UNIVERSITY OF CALIFORNIA

Los Angeles

Hierarchically Managed Reliable Distributed Shared Memory Multicomputers

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

G. Janakiraman

1997

© Copyright by

G. Janakiraman

1997

The dissertation of G. Janakiraman is approved.

Ivo Welch

David A. Rennels

Milos D. Ercegovac

Yuval Tamir, Committee Chair

University of California, Los Angeles

1997

To my parents,

who have given me everything,

and my brothers, sisters, nieces and nephews,

who have been waiting with eagerness.

Table of Contents

Chapter One - Introduction 1

1.1. Hierarchically Managed DSM 4

1.2. Reliable DSM 5

1.3. Organization of this Work 6

Chapter Two - Distributed Shared Memory Issues 8

2.1. Consistency 9

2.2. Coherency 12

2.3. System Reliability through Fault Tolerance 16

2.4. Storage Overhead for Coherency Management in DSM Systems 20

2.5. Implementations of DSM Systems 21

Chapter Three - Hierarchical Management of DSM systems 27

3.1. Single-Level Management of DSM 29

3.1.1. Management at the Granularity of ‘‘Pages 31

3.1.2. Management at the Granularity of ‘‘Cache Blocks 32

3.1.3. Limited Copy Directories 33

3.1.4. Storage Overhead Example 37

3.2. Hierarchical Management of Distributed Shared Memory 37

3.2.1. The Two-Level Management Scheme 38

3.2.2. The Copy Directory 42

3.2.3. Storage Overhead of the Two-Level Management Scheme 45

3.3. Related Work 47

Chapter Four - An Implementation of Hierarchically Managed DSM 50

4.1. Memory Management Unit design 51

4.1.1. Present Pages Table and Shared Pages Table 55

4.1.2. Block Map per Processor Copy Directory 59

4.2. Distributed Directory Coherence Protocol Design Issues 62

4.2.1. Deadlock 64

4.2.2. Handling of Race Conditions 69

4.2.2.1. Race Conditions in DSM Systems 73

4.2.2.2. DSM Coherency based on Bilateral Exchanges 77

4.2.3. Necessity of Transient Block States 79

4.2.4. Handling of Write Requests 86

4.3. A Coherence Protocol for Hierarchically Managed DSM 88

4.3.1. Protocol Ownerships 89

4.3.2. Page and Block States 90

4.3.3. Protocol Messages 94

4.3.4. The Flow of Protocol Actions During Reads, Writes

and Replacements 96
4.3.4.1. Anatomy of a Read Request 100
4.3.4.2. Anatomy of a Write Request 108
4.3.4.3. Anatomy of a Replacement 116
4.4. Simulation of Our DSM System Design 123
4.4.1. Implementation of a Simple Trace-Driven Simulator 125
4.4.2. Implementation of an Execution-Driven Simulator 126
4.4.2.1. The Proteus Simulation Environment 127
4.4.2.2. Implementation of Hierarchically Managed DSM on Proteus 131
Chapter Five - Evaluation of Hierarchically Managed DSM 137
5.1. Evaluation based on Execution-driven Simulation 138
5.1.1. One-level Scheme Implementation 143
5.1.2. Application Benchmarks 146
5.1.3. Measurement of the Shared Working Set Size 155
5.1.4. Measurement of the Effectiveness of the Copy Directory Organization 158
5.1.5. Performance Comparison Between the Hierarchical and One-Level Schemes 163
5.1.6. Characterization and Measurement of Non-Local Read and Write Accesses 170
5.1.6.1. Measurement of the Miss Rate of

