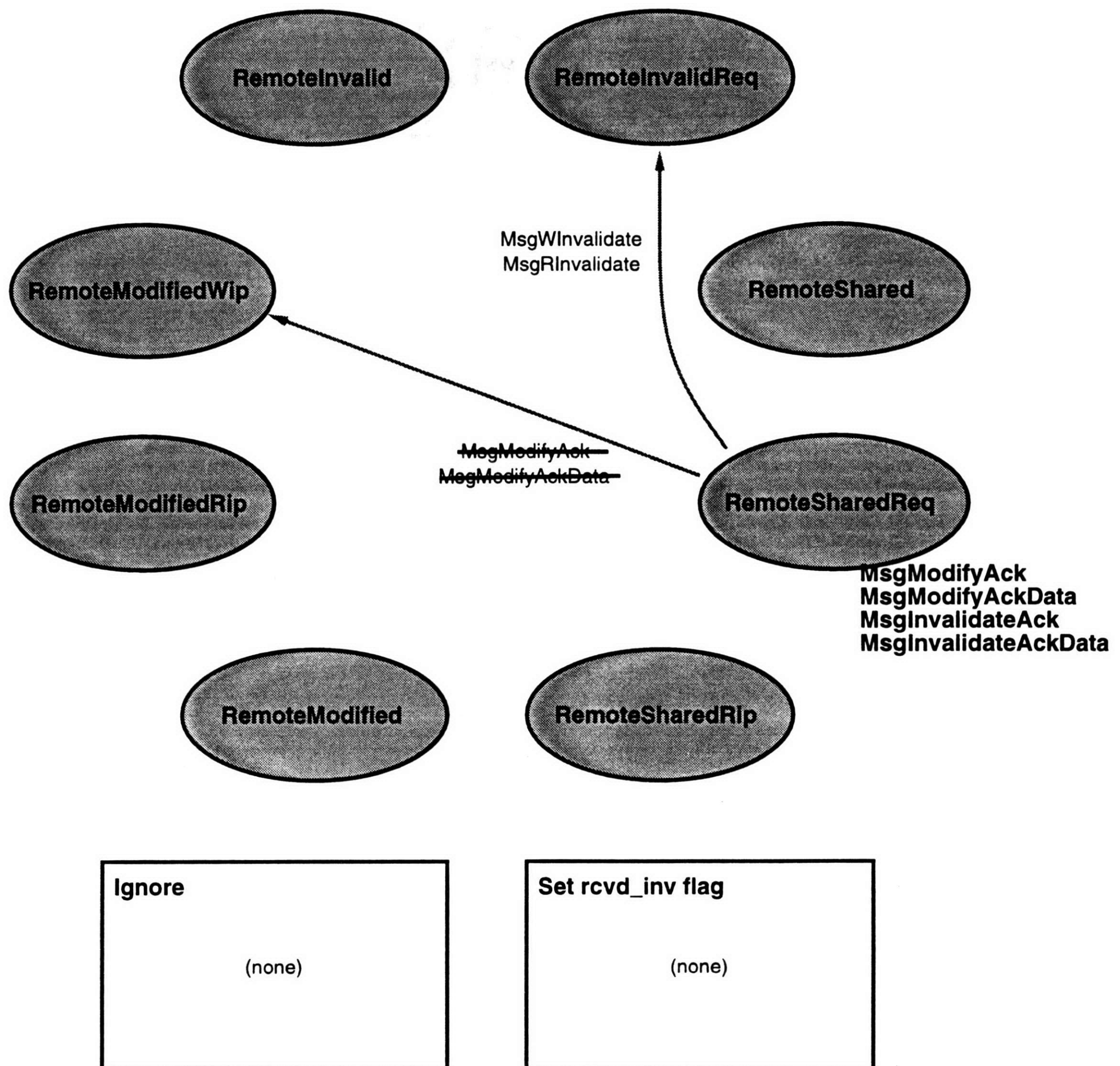

Figure B-13: RemoteSharedReq: state transition diagram.

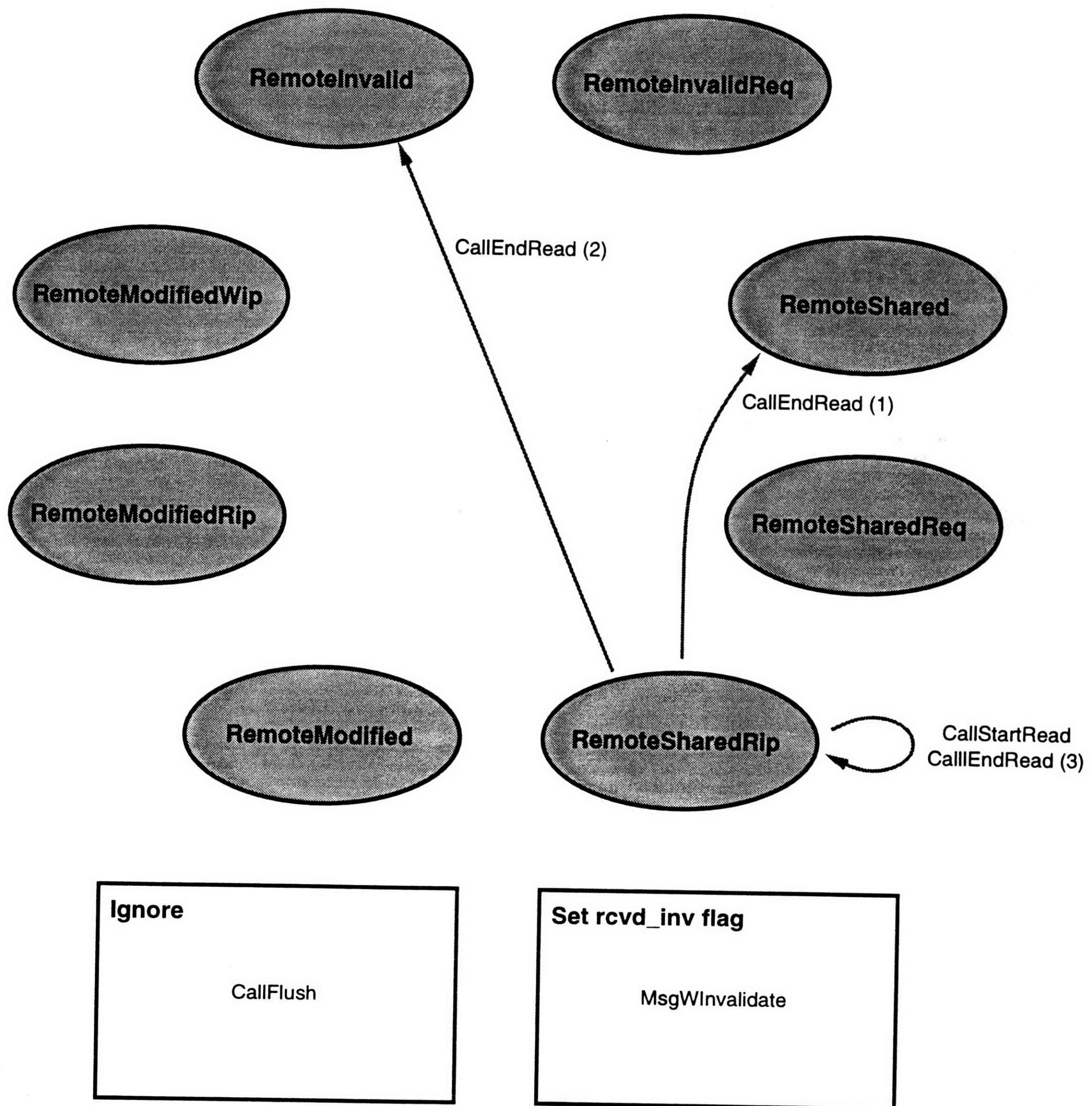

Figure B-14: RemoteSharedRip: state transition diagram.

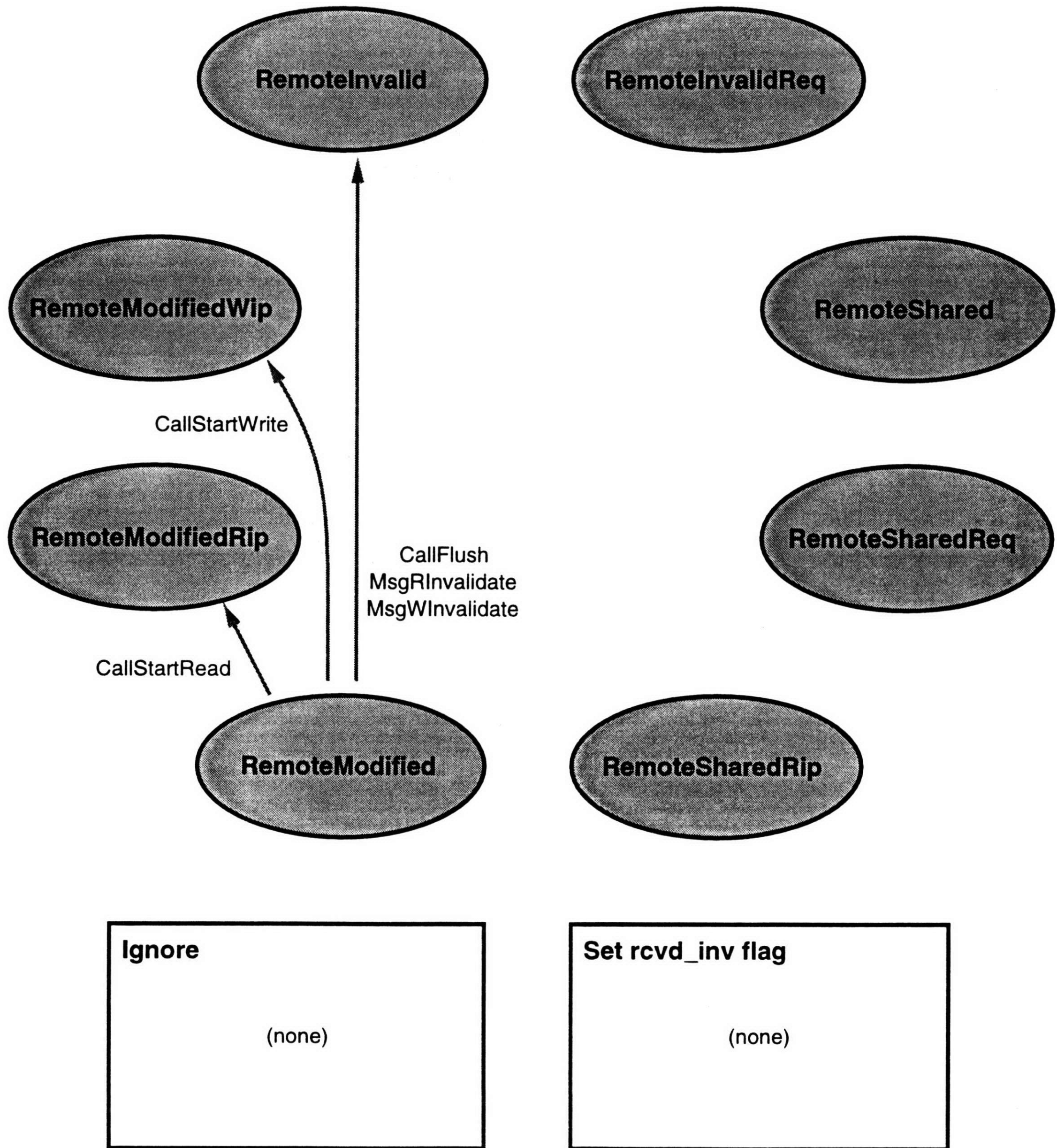

Figure B-15: RemoteModified: state transition diagram.

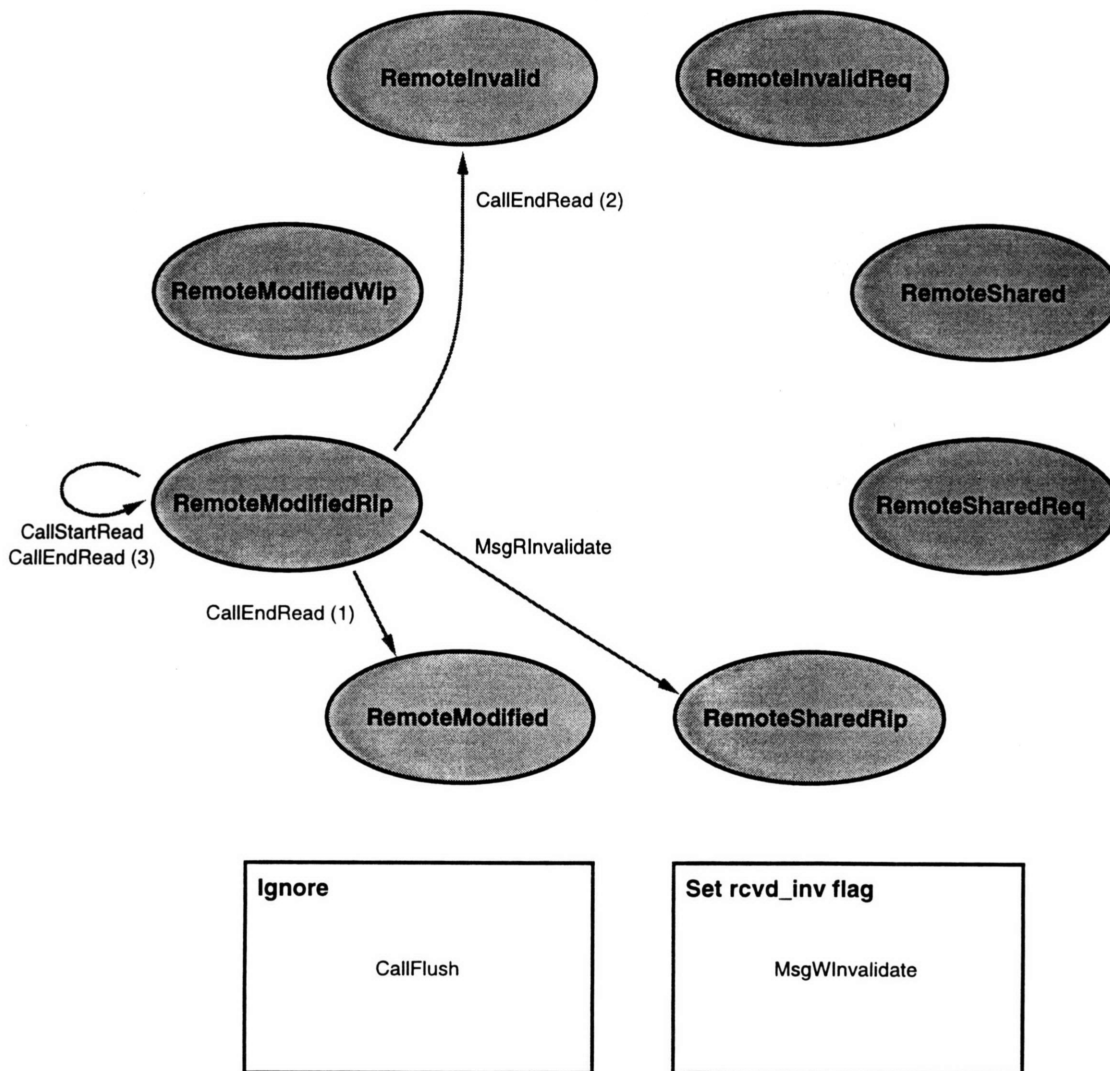

Figure B-16: RemoteModifiedRip: state transition diagram.