Shared Accesses 172

5.1.6.2. Classification of Protocol Flows Used to Satisfy Non-Local Requests 178

5.1.6.3. Breakdown of Non-Local Requests by Hop-Sequence Type of Protocol Flow 188

5.1.6.4. Breakdown of Non-Local Request Latency 198

5.1.7. Profile of Execution Time 215

5.2. Evaluation based on Trace-driven Simulation 221

Chapter Six - Error Recovery for Distributed Shared Memory Systems 231

6.1. Coordinated Checkpointing and Rollback 233

6.2. System Model and Assumptions 235

6.3. Coordinated Checkpointing and Rollback for DSM Systems 236

6.3.1. Tracking Interprocess Communication 237

6.3.2. Transactions in Progress when Checkpointing or Rollback are Initiated 239

6.3.3. Checkpointing 242

6.3.4. Recovery 245

6.3.4.1. State Restoration 246

6.3.4.2. Directory Repair 249

6.4. Coordinated Checkpoint and Rollback for Hierarchical DSM Systems 251

6.4.1. The Hierarchical DSM System 252

6.4.2. Tracking Interprocess Communication 253

6.4.3. Handling of Pending Transactions 255

6.4.4. Checkpointing and Recovery of Process States 256

6.4.5. Directory Repair 257

6.5. Previous DSM Recovery Schemes 261

6.6. Evaluation 263

6.6.1. Results for the Block-Level DSM system 265

6.6.2. Results for the Hierarchical DSM System 267

Chapter Seven - Summary 273

Bibliography 277

List of Figures

1.1 Multicomputer 2

2.1 System with shared bus for snoop-based coherence protocols 13

2.2 Centralized directory for directory-based coherence protocol 14

2.3 Distributed directory for directory-based coherence protocol 15

3.1 A multicomputer node 30

3.2 Address translation tables for the block-level scheme 33

3.3 Copy directory analysis: access bursts prior to invalidate 36

3.4 Address translation tables for the two-level scheme 39

3.5 Address translation for local requests 41

3.6 Processor List per Block (PLpB) copy directory organization 43

3.7 Block Map per Processor (BMpP) copy directory organization 44

3.8 BMpP copy directory size 46

4.1 Memory Management Unit Design 52

4.2 Present Pages Table (PPT) entry format 55

4.3 Shared Pages Table (SPT) entry format 56

4.4 BMpP copy directory organization 60

4.5 Deadlock due to competing writes 66

4.6 Deadlock avoided by sinking invalidate 67

4.7 Forwarding in Protocol 72

4.8 Races in protocols with forwarding 74

4.9 Race condition that cannot be identified locally 76

4.10 Protocol based on bilateral exchanges 78

4.11 Protocol based on bilateral exchanges avoids races 79

4.12 Protocol actions for a read access 81

4.13 Replacement during read access 83

4.14 Addition of new protocol state to handle replacement 84

4.15 Comparison of bilateral exchange and forwarding 97

4.16 Anatomy of a read request 100

4.17 Read access when page owner is not known locally 103

4.18 Additional protocol flow sequences for read requests 104

4.19 Race between *ACK* and *RREQ* 105

4.20 Anatomy of a write request 112

4.21 Additional protocol flow sequences for write requests 114

5.1 Model of the execution-driven simulation. 138

5.2 One-level scheme protocol flow sequences for read requests 144

5.3 One-level scheme protocol flow sequences for

write requests 145

5.4 Distribution of application instructions amongst processors. 152

5.5 Distribution of application shared reads amongst processors. 153

5.6 Distribution of application shared writes amongst

processors. 154

5.7 Small Shared Working Set. 157

5.8 BMpP organization of the copy directory is more efficient. 161

5.9 Copy Directory can be bounded. 162

5.10 Performance of the hierarchical scheme. 163

5.11 False sharing is higher at larger block sizes. 166

5.12 Latency of remote read accesses is higher at
larger block sizes. 168

5.13 Processor utilization is lower at larger block sizes. 169

5.14 Miss Rate for Shared Accesses. 173

5.15 Break down of read accesses by hop-sequence type in
the one-level scheme. 189

5.16 Break down of read accesses by hop-sequence type in
the hierarchical scheme. 191

5.17 Break down of write accesses by hop-sequence type in
the one-level scheme. 195

5.18 Break down of write accesses by hop-sequence type in
the hierarchical scheme. 196

5.19 Access Latency Breakdown (Cholesky). 207

5.20 Access Latency Breakdown (Cholesky) with MBT access
latency of 4 cycles. 210

5.21 Access Latency Breakdown (LU). 211

5.22 Access Latency Breakdown (LU) with MBT access latency of 4 cycles. 212
5.23 Access Latency Breakdown (Radix). 213
5.24 Access Latency Breakdown (Radix) with MBT access latency of 4 cycles. 214
5.25 Execution cycles breakdown (LU). 218
5.26 Execution cycles breakdown (Cholesky). 219
5.27 Execution cycles breakdown (Radix). 221
5.28 Data miss percentage versus SPT size: 4KB pages 224
5.29 Data miss percentage versus SPT size: 1KB pages 225
5.30 Network traffic versus SPT size: 4KB pages 226
5.31 Network traffic versus SPT size: 1KB pages 227
5.32 BMpP copy directory performance 228
5.33 Performance comparison of BMpP and PLpB organizations 229
6.1 Checkpoint Mechanism 244
6.2 Recovery Mechanism 247
6.3 Repair of the directory state 250
6.4 Repair of coherency directory in a hierarchical DSM system — 1 260
6.5 Repair of the coherency directory in a hierarchical DSM system — 2 261
6.6 Network traffic with checkpoints versus block size: single-level

coherency scheme 265

6.7 Relative traffic overhead and number of checkpoints in a block-level system versus checkpoint interval 266

6.8 Comparison of network traffic in hierarchical and block-level systems at various block sizes 268

6.9 Comparison of interacting set size and checkpoint interval on the hierarchical and block-level systems 270

6.10 Comparison of relative traffic overhead in the hierarchical and block-level systems at various block sizes 271

List of Tables

3.1 Comparison of the storage overhead in the one-level and two-level schemes 45

4.1 Deadlock due to contention for physical resources 69

4.2 *access_permits* of a block 91

4.3 *access_waitlists* of a block 92

4.4 block states — composition of access_permit and access_waitlist 93

4.5 Protocol messages and their function 94

5.1 Table Access Latencies. 141

5.2 PPT, SPT and Copy directory sizes in simulations. 142

5.3 Memory usage in application benchmarks. 148

5.4 Number of instructions and shared accesses in the applications. 149

5.5 Number of instructions and shared accesses in the parallel phase of the applications. 150

5.6 Number of Shared Reads Satisfied by Disk Accesses. 175

5.7 Number of Shared Writes Satisfied by Disk Accesses. 176

5.8 Hop-sequences of protocol for read access in one-level scheme. 180

5.9 Hop-sequences of protocol for read access in hierarchical scheme. 183
5.10 Hop-sequences of protocol for write access in one-level scheme. 185
5.11 Hop-sequences of protocol for write access in hierarchical scheme. 187
5.12 Average Link Latency and Queuing Delay of Network Messages 201
5.13 Summary of cost of hop-sequences of protocol for read accesses. 202
5.14 Summary of cost of hop-sequences of protocol for write accesses. 203
5.15 Hop-sequence used as base type in access latency breakdown analysis. 205
5.16 Trace characteristics 222

ACKNOWLEDGMENTS

I have been fortunate to have had the valuable guidance of a number of people in pursuing my research aspirations. The camaraderie of many more has also made the whole experience enjoyable. I wish to express my gratitude for all their support.

A large part of my research accomplishments would have not been possible without the guidance of my advisor, Yuval Tamir. Yuval was instrumental in my choosing this topic for my dissertation. The quality of my research has greatly benefited from his thorough discussion of ideas. Through his experience, and vast knowledge, he has taught me a great deal. By working with me on our research papers, our research presentations, and my recitation sections, he has also improved my ability to describe concepts and ideas clearly. I thank him for all his contributions to my research, and for his efforts in ensuring that the research set a good foundation for my career.

I am very grateful to Prof. Milos Ercegovac, Prof. David Rennels, and Prof. Ivo Welch for being on my committee and providing insightful comments. Prof. Ercegovac and Prof. Rennels have also broadened my knowledge of computer architecture and improved my teaching skills, when I was a teaching assistant for their courses. I would like to give special thanks to Prof. Richard Muntz for graciously taking the time to discuss my research and my career, even though he was not on my committee.

Several people in the department have helped me in many ways. Verra Morgan has been extremely generous with her help, irrespective of the time or task. Verra's support will always be special, and I cannot thank her sufficiently for what she has given. I thank Shelly Backlar, and Diane Barrie for, among other things, making my

coffee-breaks very relaxing.

It has been a pleasure to work with my fellow office-mates in the VLSI Systems Laboratory, Yoshio Turner, Hsin-Chou Chi, Tiffany Frazier, Greg Frazier, Gonzalo Archondo-Callao and Peter Trajmar. Yoshio has been especially helpful with critical discussions. I owe my colleagues in the Department of Radiological Sciences, Lu Huang, Kinchi Kong, and George Zachariah thanks for the good cheer that they brought to my work atmosphere.

I am extremely grateful to my room-mates and friends Deepak Nayak, Vikas Jha, Ashish Gupta, and Prasasth Palnati for providing a comfortable atmosphere to unwind from research. I am equally thankful for the entertaining company of Kannan Rangaramanujam, Jai Shankar, Chitra Shankar, Kumar Srinivasan, Bhuvana Srinivasan, Ravi Raj, and Bhuvana Raj.