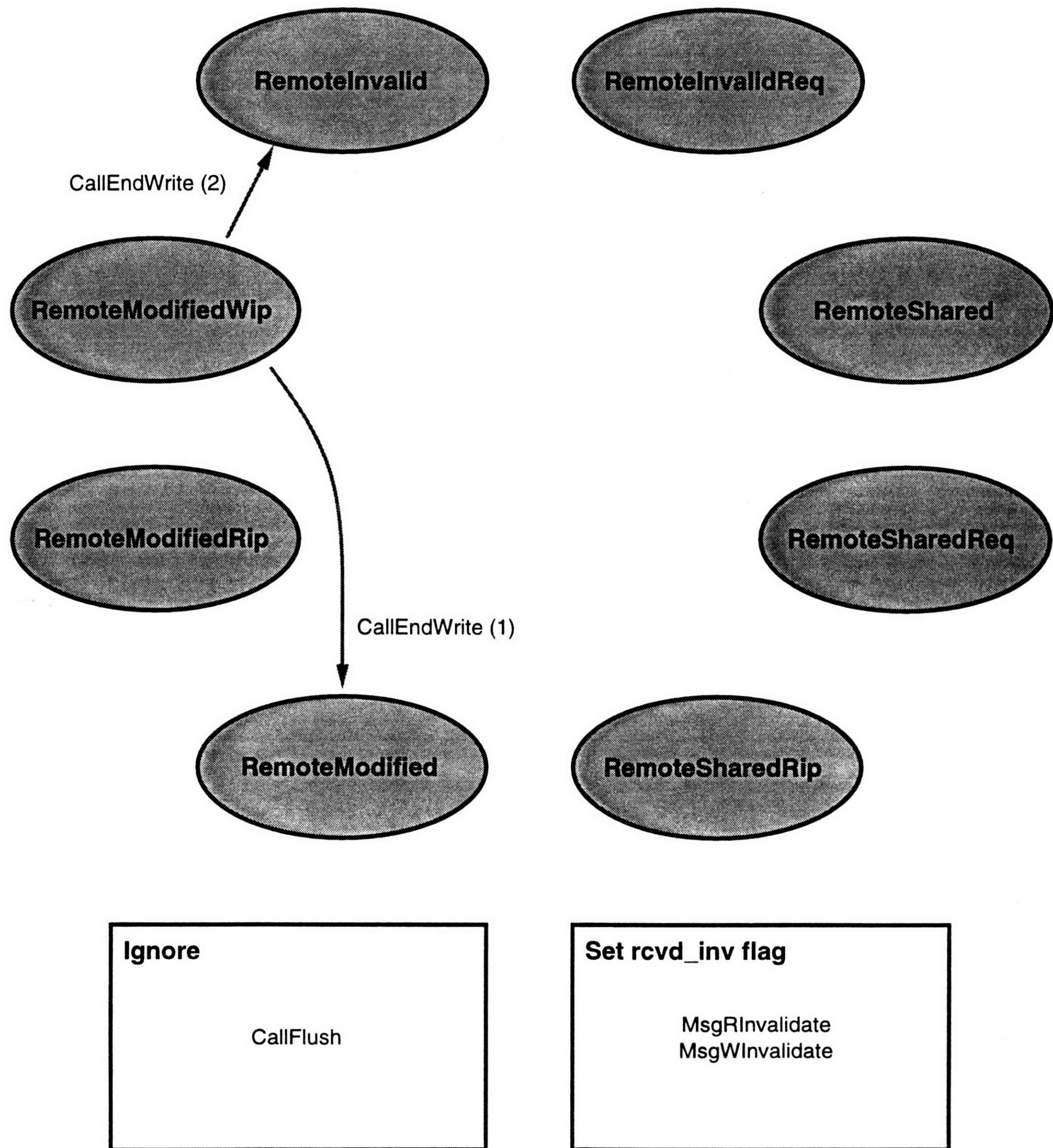

Figure B-17: RemoteModifiedWip: state transition diagram.

State	Event	Actions
RemoteInvalid	CallStartRead	send MsgSharedReq to home *tx_cont* = cont:CallStartRead *state* = RemoteInvalidReq ~~poll until reply is received~~ poll until *tx_cont* has been invoked
	cont:CallStartRead	*read_cnt* = 1 *state* = RemoteSharedRip
	CallStartWrite	send MsgExclusiveReq to home *tx_cont* = cont:CallStartWrite *state* = RemoteInvalidReq ~~poll until reply is received~~ poll until *tx_cont* has been invoked
	cont:CallStartWrite	*state* = RemoteModifiedWip
	CallFlush	do nothing
	MsgRInvalidate, MsgWInvalidate	do nothing

Table B.17: RemoteInvalid: protocol events and actions.

State	Event	Actions
RemoteInvalidReq	MsgSharedAckData, MsgExclusiveAckData, MsgModifyAckData	invoke *tx_cont*

	MsgInvalidateAck,
	MsgInvalidateAckData
		if (*num_invalidate_acks* is unset)
		num_invalidate_acks = # of MsgInvalidateAcks expected
		num_invalidate_acks -= 1
		if (*num_invalidate_acks* == 0)
		invoke *tx_cont*

	MsgRInvalidate, MsgWInvalidate	do nothing

Table B.18: RemoteInvalidReq: protocol events and actions.

State	Event	Actions
RemoteShared	CallStartRead	*read_cnt* = 1 *state* = RemoteSharedRip
	CallStartWrite	send MsgModifyReq to home *tx_cont* = cont:CallStartWrite *state* = RemoteSharedReq ~~poll until reply is received~~ poll until *tx_cont* has been invoked
	cont:CallStartWrite	*state* = RemoteModifiedWip
	CallFlush	send MsgFlush to home *state* = RemoteInvalid
	MsgRInvalidate	do nothing

	MsgWInvalidate	~~send MsgInvalidateAck to home~~
		if (need to send data)
		send MsgInvalidateAckData to requester
		else
		send MsgInvalidateAck to requester
		state = RemoteInvalid

Table B.19: RemoteShared: protocol events and actions.

State	Event	Actions

RemoteSharedReq	MsgInvalidateAck,
	MsgInvalidateAckData
		if (*num_invalidate_acks* is unset)
		num_invalidate_acks = # of MsgInvalidateAcks expected
		num_invalidate_acks -= 1
		if (*num_invalidate_acks* == 0)
		invoke *tx_cont*

	MsgWInvalidate,
	MsgRInvalidate
		~~send MsgInvalidateAck to home~~
		if (need to send data)
		send MsgInvalidateAckData to requester
		else
		send MsgInvalidateAck to requester
		state = RemoteInvalidReq

	MsgModifyAck, MsgModifyAckData	~~state = RemoteModifiedWip~~ invoke *tx_cont*

Table B.20: 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~~
		    if (need to send data)
		      send MsgInvalidateAckData to requester
		    else
		      send MsgInvalidateAck to requester
		    rcvd_inv = 0
		    state = RemoteInvalid

	CallFlush	do nothing
	MsgWInvalidate	*rcvd_inv* = WInvalidate

Table B.21: 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	~~send MsgInvalidateAckData to home~~ send MsgInvalidateAckData to requester if (requester != home) send MsgInvalidateAckData to home *state* = RemoteInvalid
	MsgWInvalidate	~~send MsgInvalidateAckData to home~~ send MsgInvalidateAckData to requester *state* = RemoteInvalid

Table B.22: 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~~
		send MsgInvalidateAckData to requester
		if ((*rcvd_inv* == RInvalidate) AND (requester != home))
		send MsgInvalidateAckData to home
		rcvd_inv = 0
		state = RemoteInvalid

	CallFlush	do nothing
	MsgRInvalidate	send MsgRelease to home if (requester != home) send MsgInvalidateAckData to requester *state* = RemoteSharedRip
	MsgWInvalidate	*rcvd_inv* = WInvalidate

Table B.23: RemoteModifiedRip: protocol events and actions.

State	Event	Actions

RemoteModifiedWip	CallEndWrite	if (*rcvd_inv* == 0)
		state = RemoteModified
		else
		~~send MsgInvalidateAckData to home~~
		send MsgInvalidateAckData to requester
		if ((*rcvd_inv* == RInvalidate) AND (requester != home))
		send MsgInvalidateAckData to home
		rcvd_inv = 0
		state = RemoteInvalid

	CallFlush	do nothing
	MsgRInvalidate	*rcvd_inv* = RInvalidate
	MsgWInvalidate	*rcvd_inv* = WInvalidate

Table B.24: RemoteModifiedWip: protocol events and actions.

Appendix C

Implementation Details of the Floating-Home-Node Protocol

This appendix contains a more in-depth description of the Floating-Home-Node protocol than was given in Chapter 6. The first section describes the protocol states and events. The second and third sections describe the home- and remote-side protocol state machines, respectively. Since this protocol's base was the Three-Message-Invalidate protocol, which was described in Appendix Chapter B, annotations and markings appear in the state diagrams and pseudocode to indicate what was changed from this base. 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.

C.1 Protocol States and Events

The CRL Floating-Home-Node protocol uses a total of nineteen states, ten for the home-side state machine and nine for the remote-side state machine. The Three-Message-Invalidate CRL protocol also uses every one of these states, except for the new home state HomeBecomingRemote and the new remote state RemoteBecomeHomeReq. These states are described in Tables C.1 and C.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. The only new call in the Floating-Home-Node protocol is CallBecomeHome, which starts the process of making the calling node the home node of a region. Table C.3 describes the six types of call events used in CRL.

Four new messages have been added to the Three-Message-Invalidation. MsgNewHomeInfo is the new home-to-remote protocol message. The other three messages are original-home protocol messages that are either sent from or sent to a region's original-home node, which is the node on which the

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 for the home node)
HomeIipSpecial	An invalidation of remote copies of the region is in progress (to obtain a shared copy for the home node)
HomeInvalid	A single remote node has a valid copy of the region
HomeThreeWayIipSpecial	An invalidation of remote copies of the region is in progress (to obtain a shared copy for a remote node)
HomeBecomingRemote	An invalidation of remote copies of the region is in progress (to change the home node to some other node)

Table C.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
RemoteBecomeHomeReq	This node has sent a request to become the home node and is waiting for MsgInvalidateAcks

Table C.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`)
CallBecomeHome	Become the home node for a region (corresponds to `rgn_become_home`)

Table C.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
MsgNewHomeInfo	Update the home node information for this region on the destination node

Table C.4: CRL home-to-remote protocol messages.

region was originally created. The messages are MsgBecomeHomeReq, MsgBecomeHomeDone, and MsgNegativeAck.

Table C.4 describes the types of protocol messages sent from home nodes to remote nodes (six types of messages); Table C.5 describes those sent from remote nodes to home nodes (six types of messages). Table C.6 describes those sent from remote nodes to requesting nodes (two types of messages). Table C.7 describes those sent from or to a region's original home node (three types of messages).

C.2 Home-Side State Machine

Figures C-1 through C-10 show the state transition diagrams for the ten home-side protocol states. In each figure, solid arrows indicate state transitions taken in response to protocol events; dashed

Message	Description
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 C.5: CRL remote-to-home 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)

Table C.6: CRL remote-to-requesting protocol messages.

Message	Description
MsgBecomeHomeReq	Message sent to the original home requesting to become the home node for a particular region
MsgBecomeHomeDone	Message sent by new home node to the original home node and the previous home node indicating that the become home operation is complete
MsgNegativeAck	Message sent from the original home node in response to a MsgBecomeHome-Req indicating that a become home operation is in progress and that the request cannot be satisfied at this point

Table C.7: CRL original-home protocol messages.

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. 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). Note that the HomeThreeWayIipSpecial and HomeBecomingRemote states do not appear in every state transition diagram. In order to make the diagrams less cluttered, they have been added only to the figures in which they are used.

Figures C-1 through C-10 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 C.8 through C.19. These tables provide pseudocode for the actions taken in response to different protocol events for each of the ten home-side states. Any events that are not listed for a particular state cause a protocol error if they occur (as in Figures C-1 through C-10).

Each of Tables C.8 through C.19 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 C.8 through C.19. 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).

requesting structure list: In the Three-Message-Invalidate protocol, the home-side metadata for a region contains a list of previous requesters for that region. The list is used to redirect flush messages that were sent by remote nodes before those nodes received an invalidate message. Operations supported on the requesting structure lists include insertion of a new structure (insert requesting structure), looking up a structure within the list (get requesting structure for flushed version) and deleting the whole list (delete requesting structure list).

As mentioned earlier, the Floating-Home-Node protocol was built using the Three-Message-Invalidation protocol as the base. Similarly, the state diagrams and pseudocode from the Three-Message-Invalidation protocol were used as a base for this appendix, and changes were marked as follows:

- Additions to the state diagram are bolded. This includes both the event names, and the arrows.

- Additions to the pseudocode are boxed.

- Deletions are crossed out.

A couple significant changes from the Three-Message-Invalidation should be noted. First, there are now transitions from home states to remote states. These transitions are followed on completion of a become home request. Second, home nodes can now receive stale invalidate messages. This may occur if an invalidate was sent when a node was a remote node, but it was received after it had become a home node. In these cases, the invalidate is just ignored.

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

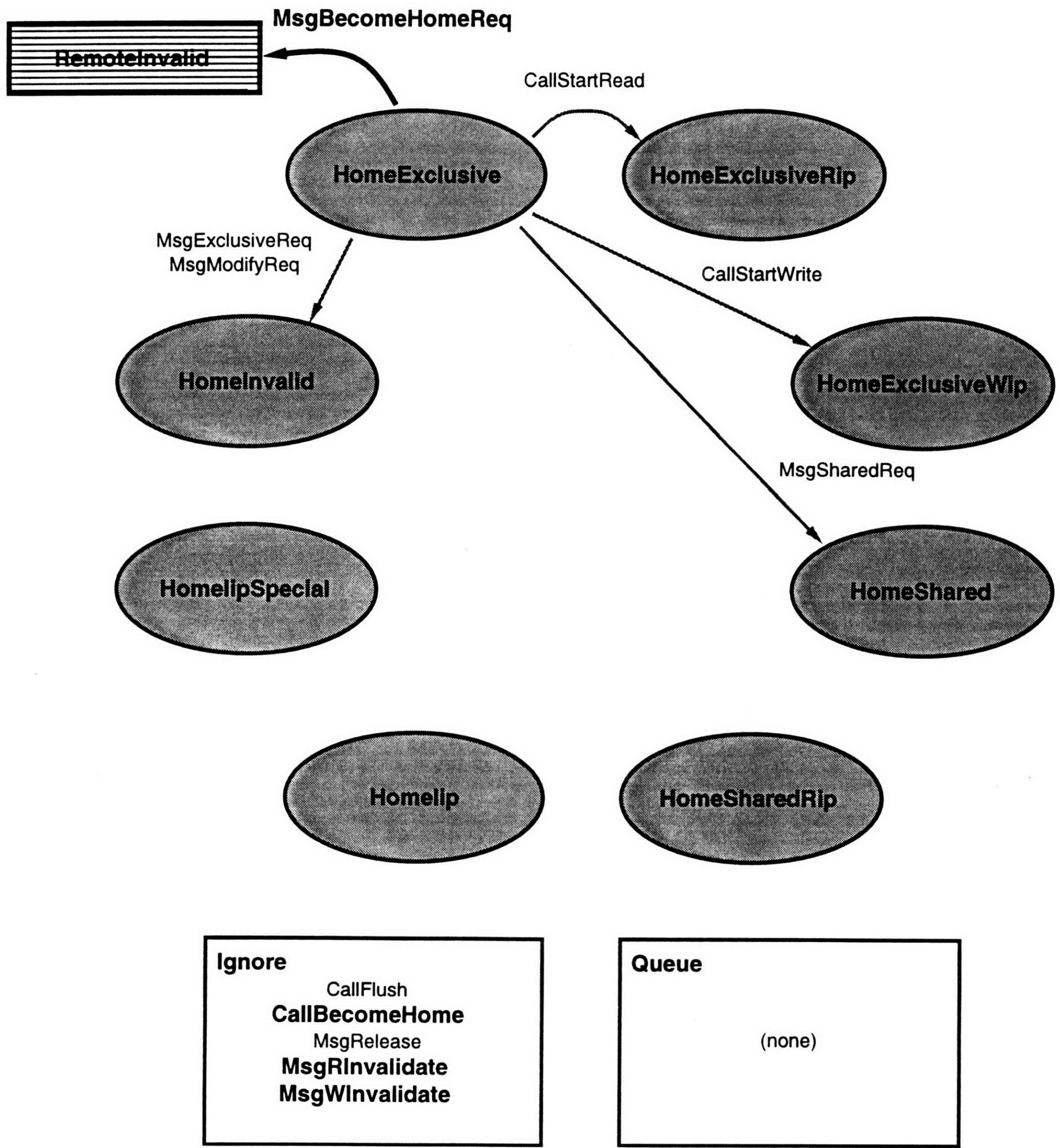

Figure C-1: HomeExclusive: state transition diagram.

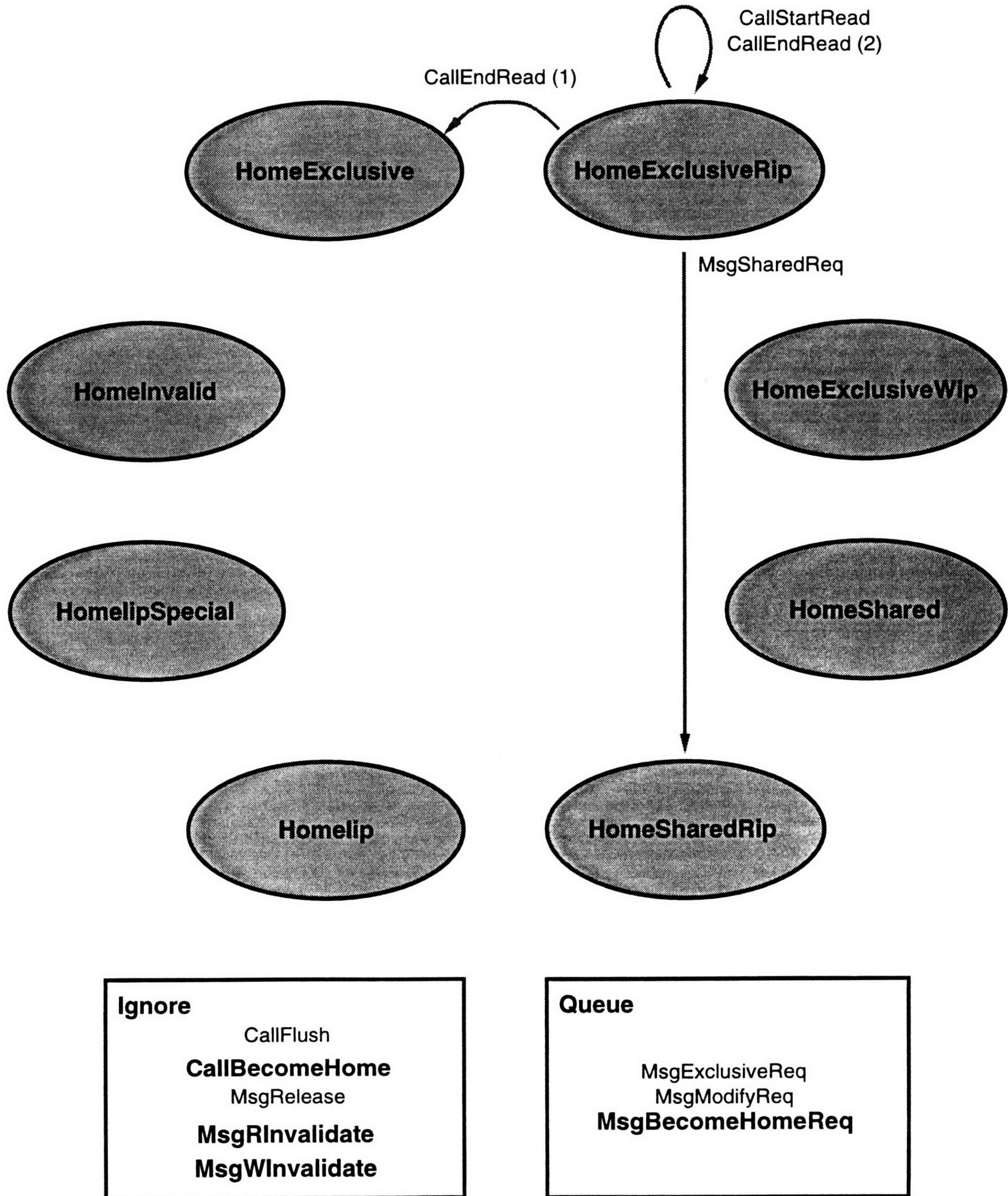

Figure C-2: HomeExclusiveRip: state transition diagram.

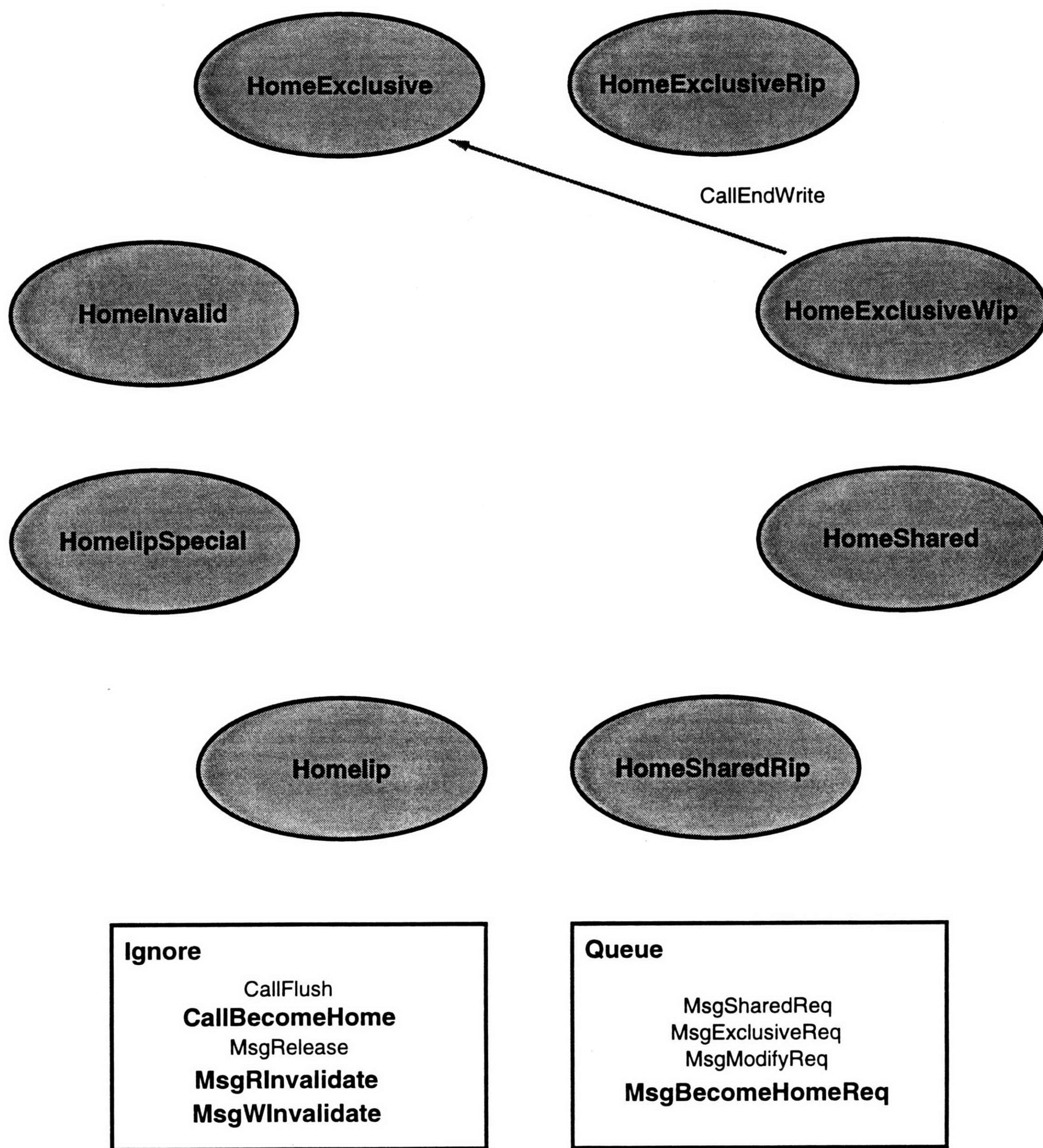

Figure C-3: HomeExclusiveWip: state transition diagram.

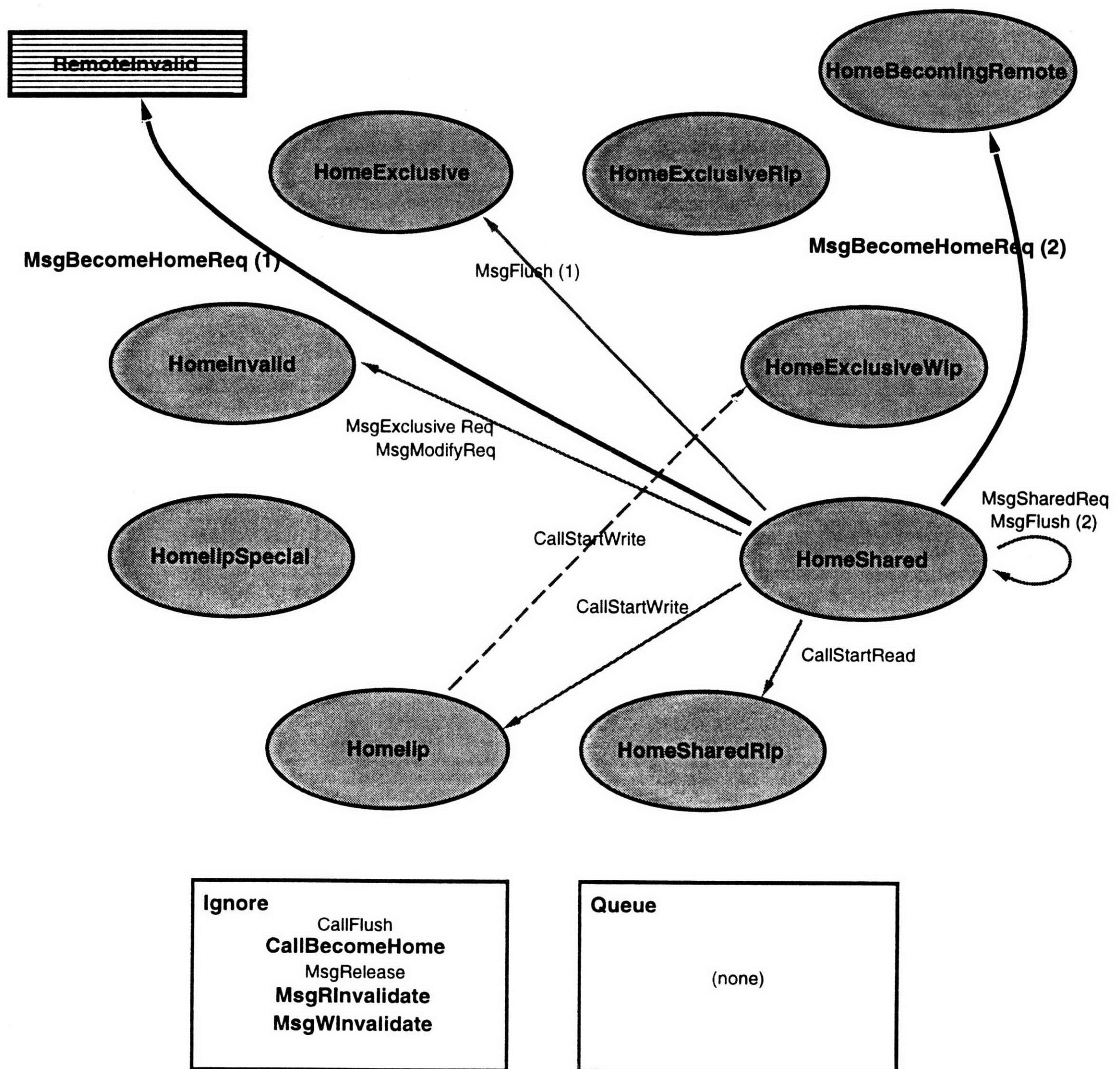

Figure C-4: HomeShared: state transition diagram.