Finally, I would like to thank my parents and family for all their encouragement. Their motivation and support have been crucial to me in this endeavour.

VITA

October 24, 1966	Born in Madras, India
1988	B.E. (Hons.) Electrical and Electronics Engineering Birla Institute of Technology and Science Pilani, India
1988	M.Sc. (Tech.) Computer Science Birla Institute of Technology and Science Pilani, India
1984-1988	BITS Merit Scholarship Birla Institute of Technology and Science Pilani, India
1988	BITS Gold Medal Birla Institute of Technology and Science Pilani, India
1988	Bhamashah Award Maharana of Mewar Foundation Mewar, India
1988-1993	Teaching Assistant, Teaching Associate, Teaching Fellow University of California, Los Angeles Los Angeles, California
1991-1993	Teaching Assistant Consultant University of California, Los Angeles Los Angeles, California
1994	Nomination for Distinguished Teaching Assistant Award University of California, Los Angeles Los Angeles, California
1989-1996	Research Assistant, Research Associate University of California, Los Angeles Los Angeles, California

PUBLICATIONS AND PRESENTATIONS

1. G. Janakiraman and Y. Tamir, ‘‘Coordinated Checkpointing-Rollback Error Recovery for Distributed Shared Memory Multicomputers,’’ *Symposium on Reliable Distributed Systems*, Dana Point, CA, pp. 42-51 (October 1994).

2. G. Janakiraman and Y. Tamir, ‘‘Coordinated Checkpointing-Rollback Error Recovery for Distributed Shared Memory Multicomputers,’’ Computer Science Department Technical Report CSD-940027, University of California, Los Angeles, CA (July 1994).

3. Y. Tamir and G. Janakiraman, ‘‘Hierarchical Coherency Management for Shared Virtual Memory Multicomputers,’’ *Journal of Parallel and Distributed Computing* **15**(4), pp. 408-419 (August 1992).

4. Y. Tamir and G. Janakiraman, ‘‘Multi-Level Coherency Management for High-Performance Shared Virtual Memory Multicomputers,’’ *IEEE International Phoenix Conference on Computers and Communications*, Scottsdale, AZ (March 1991).

5. Y. Tamir and G. Janakiraman, ‘‘Multi-Level Coherency Management for High-Performance Shared Virtual Memory Multicomputers,’’ Technical Report, CSD 900024, University of California, Los Angeles (August 1990).

ABSTRACT OF THE DISSERTATION

Hierarchically Managed Reliable
Distributed Shared Memory Multicomputers

by

G. Janakiraman
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1997
Professor Yuval Tamir, Chair

Distributed Shared Memory (DSM) Multicomputers offer the advantages of the shared memory programming model on high-performance, scalable hardware. DSM implementations must not limit scalability or introduce performance bottlenecks. The reliability requirements of large DSM multicomputers can only be met using fault tolerance techniques.

Most DSM implementations manage memory and coherency at a single granularity of fixed size pages, resulting in unnecessary network traffic due to false-sharing, or of cache blocks requiring unacceptably large mapping tables to map large address spaces. We propose hierarchical management as a solution where mapping and transfer occur at the block level only when the page is actually shared. When pages are not shared, they incur the same mapping overhead as uniform page-level schemes. Performance evaluation based on detailed execution-driven simulation has shown that hierarchical management significantly outperforms page-level management, and performs nearly as well as block-level management, while using much smaller mapping tables, on real applications.

Error recovery schemes previously proposed for DSM systems checkpoint or log data at every process communication, leading to unnecessarily high checkpointing frequency or logging overhead, and associated network traffic. Such approaches are not suitable for scalable systems intended for communication-intensive applications. Efficiently implementing the lower overhead error recovery approach based on coordinated checkpointing and rollback on DSM multicomputers presents several challenges. We solve these problems and implement this approach for the DSM multicomputer. Our performance evaluation indicates that coordinated checkpointing and rollback incurs less checkpoint traffic than recovery schemes previously proposed for DSM systems.

Chapter One
Introduction

Distributed Shared Memory (DSM) Multicomputers have the potential to provide high-performance by exploiting parallelism, while allowing the programmer to express the parallelism using the familiar shared memory programming model. The multicomputer type of system [Atha88] is composed of a large number of independent processing nodes (comprising processor as well as local memory) which are interconnected via point-to-point links. Unlike shared-memory multiprocessors [Thak88, Wils87, Good88] where processors are connected by buses, the multicomputer has no single component, such as a common bus or shared memory, which can become a performance bottleneck and whose correct operation is critical for the operation of the system. Multicomputers are, hence, capable of scaling to sizes larger than shared memory multiprocessors.