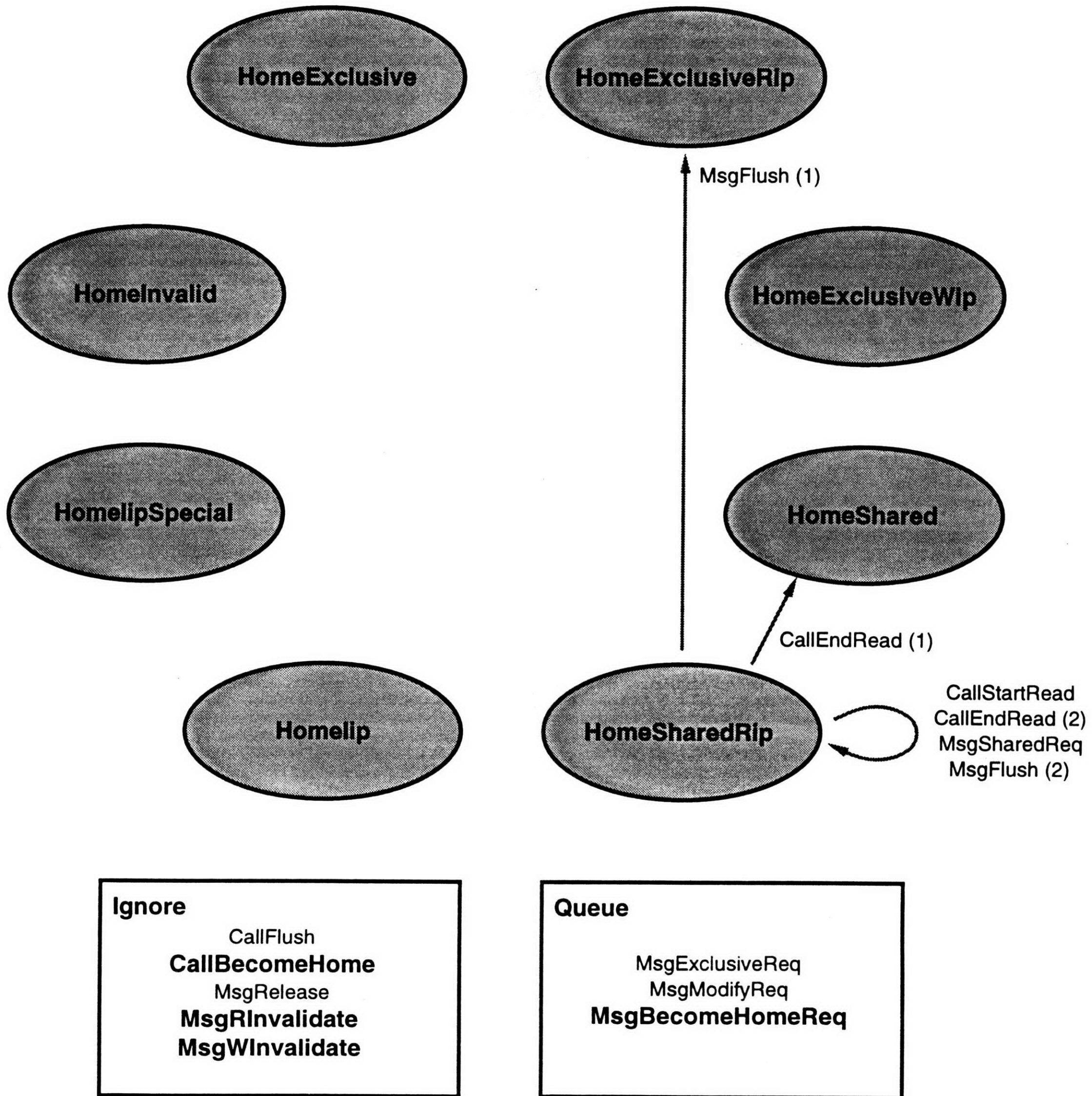

Figure C-5: HomeSharedRip: state transition diagram.

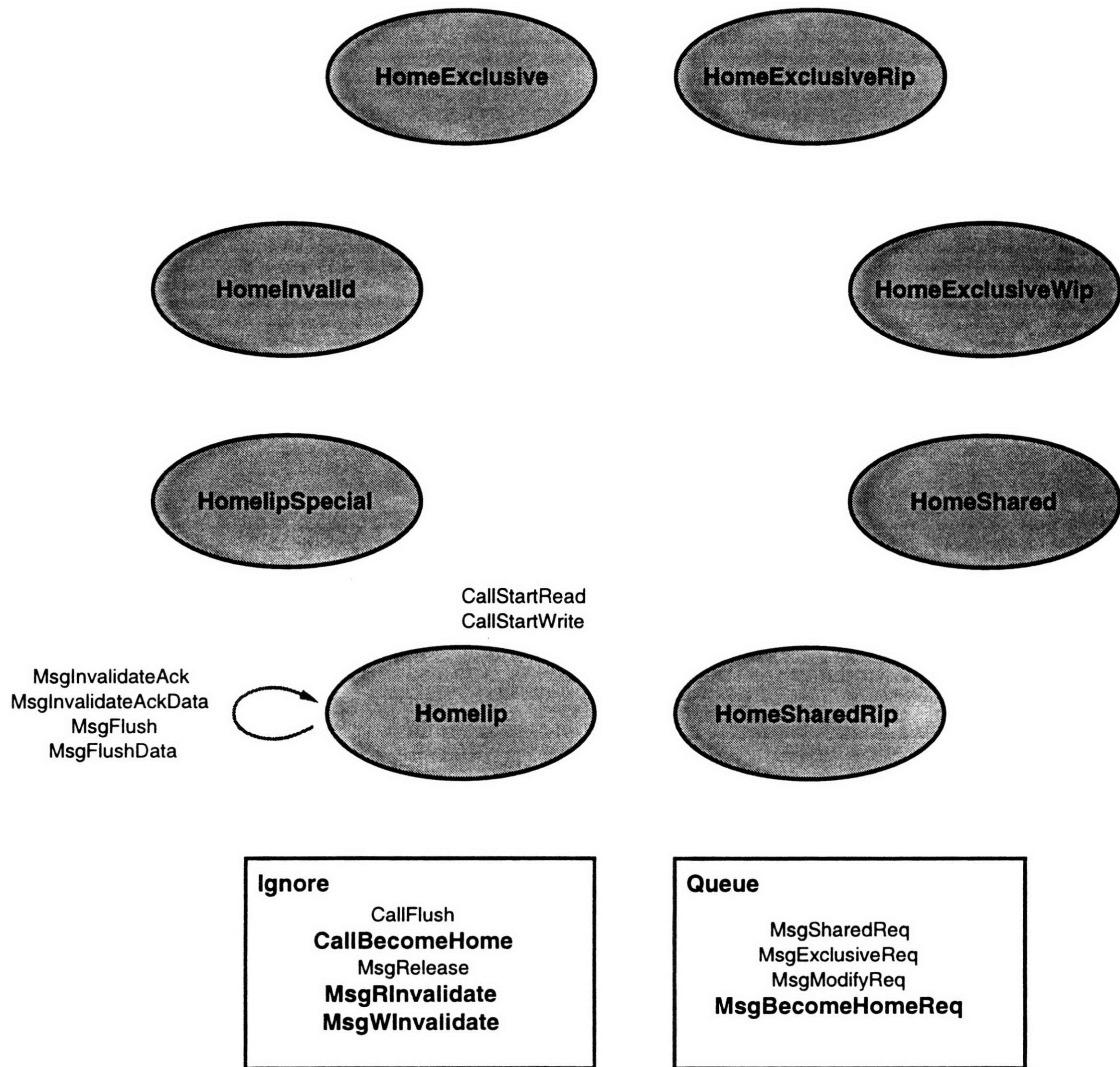

Figure C-6: HomeIip: state transition diagram.

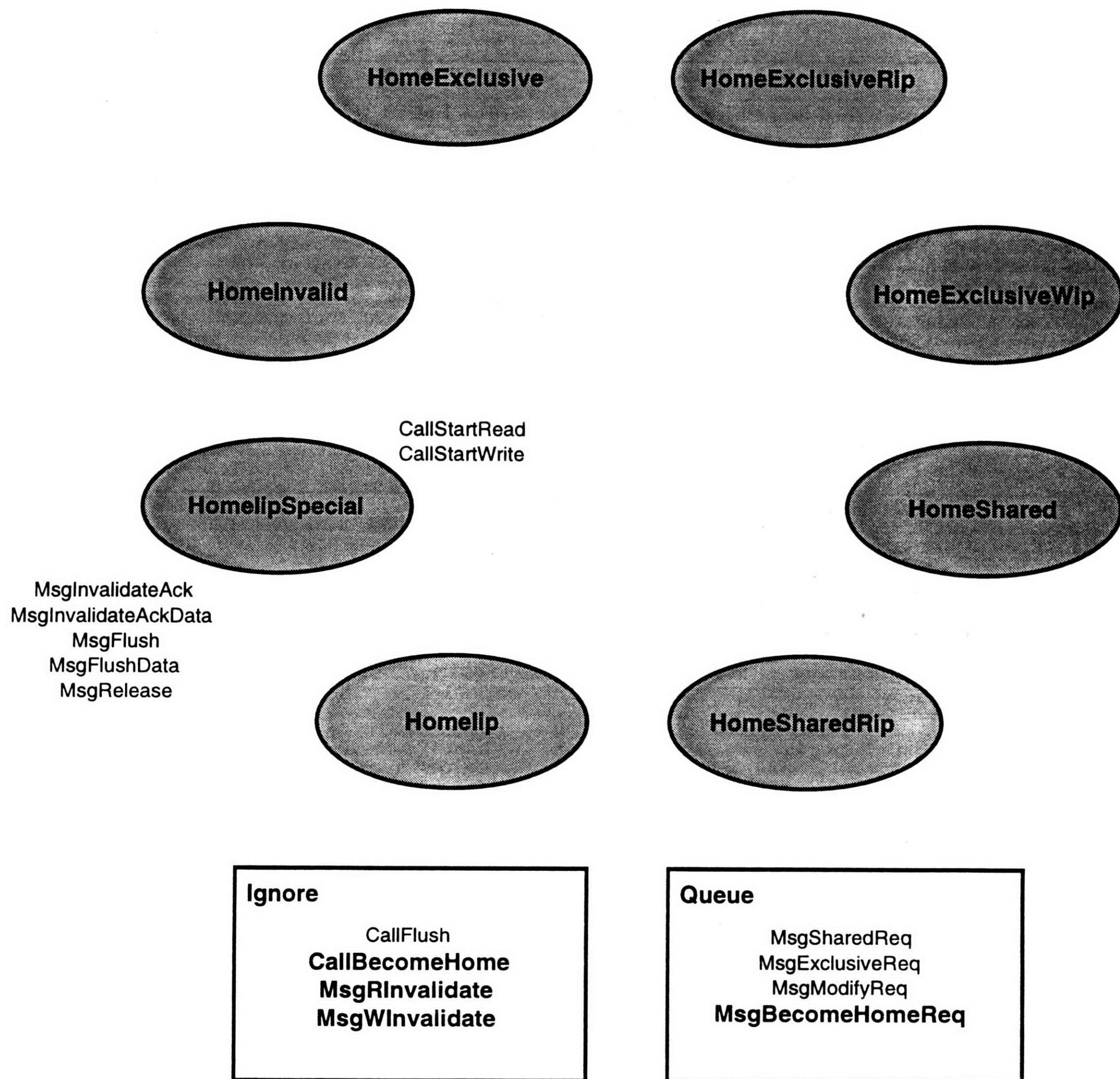

Figure C-7: HomeIipSpecial: state transition diagram.

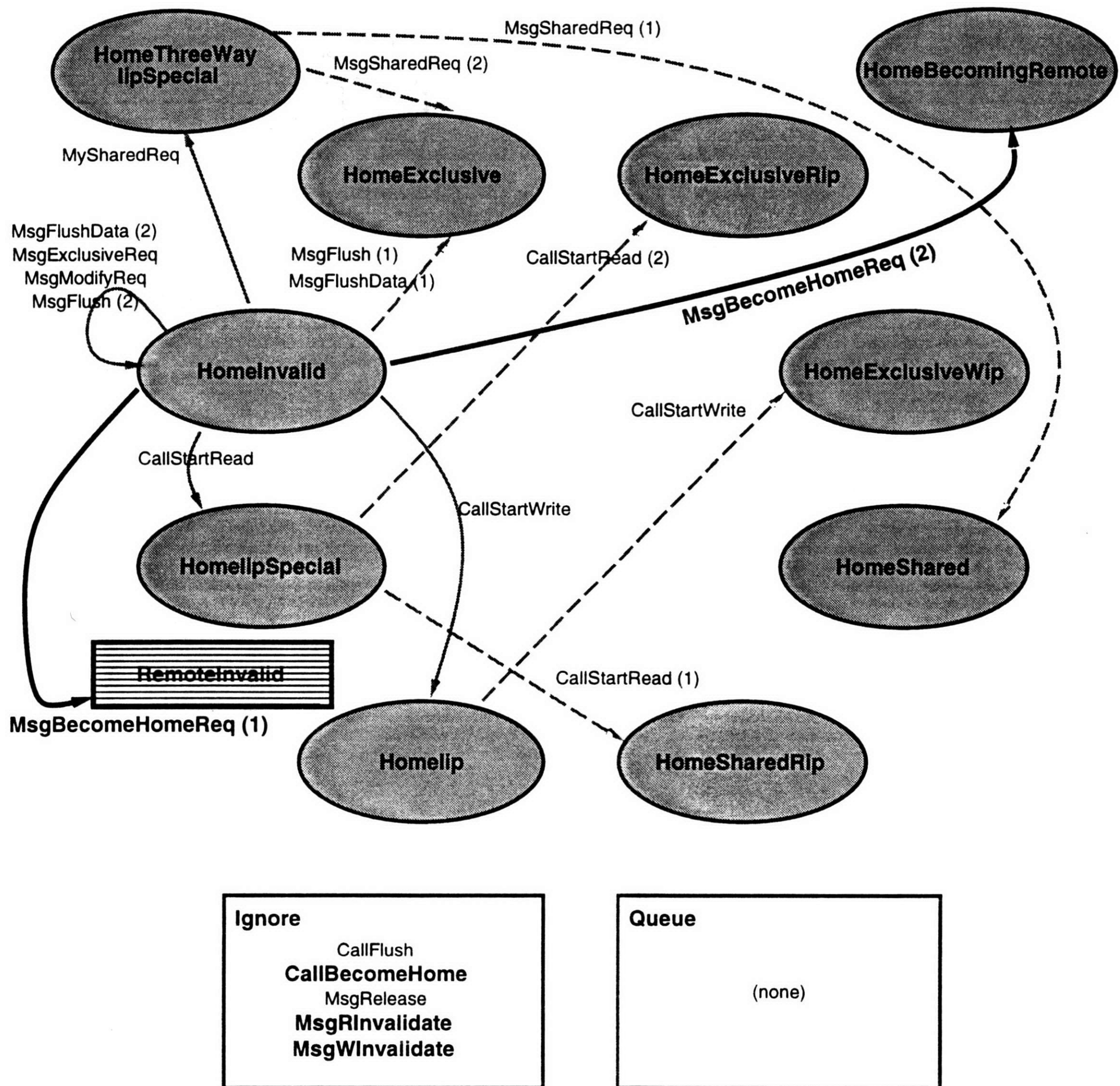

Figure C-8: HomeInvalid: state transition diagram.

Figure C-9: HomeThreeWayIipSpecial: state transition diagram.

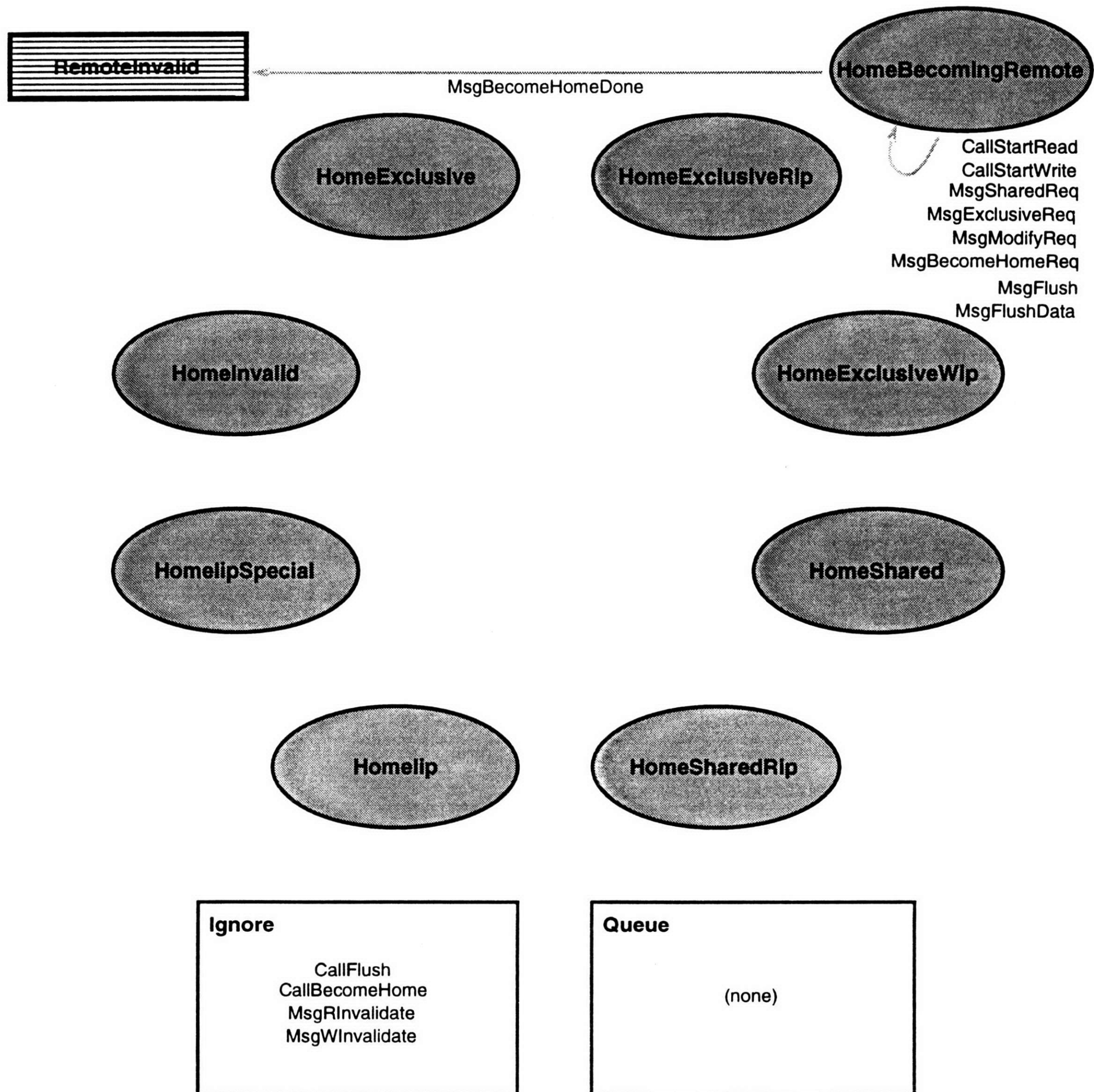

Figure C-10: HomeBecomingRemote: state transition diagram.

State	Event	Actions
HomeExclusive	CallStartRead	*read_cnt* = 1 *state* = HomeExclusiveRip
	CallStartWrite	*state* = HomeExclusiveWip
	CallFlush, CallBecomeHome	do nothing
	MsgSharedReq	if(request was forwarded) send MsgNewHomeInfo to requester send MsgSharedAckData insert pointer *state* = HomeShared
	MsgExclusiveReq	if(request was forwarded) send MsgNewHomeInfo to requester send MsgExclusiveAckData insert pointer *state* = HomeInvalid
	MsgModifyReq	if(request was forwarded) send MsgNewHomeInfo to requester send MsgModifyAckData insert pointer *state* = HomeInvalid
	MsgBecomeHomeReq	send MsgInvalidateAckData forward queued messages to requesting node setup remote region structure send MsgBecomeHomeDone to original home *state* = RemoteInvalid
	MsgRelease, MsgRInvalidate, MsgWInvalidate	do nothing

Table C.8: 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, CallBecomeHome	do nothing
	MsgSharedReq	if(request was forwarded) send MsgNewHomeInfo to requester send MsgSharedAckData insert pointer *state* = HomeSharedRip
	MsgExclusiveReq, MsgModifyReq, MsgBecomeHomeReq	queue message
	MsgRelease, MsgRInvalidate, MsgWInvalidate	do nothing

Table C.9: HomeExclusiveRip: protocol events and actions.

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

Table C.10: 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, CallBecomeHome	do nothing
	MsgSharedReq	if(request was forwarded) send MsgNewHomeInfo to requester send MsgSharedAckData insert pointer
	MsgExclusiveReq	if(request was forwarded) send MsgNewHomeInfo to requester send MsgWInvalidates to remote copies insert pointer insert requesting structure *state* = HomeInvalid

	MsgModifyReq	if(request was forwarded)
		send MsgNewHomeInfo to requester
		if (requesting node is the only pointer)
		send MsgModifyAck
		insert pointer
		state = HomeInvalid
		else
		send MsgWInvalidates to remote copies
		insert pointer
		insert requesting structure
		state = HomeInvalid

	MsgFlush	delete pointer if (no more pointers) *state* = HomeExclusive retry queued messages
	HomeShared: protocol events and actions continues in Table C.12	

Table C.11: HomeShared: protocol events and actions.

State	Event/Continuation	Actions
HomeShared	HomeShared: protocol events and actions continued from Table C.11	

	MsgBecomeHomeReq	if (requesting node is the only pointer)
		send MsgInvalidateAck
		forward queued messages to requesting node
		setup remote region structure
		send MsgBecomeHomeDone to original home
		state = RemoteInvalid
		else
		send MsgWInvalidates and MsgNewHomeInfos to remote copies
		insert pointer
		insert requesting structure
		forward queued messages to requesting node
		state = HomeBecomingRemote

	MsgRelease, MsgRInvalidate, MsgWInvalidate	do nothing

Table C.12: HomeShared: protocol events and actions (continued).

State	Event	Actions
HomeSharedRip	CallStartRead	*read_cnt* += 1
	CallEndRead	*read_cnt* -= 1 if (*read_cnt* == 0) *state* = HomeShared retry queued messages
	CallFlush, CallBecomeHome	do nothing
	MsgSharedReq	if(request was forwarded) send MsgNewHomeInfo to requester send MsgSharedAckData insert pointer
	MsgFlush	delete pointer if (no more pointers) *state* = HomeExclusiveRip retry queued messages
	MsgExclusiveReq, MsgModifyReq MsgBecomeHomeReq	queue message
	MsgRelease, MsgRInvalidate, MsgWInvalidate	do nothing

Table C.13: HomeSharedRip: protocol events and actions.

State	Event	Actions
Homelip	CallStartRead	wait until *state* != Homelip retry CallStartRead
	CallStartWrite	wait until *state* != Homelip retry CallStartWrite
	CallFlush, CallBecomeHome	do nothing

	MsgInvalidateAck,
	MsgInvalidateAckData
		num_ptrs -= 1
		if (*num_ptrs* == 0)
		  delete requesting structure list
		  invoke *tx_cont*

	MsgFlush,
	MsgFlushData
		if (flush for most recent version)
		  num_ptrs -= 1
		  if (*num_ptrs* == 0)
		    delete requesting structure list
		    invoke *tx_cont*
		else
		  get requesting structure for flushed version
		  if (flushing node was sending data)
		    send MsgInvalidateAckData to requester
		  else
		    send MsgInvalidateAck to requester

	MsgSharedReq, MsgExclusiveReq, MsgModifyReq MsgBecomeHomeReq	queue message
	MsgRelease, MsgRInvalidate, MsgWInvalidate	do nothing

Table C.14: Homelip: protocol events and actions.

State	Event	Actions
HomelipSpecial	CallStartRead	wait until *state* != HomelipSpecial retry CallStartRead
	CallStartWrite	wait until *state* != HomelipSpecial retry CallStartWrite
	CallFlush, CallBecomeHome	do nothing
	MsgInvalidateAck, MsgInvalidateAckData	delete requesting structure list invoke *tx_cont* with an arg of 0

	MsgFlush,
	MsgFlushData
		if (flush sent after a MsgRelease)
		rcvd_flush |= OWNERFLUSH
		else if (flush sent before a MsgRelease)
		delete requesting structure list
		invoke *tx_cont* with an arg of 0
		else
		get requesting structure for flushed version
		if (flushing node was sending data)
		send MsgInvalidateAckData to requester
		else
		send MsgInvalidateAck to requester

	MsgRelease	if (MsgRelease is for most recent version) delete requesting structure list invoke *tx_cont* with an arg of 1
	MsgSharedReq, MsgExclusiveReq, MsgModifyReq MsgBecomeHomeReq	queue message
	MsgRInvalidate, MsgWInvalidate	do nothing

Table C.15: HomelipSpecial: 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, CallBecomeHome	do nothing
	MsgSharedReq	if(request was forwarded) send MsgNewHomeInfo to requester send MsgRInvalidate to remote copy *tx_cont* = cont:MsgSharedReq *state* = HomeThreeWayIipSpecial

	cont:MsgSharedReq	if (*num_ptrs* > 0)
		  state = HomeShared
		else
		  state = HomeExclusive

	MsgExclusiveReq	if(request was forwarded) send MsgNewHomeInfo to requester send MsgWInvalidate to remote copy insert pointer insert requesting structure
	HomeInvalid: protocol events and actions continues in Table C.17	

Table C.16: HomeInvalid: protocol events and actions.

State	Event/Continuation	Actions
HomeInvalid	HomeInvalid: protocol events and actions continued from Table C.16	
	MsgModifyReq	if(request was forwarded) send MsgNewHomeInfo to requester send MsgWInvalidate to remote copy insert pointer insert requesting structure

	MsgFlush,
	MsgFlushData
		if (most recent requester flushed)
		delete requesting structure list
		delete pointer
		state = HomeExclusive
		else
		get requesting structure for flushed version
		if (flushing node was sending data)
		send MsgInvalidateAckData to requester
		else
		send MsgInvalidateAck to requester

	MsgBecomeHomeReq	if (requesting node is the only pointer)
		send MsgInvalidateAck
		forward queued messages to requesting node
		setup remote region structure
		send MsgBecomeHomeDone to original home
		state = RemoteInvalid
		else
		send MsgWInvalidates and MsgNewHomeInfos to remote copies
		insert requesting structure
		forward queued messages to requesting node
		state = HomeBecomingRemote

	MsgRelease, MsgRInvalidate, MsgWInvalidate	do nothing

Table C.17: HomeInvalid: protocol events and actions (continued).

State	Event	Actions
HomeThreeWayIipSpecial	CallStartRead	wait until *state* != HomeThreeWayIipSpecial retry CallStartRead
	CallStartWrite	wait until *state* != HomeThreeWayIipSpecial retry CallStartWrite
	CallFlush, CallBecomeHome	do nothing
	MsgInvalidateAck, MsgInvalidateAckData	if ((*rcvd_flush* & REQUESTERFLUSH) != 0) insert pointer delete requesting structure list invoke *tx_cont* with an arg of 0

	MsgFlush,
	MsgFlushData
		if (flush sent by previous owner after a MsgRelease)
		  rcvd_flush |= OWNERFLUSH
		else if (flush sent by requester)
		  rcvd_flush |= REQUESTERFLUSH
		else if (flush sent before a MsgRelease)
		  insert pointer
		  send MsgSharedAckData to requester
		  delete requesting structure list
		  invoke *tx_cont* with an arg of 0
		else
		  get requesting structure for flushed version
		  if (flushing node was sending data)
		    send MsgInvalidateAckData to requester
		  else
		    send MsgInvalidateAck to requester

	MsgRelease	if (MsgRelease is for most recent version)
		  if ((*rcvd_flush* & OWNERFLUSH) != 0)
		    delete pointer
		  if ((*rcvd_flush* & REQUESTERFLUSH) == 0)
		    insert pointer
		  delete requesting structure list
		  invoke *tx_cont* with an arg of 1

	MsgSharedReq, MsgExclusiveReq, MsgModifyReq MsgBecomeHomeReq	queue message
	MsgRInvalidate, MsgWInvalidate	do nothing

Table C.18: HomeThreeWayIipSpecial: protocol events and actions.

State	Event	Actions
HomeBecomingRemote	CallStartRead	wait until *state* != HomeBecomingRemote retry CallStartRead
	CallStartWrite	wait until *state* != HomeBecomingRemote retry CallStartWrite
	CallFlush, CallBecomeHome	do nothing
	MsgSharedReq, MsgExclusiveReq, MsgModifyReq	forward message to new home

	MsgFlush,
	MsgFlushData
		get requesting structure for flushed version
		if (flushing node was sending data)
		  send MsgInvalidateAckData to requester
		else
		  send MsgInvalidateAck to requester

	MsgBecomeHomeReq	Send MsgNegativeAck to requester
	MsgBecomeHomeDone	setup remote region structure *state* = RemoteInvalid
	MsgRInvalidate, MsgWInvalidate	do nothing

Table C.19: HomeBecomingRemote: protocol events and actions.

C.3 Remote-Side State Machine

Figures C-11 through C-19 show the state transition diagrams for the nine remote-side protocol states. These figures are similar to those shown for the home-side state machine (Figures C-1 through C-10), with one minor difference. 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), the **Queue** box is instead labeled **Set rcvd_inv flag**. This is true for every state except for the RemoteBecomeHomeReq state which must queue messages as the node is becoming the new home node for the region.

As was the case in Figures C-1 through C-10 (for the home-side state machine), Figures C-11 through C-19 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 pseudocode) can be found in Tables C.20 through C.28.