While the memory in a multicomputer is distributed, the DSM multicomputer provides a shared memory programming model where processes running on different nodes can execute in the the same address space. Interprocess communication in this environment is accomplished by reads and writes to shared memory, rather than through message-passing primitives such as send and receive. The shared memory model provides a straightforward means of developing applications that require different processes to share access to a large common data structure (e.g. a memory-resident collection of images), and applications that require complex data structures to be passed by passing pointers. Support for the shared memory programming model in new systems is also needed to facilitate

porting of the many existing applications written for this model. Another advantage of the shared memory model is that process migration (e.g., for load balancing) becomes relatively cheap — in order to migrate a process only the contents of a few registers need to be migrated while the memory can remain in place.

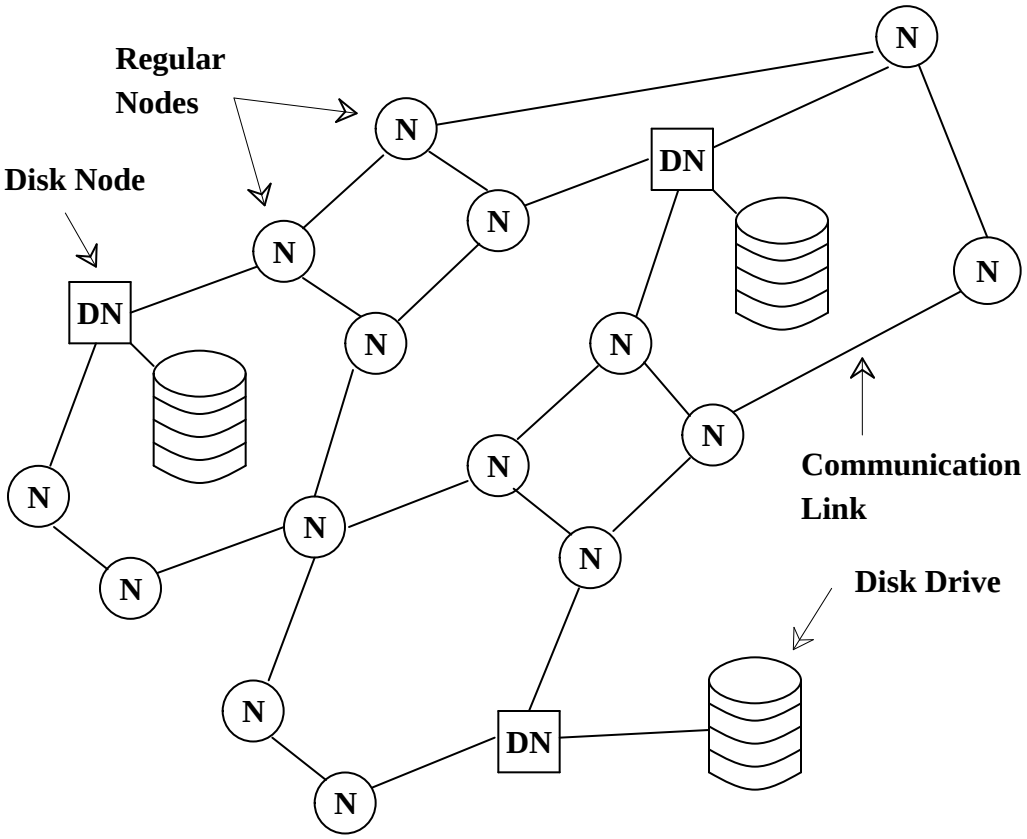


Figure 1.1: Multicomputer.

DSM multicomputers allow individual nodes to cache the data from the distributed shared address space locally to support better performance by reducing the number of remote accesses. If the word requested by a process is not available locally, the block containing the word is read either from a remote memory or from

a disk and cached in local memory. The block resides in local memory until another processor requests exclusive access to it, or, as in conventional virtual memory, until it is replaced by another block. Copies of any block may exist in multiple nodes. The DSM implementation employs a coherence protocol to maintain the node memories coherent, and guarantees that process execution is consistent.