Each of Tables C.20 through C.28 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, three other components of the remote-side protocol metadata associated with each region are referenced in Tables C.20 through C.28. 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.

num_invalidate_acks: In the Three-Message-Invalidate protocol, this field is used to count the number of invalidate acknowledgment messages that will be arriving.

As mentioned earlier, the Floating-Home-Node protocol was built using the Three-Message-Invalidation protocol as the base. Similarly, the state diagrams and pseudocode from the Three-Message-Invalidation protocol were used as a base for this appendix, and changes were marked as follows:

- Additions to the state diagram are bolded. This includes both the event names, and the arrows.
- Additions to the pseudocode are boxed.
- Deletions are crossed out.

A couple significant changes from the Three-Message-Invalidation should be noted. First, there are now transitions from remote states to home states. These transitions are followed on completion

of a become home request. Second, remote nodes can now receive request messages. This may occur if the request was sent by a node with stale region home information. In these cases, if the receiving node is the original home node, then the message is forwarded to the current home node. If the receiving node is some other previous home node, then the message is forwarded to the original home node.

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

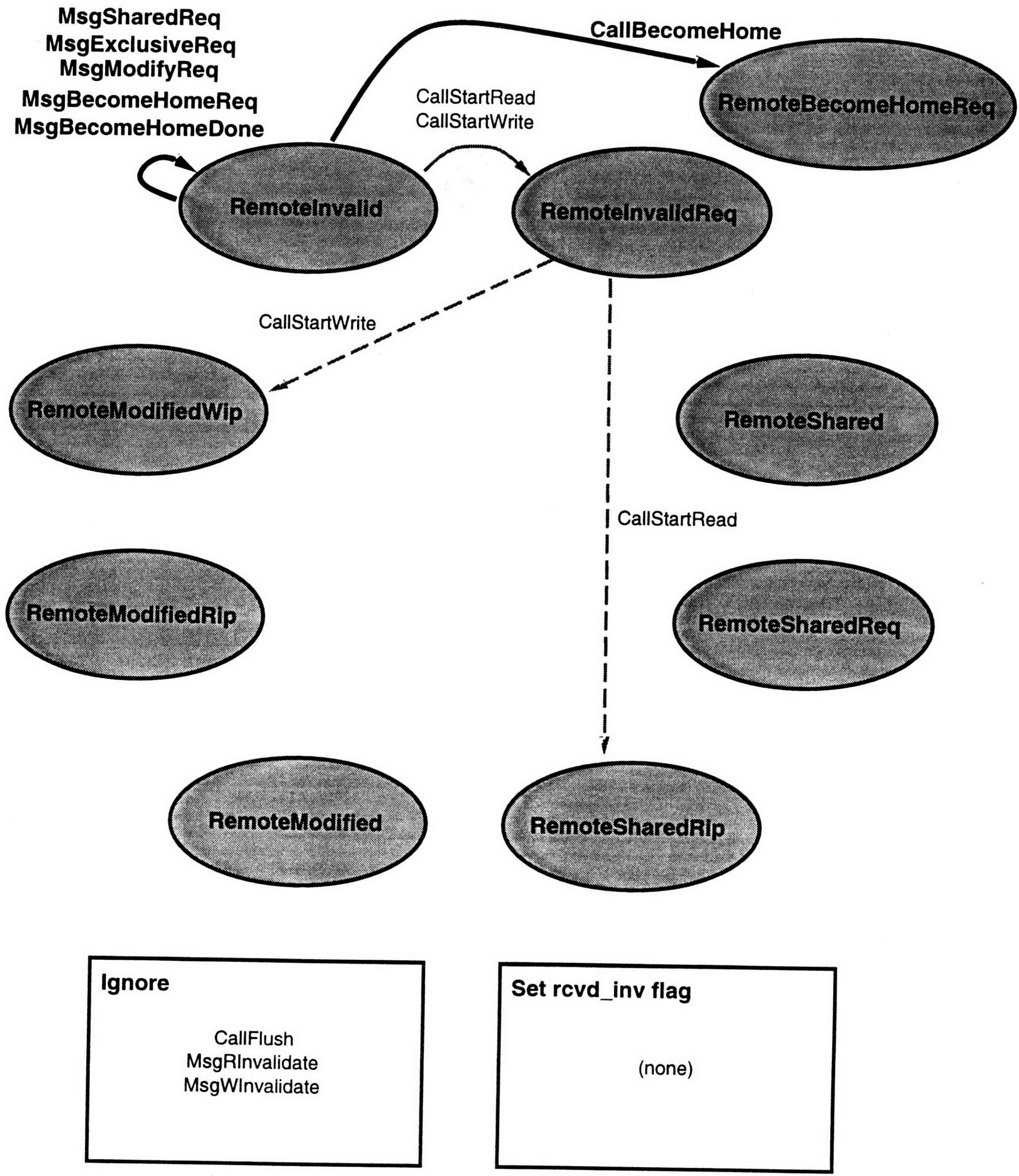

Figure C-11: RemoteInvalid: state transition diagram.

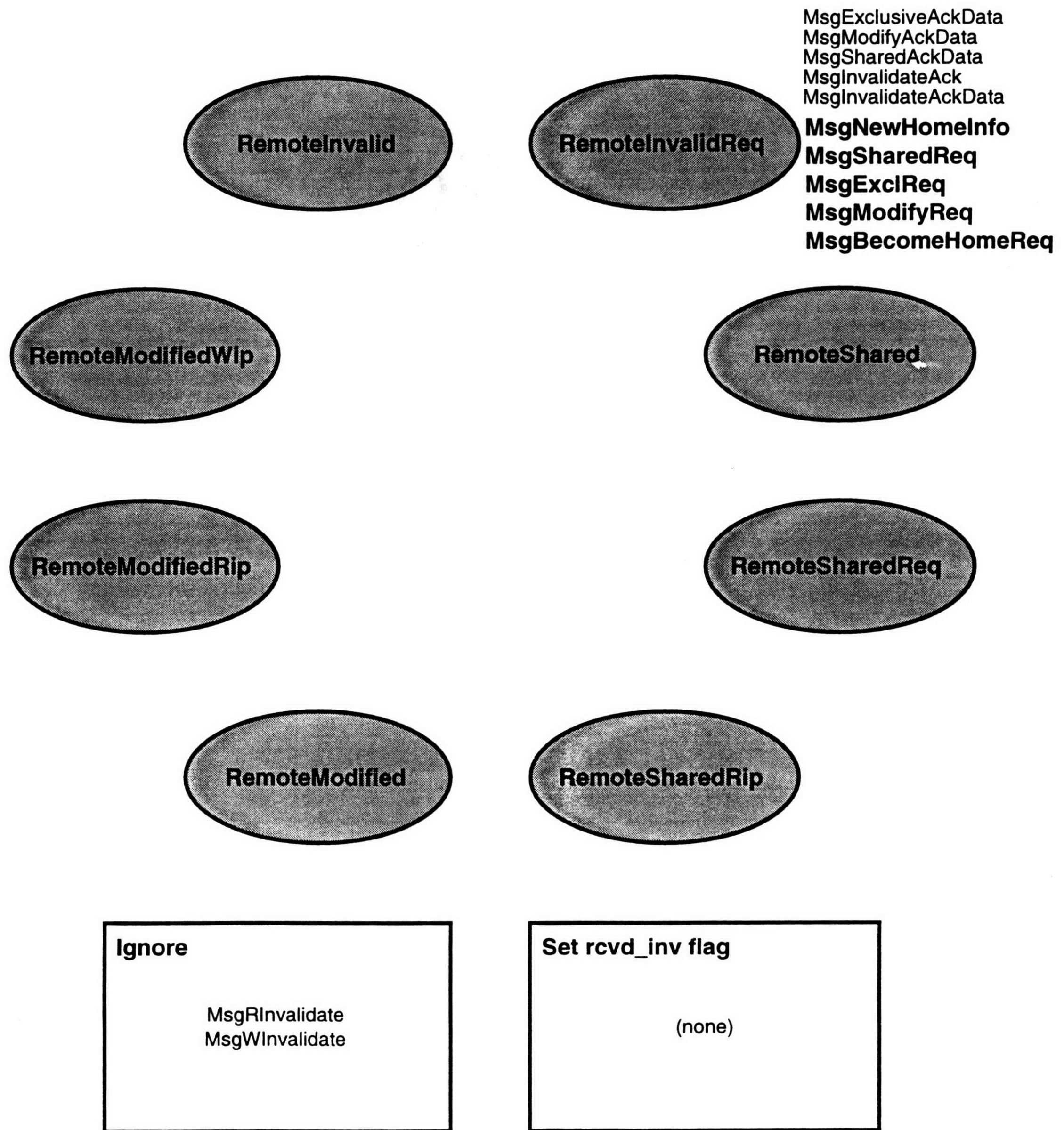

Figure C-12: RemoteInvalidReq: state transition diagram.

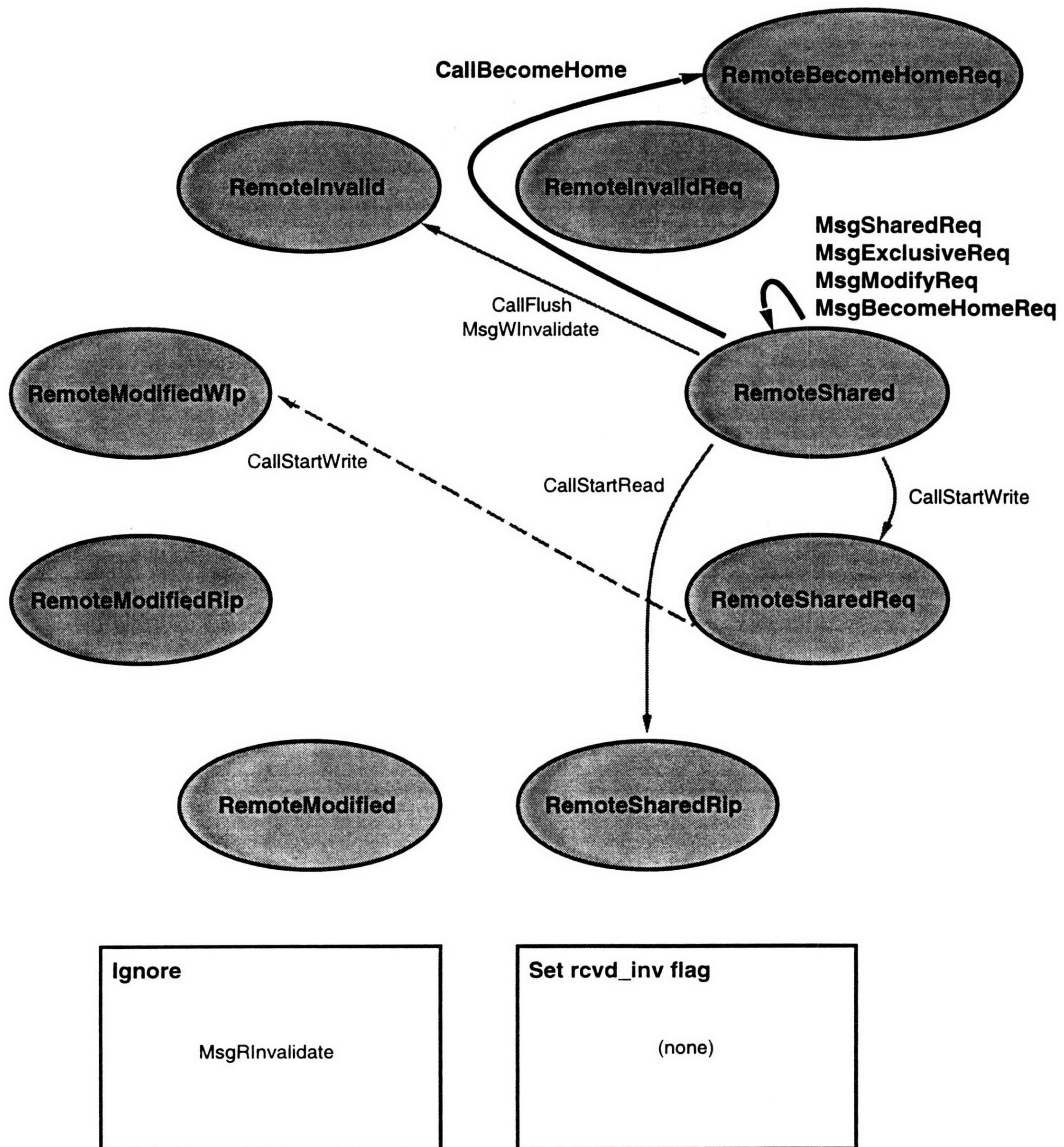

Figure C-13: RemoteShared: state transition diagram.

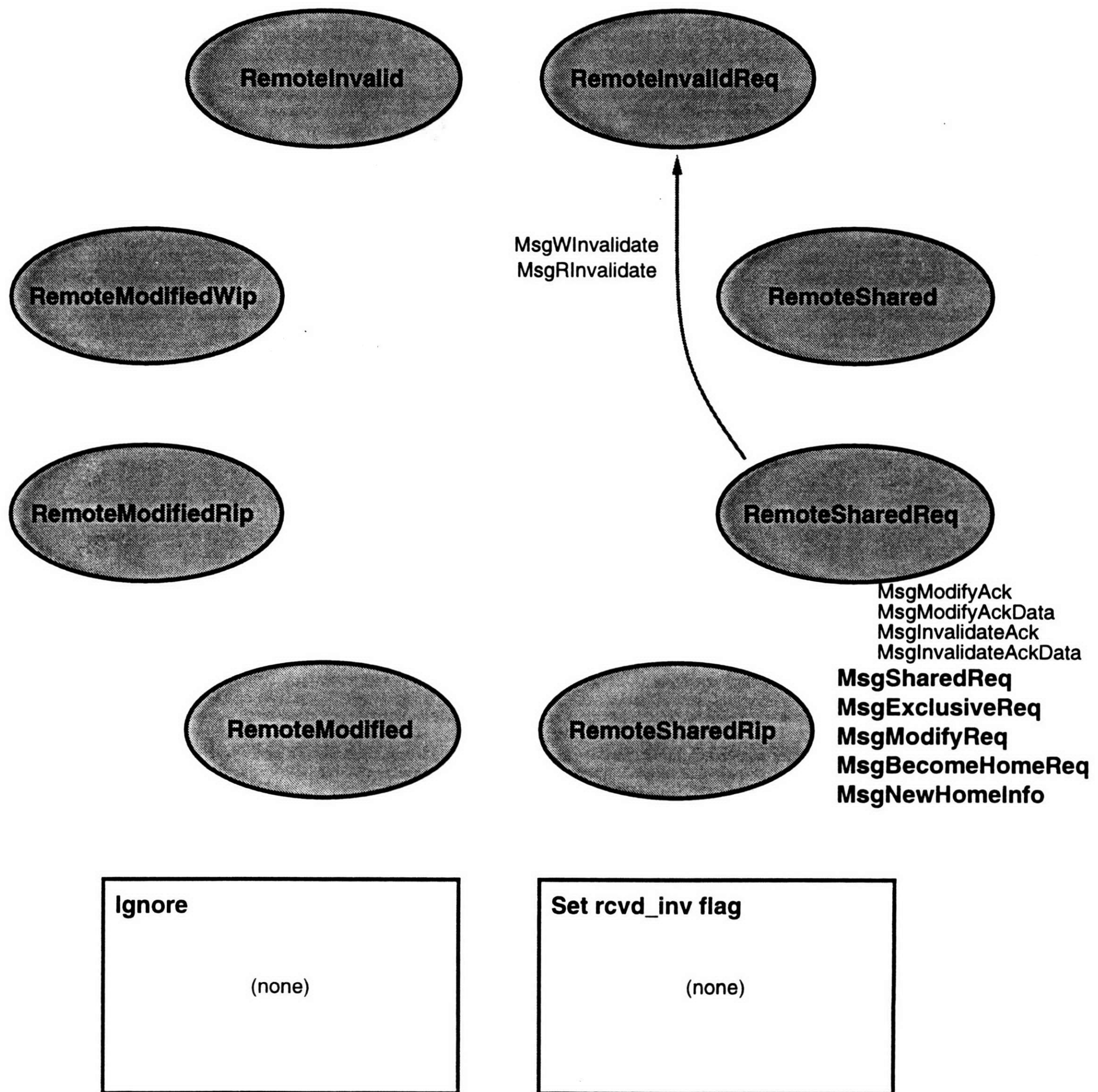

Figure C-14: RemoteSharedReq: state transition diagram.