DSM multicomputers have been the focus of significant research and development [Li86b, Cher91, Hage92, Burk92, Leno90, Kusk94, Agar95] due to its advantages. These efforts have investigated various issues in caching policies, coherence protocols, consistency models, hardware organization etc. Many of the proposed or implemented systems have not fully addressed critical memory mapping and memory management issues that arise while supporting DSM on a large scale multicomputer. We address these issues in this dissertation and propose a solution based on hierarchical management [Tami92] that manages the distributed shared memory more efficiently than conventional single level schemes, making it suitable for large systems.

The large number of components in a DSM multicomputer increases the likelihood that some component of the system may fail at some time. Since the shared address space is distributed and replicated, such failures have the potential to halt progress of the entire system. Fault-tolerance support is needed to tolerate such component failures and improve the reliability of the system. Fault-tolerance schemes that have been proposed for DSM systems by other researchers [Wu90a, Tam90, Stum90] have adopted approaches which incur high overhead during normal operation and are more expensive for communication-intensive

applications. We propose an application-transparent recovery scheme based on coordinated checkpoint and rollback for the DSM multicomputer [Jana94] that can potentially offer better performance than previous recovery schemes.

1.1. Hierarchically Managed DSM

Most existing techniques for providing distributed shared memory on multicomputers perform the mapping, data transfers, and ownership management at a single granularity of blocks (or pages). If large block sizes (pages) are used [Li89a], these techniques suffer from unnecessary long latencies for remote requests, they require higher network bandwidth than is inherently needed by the application, and they lead to unnecessary transfers due to false sharing. A smaller block size (blocks) [Leno90, Kusk94, Agar95] will lead to increased block ownership time [Bayl89], a decrease in miss ratio [Dubo88], and is expected to reduce remote access latencies and the required network bandwidth. A straightforward implementation at a block level for a large DSM system, however, will result in unacceptably high storage overhead for the various mapping tables.

We present *hierarchical management* [Tami92] as a solution to this problem. The proposed scheme dynamically distinguishes between shared and private pages in order to manage each class efficiently. Pages that are not shared are mapped and managed only at the granularity of pages, reducing mapping overhead. Shared pages, however, are managed at both the page and block level, to support efficient sharing. This distributes the coherency load according to the characteristics of the application. Our evaluation indicates that significant performance improvement over conventional page-level schemes are possible with our two-level scheme,

without incurring the storage overhead that makes uniform block-level schemes impractical in large systems.

1.2. Reliable DSM

Most error recovery schemes are based on periodically saving a correct state (checkpointing) in reliable storage, and restoring a previously saved state (rollback) upon an error in the current state. Since process states in a DSM system are interdependent due to interprocess communication, when a process rolls back to a checkpointed state at recovery, the states of other processes that communicated with it prior to failure may become invalid. Hence, other processes may also need to be rolled back (propagation of rollback). Error recovery mechanisms proposed for DSM systems (as for message passing systems) must ensure that process states are consistent [Rand78, Chan85] after recovery from an error.

Recovery schemes that have previously been proposed for DSM systems [Wu90a, Tam90, Stum90] all attempt to avoid the cost of rollback propagation by forcing a checkpoint or logging when processes ‘‘communicate.’’ The elimination of rollback propagation make recoveries, which are infrequent, cheap. But, due to the frequent checkpointing or logging, the system load during normal operation is high. The performance degradation is particularly high for communication-intensive applications [Fraz89].

Coordinated checkpointing and rollback based error recovery is an alternative approach used in message-passing systems that incurs lower overhead during normal operation of the system [Bari83, Tami84, Koo87, Tami89]. In this approach, process interactions are tracked, and processes that have communicated

are checkpointed and rolled back together. Adopting this approach for DSM systems efficiently presents many difficulties [Jana94]. Since processes communicate through read and write accesses to a common address space, rather than through explicit message sends and receives between specific participants, process interactions are less readily identified. The handling of reads and writes that are in progress during checkpoint or recovery is more complicated than the handling of messages in transit due to the coherence protocol. Since the address space and the coherence directory are shared, a naive implementation can propagate checkpoints and recoveries globally leading to a checkpoint or rollback of the entire system. We present an adaptation of coordinated checkpoint and rollback based error recovery for the DSM system [Jana94] which handles these problems efficiently, and prove the correctness of the scheme. Our evaluations indicate that this approach has the potential to reduce checkpointing overhead over previous schemes.

1.3. Organization of this Work

We begin by discussing (in Chapter 2) several issues that are relevant in the implementation of distributed shared memory on a multicomputer. The chapter introduces the problems of consistency, coherency and reliability in the DSM multicomputer, and discusses the performance and storage overhead associated with the implementation in a large scale system. We also briefly describe the salient features of other implementations of distributed shared memory.