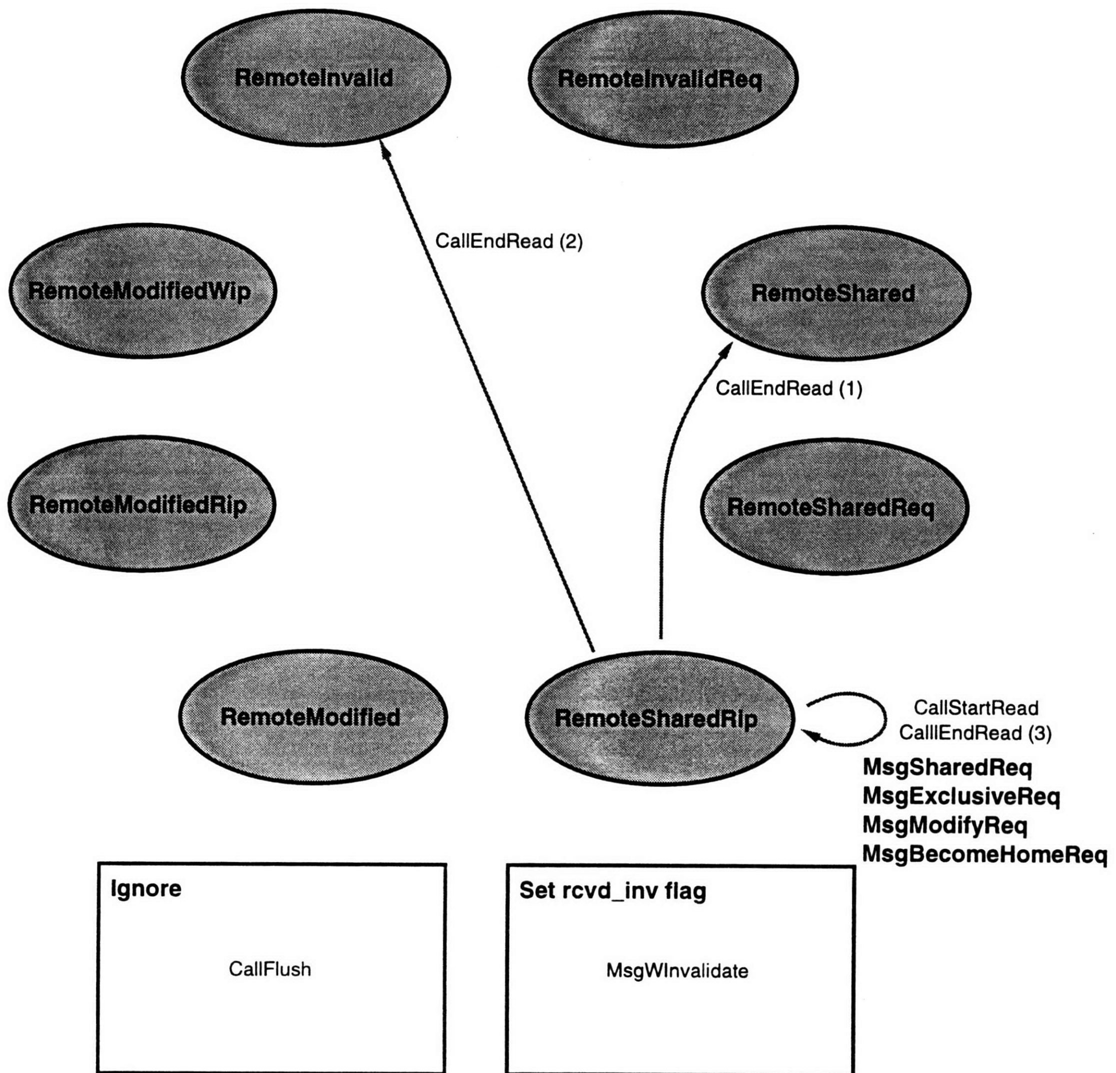

Figure C-15: RemoteSharedRip: state transition diagram.

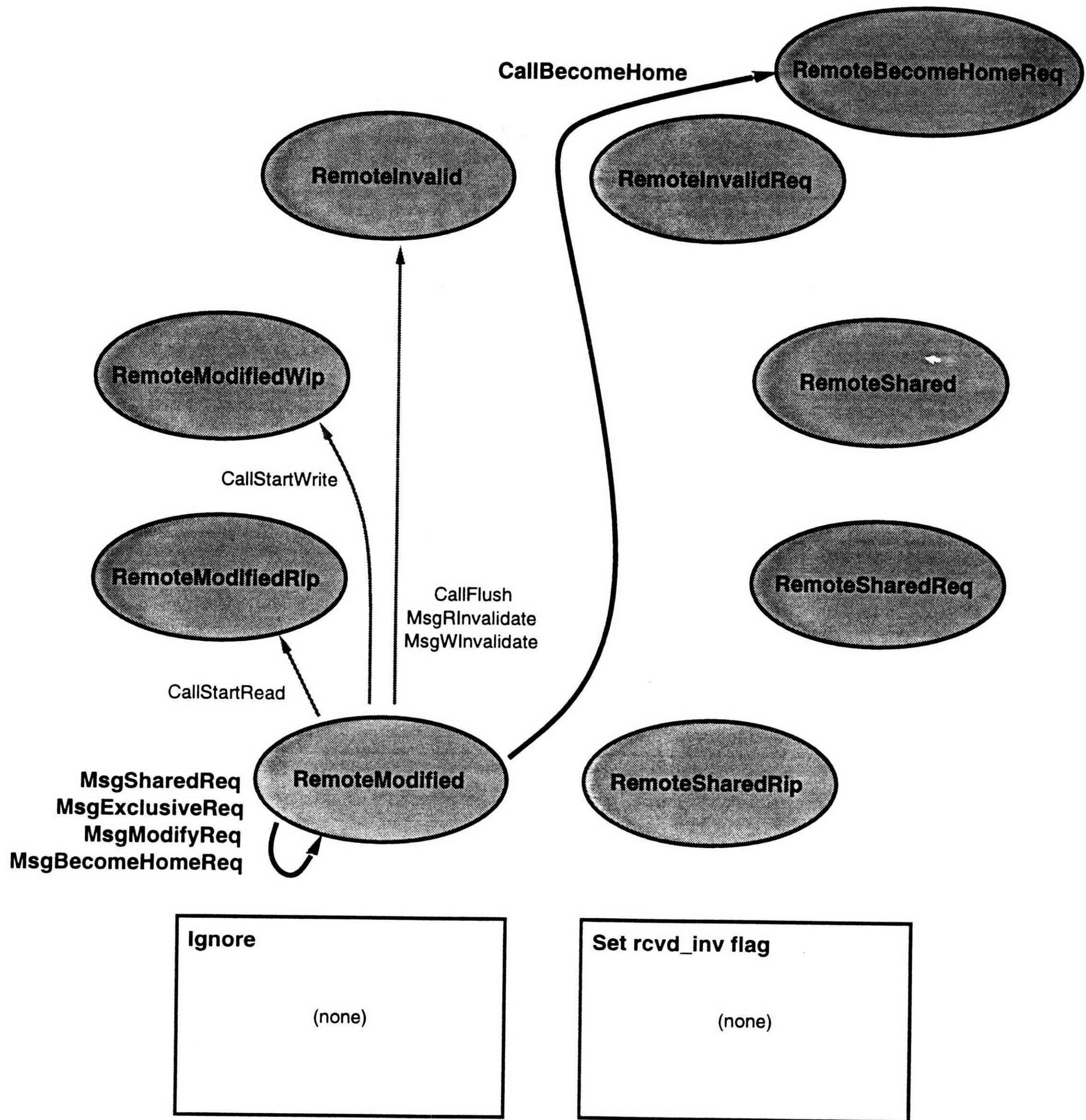

Figure C-16: RemoteModified: state transition diagram.

Figure C-17: RemoteModifiedRip: state transition diagram.

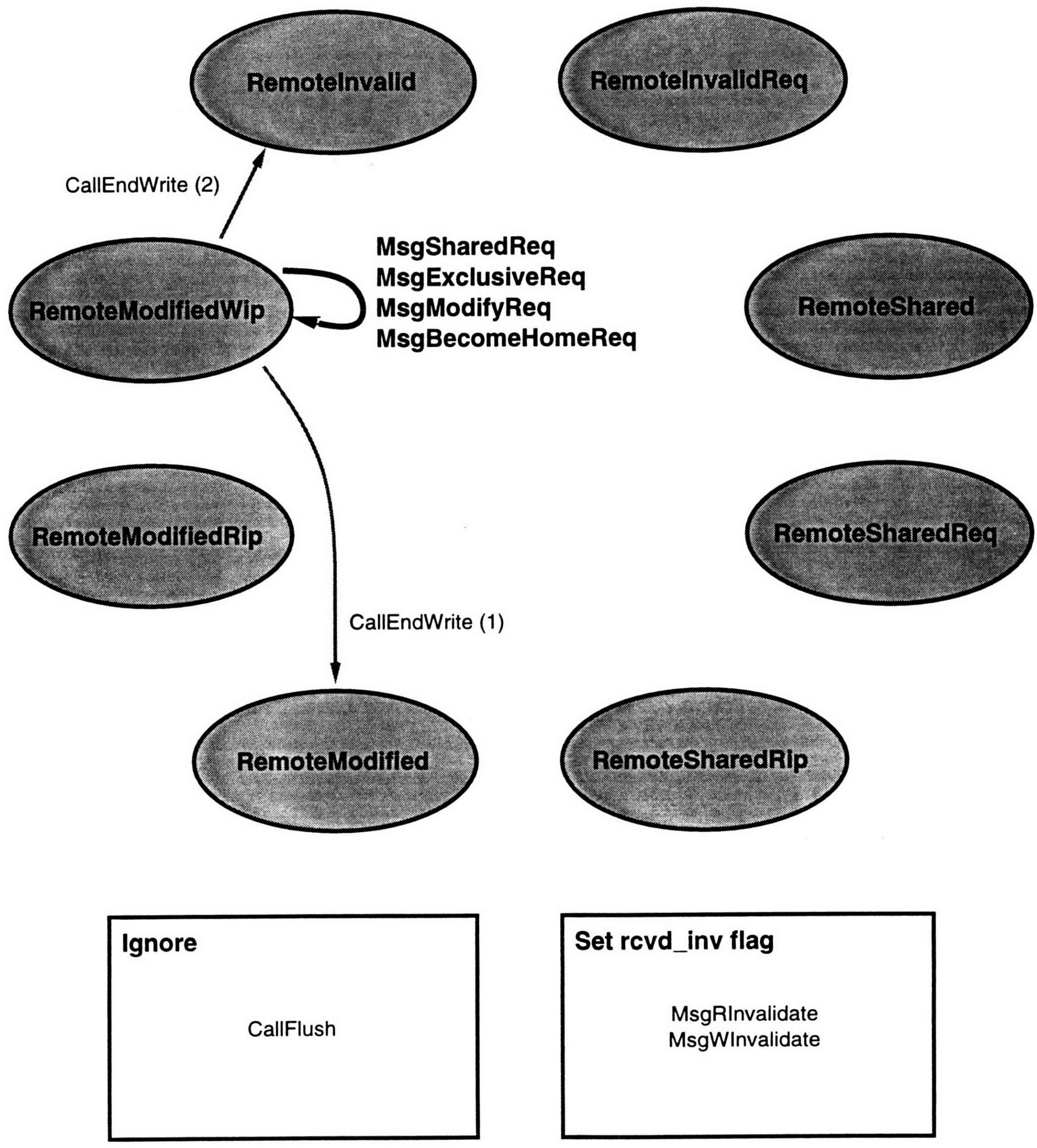

Figure C-18: RemoteModifiedWip: state transition diagram.