Our hierarchical management scheme for DSM is presented in detail in Chapter 3. Chapter 4 describes our implementation of DSM, for execution-driven

and trace-driven simulations, in detail. It describes our innovations in the design of the coherence protocol, and the support for DSM in the memory management unit. The results of our performance evalution of the hierarchical scheme using execution-driven and trace-driven simulations are presented and discussed in Chapter 5.

In Chapter 6 we present an adaptation of coordinated checkpoint-rollback based error recovery to a DSM system. We describe the scheme for a DSM system managed at a single level of granularity as well as extensions for a hierarchically managed DSM system. We present results from trace-driven simulations comparing the performance of the coordinated checkpoint-rollback scheme with another scheme proposed earlier.

Chapter Two
Distributed Shared Memory Issues

Distributed shared memory provides a virtual machine where processes executing on all nodes share a common address space. The shared address space is physically distributed among the nodes in the system. In most implementations of DSM, the distributed address space is allowed to be replicated and cached locally by the nodes for faster access. The distribution and replication of the address space raise several issues that need to be addressed to guarantee ‘‘correct’’ execution of applications. First, since different accesses are served by different nodes, the access paths and access latencies differ from access to access. Therefore, accesses from different processes can get ordered in several different ways. Second, more than one access to the same address can be serviced concurrently. The DSM implementation should ensure that access orderings that lead to incorrect execution of the application are not possible. Specifically, concurrent accesses to the same address should be managed so as to provide the abstraction of a single common memory module.

The support of DSM involves costs in the storage overhead of tables, as well as in the performance overhead of moving data. Both these costs grow with the size of the system. Naive mechanisms for supporting DSM can load large systems excessively, making the use of DSM impractical for large systems. DSM support mechanisms must be designed such that they scale well to larger systems. The reliability of the DSM multicomputer is another important issue to be addressed. Since the distributed address space in a DSM system is shared by all processors on

the system, when a single node's memory fails, a part of the shared address space is lost. Hence the failure of a single node in the system can halt the progress of the entire system (similar scenario is also possible in message-passing multicomputer systems). Error recovery mechanisms that improve the reliability of the DSM system must also be developed.

We describe these issues here in more detail, and discuss some approaches to handling these issues. Finally, we summarize the key aspects of other DSM architectures and highlight their approach to these issues.

2.1. Consistency

Processes interact with memory through read and write accesses. Write accesses modify the contents of locations in memory. Read accesses retrieve the contents of memory locations. By reading the state of memory, a process may ''observe'' the order in which the write accesses were completed by memory. For example, consider write accesses to two different memory locations issued by two processes. By reading these locations, a third process may observe (determine from its perspective) that one of these write accesses was completed before the other. This third process may thus establish (from its perspective) an ordering between the two write accesses. A *consistency model* defines the possible relationships between the order in which accesses are initiated by a process and the order in which these accesses may be observed by other processes in the system. Applications may rely on a well-defined consistency model for correctness. For example, in the implementation of mutual exclusion without hardware interlocks [Lamp79]. If memory access orderings that are not expected by the

application are possible, the application can execute incorrectly (such as multiple processes entering a critical section). The implementation of an architecture must ensure that only orderings legal in the consistency model are possible during execution.

On conventional, simple uniprocessors, only one thread of computation (e.g., process) is active at any instant. This single thread of control provides a simple consistency model which is straightforward to understand and easy to implement. In this consistency model, all accesses issued to memory are totally ordered in real time, and each memory access observes the effects of all preceding accesses. Hence, all the accesses of a process in this system can be sequentially ordered.

In a DSM system, accesses can be issued to memory in parallel by multiple processes. Since these accesses can be serviced concurrently by the distributed and replicated memory, the ordering of memory accesses in such a system is not straightforward. Since multiple paths exist between processors in the multicomputer and data is dynamically replicated in DSM, a set of accesses that are serviced concurrently can be ordered in any of several possible ways. Furthermore, the mechanism used to access the distributed memory may allow different processes to observe different orderings. Hence, it is essential to establish a consistency model for this system.