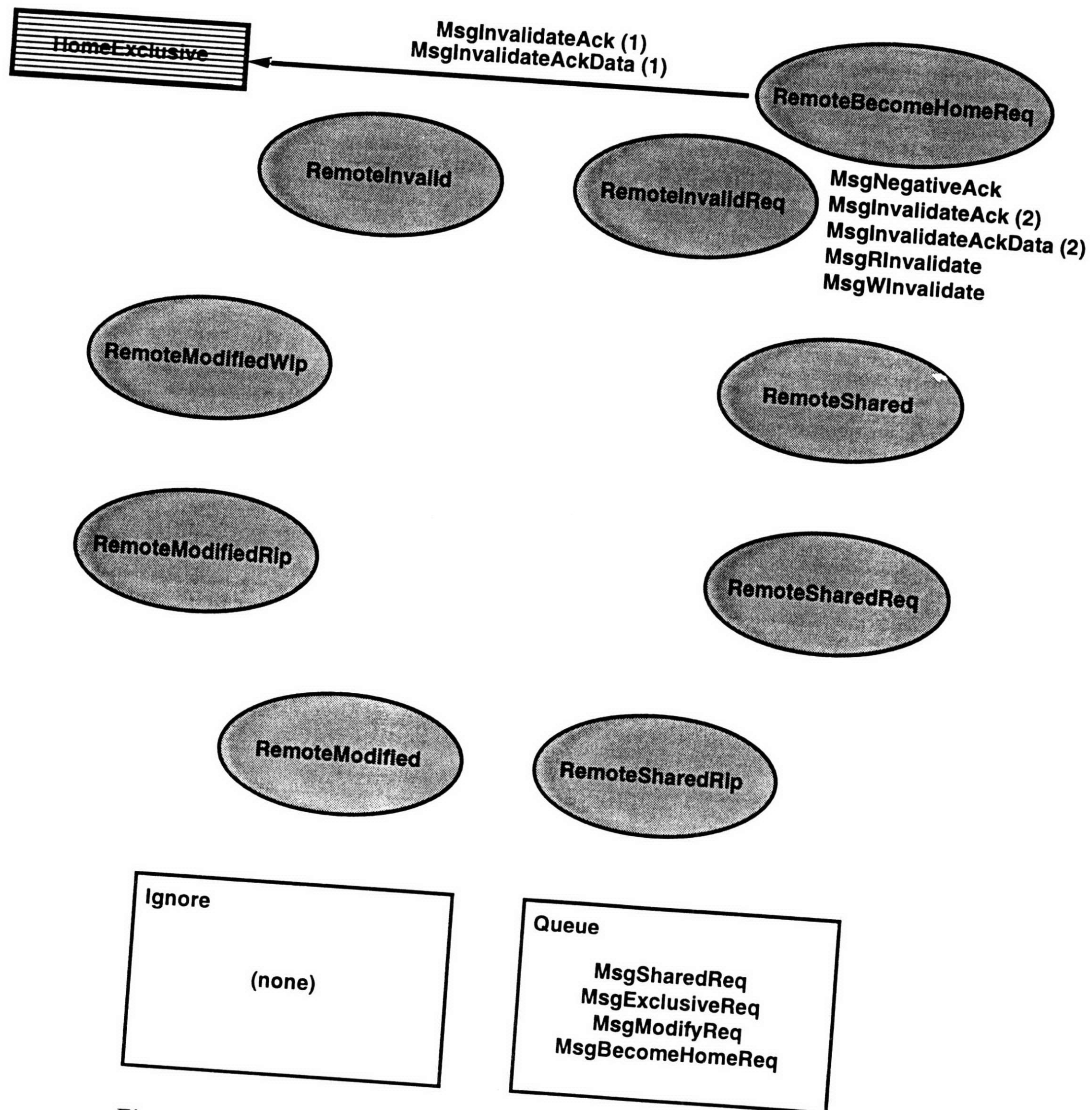

Figure C-19: RemoteBecomeHomeReq: state transition diagram.

State	Event	Actions
RemoteInvalid	CallStartRead	if(this is original home AND a become home request is in progress) poll until become home request is complete send MsgSharedReq to home *tx_cont* = cont:CallStartRead *state* = RemoteInvalidReq poll until *tx_cont* has been invoked
	cont:CallStartRead	*read_cnt* = 1 *state* = RemoteSharedRip
	CallStartWrite	if(this is original home AND a become home request is in progress) poll until become home request is complete send MsgExclusiveReq to home *tx_cont* = cont:CallStartWrite *state* = RemoteInvalidReq poll until *tx_cont* has been invoked
	cont:CallStartWrite	*state* = RemoteModifiedWip
	CallBecomeHome	if(this is original home AND a become home request is in progress) poll until become home request is complete setup home region structure send MsgBecomeHomeReq to original home *state* = RemoteBecomeHomeReq poll until this is the home node
	CallFlush	do nothing

	MsgSharedReq,
	MsgExclusiveReq,
	MsgModifyReq
		if(this is the original home)
		forward message to current home node
		else
		forward message to original home

	MsgBecomeHomeReq	if(become home request in progress)
		send MsgNegativeAck
		else
		forward message to current home node

	MsgBecomeHomeDone	forward message to previous home
	MsgRInvalidate, MsgWInvalidate	do nothing

Table C.20: RemoteInvalid: protocol events and actions.

State	Event	Actions

RemoteInvalidReq	MsgSharedAckData,
	MsgExclusiveAckData,
	MsgModifyAckData
		~~invoke *tx_cont*~~
		if (*num_invalidate_acks* is unset)
		num_invalidate_acks = # of MsgInvalidateAcks expected
		num_invalidate_acks -= 1
		if (*num_invalidate_acks* == 0)
		invoke *tx_cont*

	MsgInvalidateAck,
	MsgInvalidateAckData
		if (*num_invalidate_acks* is unset)
		num_invalidate_acks = # of MsgInvalidateAcks expected
		num_invalidate_acks -= 1
		if (*num_invalidate_acks* == 0)
		invoke *tx_cont*

	MsgSharedReq,
	MsgExclusiveReq,
	MsgModifyReq
		if(this is the original home)
		forward message to current home node
		else
		forward message to original home

	MsgBecomeHomeReq	if(become home request in progress)
		send MsgNegativeAck
		else
		forward message to current home node

	MsgNewHomeInfo	set region home information
		if (*num_invalidate_acks* is unset)
		num_invalidate_acks = # of MsgInvalidateAcks expected
		num_invalidate_acks -= 1
		if (*num_invalidate_acks* == 0)
		invoke *tx_cont*

	MsgRInvalidate, MsgWInvalidate	do nothing

Table C.21: RemoteInvalidReq: protocol events and actions.

State	Event	Actions
RemoteShared	CallStartRead	*read_cnt* = 1 *state* = RemoteSharedRip
	CallStartWrite	send MsgModifyReq to home *tx_cont* = cont:CallStartWrite *state* = RemoteSharedReq poll until *tx_cont* has been invoked
	cont:CallStartWrite	*state* = RemoteModifiedWip
	CallFlush	send MsgFlush to home *state* = RemoteInvalid
	CallBecomeHome	if(this is original home AND a become home request is in progress) poll until become home request is complete setup home region structure send MsgBecomeHomeReq to original home *state* = RemoteBecomeHomeReq poll until this is the home node

	MsgSharedReq,
	MsgExclusiveReq,
	MsgModifyReq
		if(this is the original home)
		forward message to current home node
		else
		forward message to original home

	MsgBecomeHomeReq	if(become home request in progress)
		send MsgNegativeAck
		else
		forward message to current home node

	MsgRInvalidate	do nothing

	MsgWInvalidate	if (need to send data)
		send MsgInvalidateAckData to requester
		else
		send MsgInvalidateAck to requester
		state = RemoteInvalid

Table C.22: RemoteShared: protocol events and actions.

State	Event	Actions

RemoteSharedReq	MsgInvalidateAck,
	MsgInvalidateAckData
		if (*num_invalidate_acks* is unset)
		  num_invalidate_acks = # of MsgInvalidateAcks
		    expected
		num_invalidate_acks -= 1
		if (*num_invalidate_acks* == 0)
		  invoke *tx_cont*

	MsgWInvalidate,
	MsgRInvalidate
		if (need to send data)
		  send MsgInvalidateAckData to requester
		else
		  send MsgInvalidateAck to requester
		state = RemoteInvalidReq

	MsgModifyAck, MsgModifyAckData	invoke *tx_cont*

	MsgSharedReq,
	MsgExclusiveReq,
	MsgModifyReq
		if(this is the original home)
		  forward message to current home node
		else
		  forward message to original home

	MsgBecomeHomeReq	if(become home request in progress)
		  send MsgNegativeAck
		else
		  forward message to current home node

	MsgNewHomeInfo	set region home information
		if (*num_invalidate_acks* is unset)
		  num_invalidate_acks = # of MsgInvalidateAcks
		    expected
		num_invalidate_acks -= 1
		if (*num_invalidate_acks* == 0)
		  invoke *tx_cont*

Table C.23: **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
		    if (need to send data)
		      send MsgInvalidateAckData to requester
		    else
		      send MsgInvalidateAck to requester
		    rcvd_inv = 0
		    state = RemoteInvalid

	CallFlush	do nothing
	MsgWInvalidate	*rcvd_inv* = WInvalidate

	MsgSharedReq,
	MsgExclusiveReq,
	MsgModifyReq
		if(this is the original home)
		  forward message to current home node
		else
		  forward message to original home

	MsgBecomeHomeReq	if(become home request in progress)
		  send MsgNegativeAck
		else
		  forward message to current home node

Table C.24: **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
	CallBecomeHome	if(this is original home AND a become home request is in progress) poll until become home request is complete setup home region structure send MsgBecomeHomeReq to original home *state* = RemoteBecomeHomeReq poll until this is the home node
	MsgRInvalidate	send MsgInvalidateAckData to requester if (requester != home) send MsgInvalidateAckData to home *state* = RemoteInvalid
	MsgWInvalidate	send MsgInvalidateAckData to requester *state* = RemoteInvalid

	MsgSharedReq,
	MsgExclusiveReq,
	MsgModifyReq
		if(this is the original home)
		forward message to current home node
		else
		forward message to original home

	MsgBecomeHomeReq	if(become home request in progress)
		send MsgNegativeAck
		else
		forward message to current home node

Table C.25: **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 requester
		if ((*rcvd_inv* == RInvalidate) AND (requester != home))
		send MsgInvalidateAckData to home
		rcvd_inv = 0
		state = RemoteInvalid

	CallFlush	do nothing
	MsgRInvalidate	send MsgRelease to home if (requester != home) send MsgInvalidateAckData to requester *state* = RemoteSharedRip
	MsgWInvalidate	*rcvd_inv* = WInvalidate

	MsgSharedReq,
	MsgExclusiveReq,
	MsgModifyReq
		if(this is the original home)
		forward message to current home node
		else
		forward message to original home

	MsgBecomeHomeReq	if(become home request in progress)
		send MsgNegativeAck
		else
		forward message to current home node

Table C.26: **RemoteModifiedRip**: protocol events and actions.

State	Event	Actions

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

	CallFlush	do nothing
	MsgRInvalidate	*rcvd_inv* = RInvalidate
	MsgWInvalidate	*rcvd_inv* = WInvalidate

	MsgSharedReq,
	MsgExclusiveReq,
	MsgModifyReq
		if(this is the original home)
		  forward message to current home node
		else
		  forward message to original home

	MsgBecomeHomeReq	if(become home request in progress)
		  send MsgNegativeAck
		else
		  forward message to current home node

Table C.27: RemoteModifiedWip: protocol events and actions.

State	Event	Actions

RemoteBecomeHomeReq	MsgInvalidateAck,
	MsgInvalidateAckData
		if (number of expected MsgInvalidateAcks is 0)
		  setup home region structure
		  state = HomeExclusive
		  retry queued messages
		else
		  if (*num_invalidate_acks* is unset)
		    num_invalidate_acks = # of MsgInvalidateAcks expected
		  num_invalidate_acks -= 1
		  if (*num_invalidate_acks* == 0)
		    invoke *tx_cont*
		    send MsgBecomeHomeDone to ordinal home
		    state = HomeExclusive
		    retry queued messages

	MsgWInvalidate,
	MsgRInvalidate
		if (need to send data)
		  send MsgInvalidateAckData to requester
		else
		  send MsgInvalidateAck to requester

	MsgNegativeAck	send MsgBecomeHomeReq to original home
	MsgSharedReq, MsgExclusiveReq, MsgModifyReq, MsgBecomeHomeReq	queue message

Table C.28: RemoteBecomeHomeReq: protocol events and actions.

Bibliography

[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, David Chaiken, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, and Dan Nussbaum. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. Technical Report MIT/LCS/TM-454, MIT Laboratory for Computer Science, June 1991.

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

[4] 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.

[5] Satish Chandra, Brad Richards, and James R. Larus. Teapot: Language Support for Writing Memory Coherence Protocols. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[6] Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, and Thorsten von Eicken. Low-Latency Communication on the IBM RISC System/6000 SP. To appear in *Proceedings of Supercomputing '96*, November 1996.

[7] Chi-Chao Chang, Grzegorz Czajkowski, and Thorsten von Eicken. Design and Performance of Active Messages on the IBM SP-2. Technical Report CS-TR-96-1572, Cornell University, 1996.

[8] Comparison of MPL and MPI latency and bandwidth. Available on the World Wide Web at URL http://www.rs6000.ibm.com/software/sp_products/performance/switch.html.

[9] Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, Ioannis Schoinas, Mark D. Hill, James R. Larus, Anne Rogers, and David A. Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of Supercomputing '94*, pages 380–389, November 1994.

[10] 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.

[11] Sandeep K. Gupta, Alejandro A. Schäffer, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Integrating Parallelization Strategies for Linkage Analysis. *Computers and Biomedical Research*, 28:116–139, 1995.

[12] Erik Hagersten, Anders Landin, and Seif Haridi. DDM – a Cache-Only Memory Architecture. *IEEE Computer*, pages 44–54, September 1992.

[13] Mark D. Hill, James R. Larus, and David A. Wood. Tempest: A Substrate for Portable Parallel Programs. In *Proceedings of COMPCON Spring 95*, pages 327–332, March 1995.

[14] IBM SP/2 Home Page. Available on the World Wide Web at URL http://www.rs6000.ibm.com/.

[15] International Business Machines. *IBM AIX Parallel Environment, Operation and Use, Release 2.0*, June 1994.

[16] International Business Machines. *IBM AIX Parallel Environment, Parallel Programming Subroutine Reference, Release 2.0*, June 1994.

[17] International Business Machines. *IBM AIX Parallel Environment, Programming Primer, Release 2.0*, June 1994.

[18] Kirk L. Johnson. *High-Performance All-Software Distributed Shared Memory.* PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, 1995.

[19] 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/.

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

[21] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, pages 213–228, December 1995.

[22] 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.

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

[24] 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.

[25] 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.

[26] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, pages 41–61, January 1993.

[27] Kevin Lew. A Case Study of Shared Memory and Message Passing: The Triangle Puzzle. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1995.

[28] Kevin Lew, Kirk Johnson, and Frans Kaashoek. A Case Study of Shared-Memory and Message-Passing Implementations of Parallel Breadth-First Search: The Triangle Problem. In *Proceedings of the Third DIMACS International Algorithm Implementation Challenge Workshop*, October 1994.

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

[30] Honghui Lu, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. Message Passing Versus Distributed Shared Memory on Networks of Workstations. In *Proceedings of Supercomputing '95*, December 1995.

[31] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3-4):169–416, fall/winter 1994.

[32] PVM Home Page. Available on the World Wide Web at URL http://www.epm.ornl.gov/pvm/.

[33] 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.

[34] E. Rosti, E. Smirni, T.D. Wagner, A.W. Apon, and L.W. Dowdy. The KSR1: Experimentation and Modeling of Poststore. In *Proceedings of ACM SIGMETRICS '93*, pages 74–85, May 1993.

[35] Ashley Saulsbury, Tim Wilkinson, John Carter, and Anders Landin. An Argument for Simple COMA. In *Proceedings of the First Symposium on High Performance Computer Architecture*, pages 276–285, January 1995.

[36] Vaidy Sunderam, Al Geist, Jack Dongarra, and Robert Mancheck. The PVM Concurrent Computing System: Evolution, Experiences, and Trends. *Parallel Computing*, 20(4):531–545.

[37] 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.

[38] 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.