Different consistency models impose different requirements on the implementation and on the application programmer or compiler. A consistency model that places strong constraints on the access ordering requires a conservative implementation, which can limit the performance of the system [Ghar91]. However, such a model can ease the programmer's (or compiler's) task of writing

correct programs since the implementation has limited freedom in ordering accesses. A consistency model that is weaker provides the hardware flexibility to choose orderings that support more parallelism. This enables aggressive (potentially costlier) implementations that can support higher performance. However, the programmer will have more responsibility in writing the application such that only access orderings which lead to correct application execution are possible.

Several consistency models have been proposed in the literature. The model that is considered ‘‘intuitive’’ by most programmers is *sequential consistency* [Lamp79]. Sequential consistency derives from the execution model of a sequential system. It requires that the execution on a parallel system is equivalent to some interleaving of the execution of the parallel processes on a sequential machine. Scheurich and Dubois [Sche87] have identified conditions that are sufficient to guarantee that a system is sequentially consistent. Specifically, three conditions must hold:

1) Each processor must issue all its memory accesses in the order that they appear in the program.
2) A write access must block the processor issuing the access until it is guaranteed that no processor in the system can read an older value.
3) A read access must block the processor until a value is received for the read and it can be guaranteed that no processor in the system can receive an older value.

The sequential consistency model is conceptually simpler than weaker consistency models such as *weak ordering* [Dubo86, Adve90]. In the weak

ordering model, all synchronization accesses must be identifiable by the hardware, and the hardware need only guarantee that all synchronization accesses can be ordered sequentially. By allowing a processor to continue without blocking on non-synchronization accesses, a weakly ordered system can support better performance than the sequentially consistent system. However, the execution of an application on the weakly ordered system will appear sequentially consistent only if all shared variables in the program are accessed within synchronization constructs that are visible to the hardware.

2.2. Coherency

To support higher performance, multiprocessor systems (including DSM multicomputers) allow the same shared data to be cached by multiple processors. Since several copies of a memory location exist, the memory system can service several accesses to the same location at the same time. The memory system is defined to be *coherent* if all write operations to any given memory location are observed by all processors in the system in the same order [Arch87]. Coherency allows a system to provide the abstraction of a single master copy while servicing multiple accesses concurrently.

The replication of data enables several processors to concurrently operate on the same memory location with low latency. However, it allows the possibility that the system may associate more than one value with a memory location at the end of an execution. In addition, the value at the end of execution may not be the value written by the ‘‘last’’ write to the location. A coherence protocol maintains coherence by managing all the copies such that the effect of the operations on the

several copies are equivalent to a single sequence of operations on one master copy.

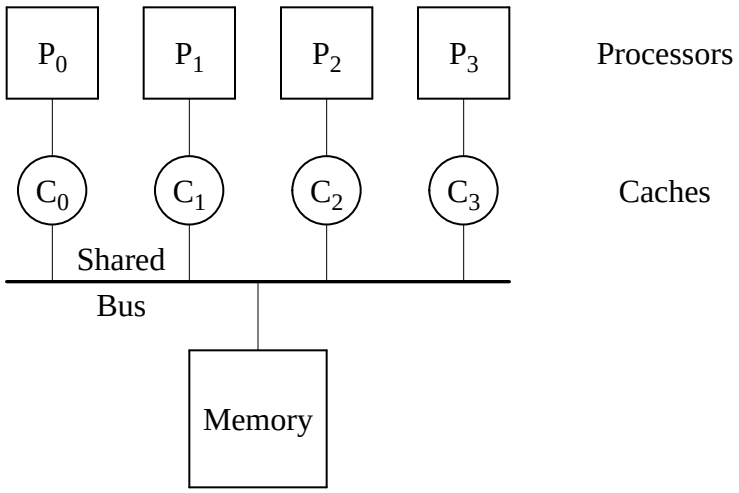


Figure 2.1: System with shared bus for snoop-based coherence protocols.

Maintaining *cache coherence* is a critical requirement in shared memory multiprocessor architectures [Cens78, Arch85, Agar88, Li89b]. Cache coherence protocols can be divided into two classes: *snoop-based coherence protocols* [Katz85, Good83] and *directory-based coherence protocols* [Tang76, Agar88]. Snoop-based coherence protocols are suitable only for systems where processors share a common communication medium such as a bus (Figure 2.1). Each processor cache in the system can snoop on all traffic on the bus and is required to take appropriate actions for addresses that are present in the cache.

In order to implement snoop-based coherence protocols, each processor cache maintains the status of its blocks as shared or unshared. When a processor writes to a shared block, the write is propagated on the bus. Upon seeing the write bus transaction, all other processors that have a copy of the block either invalidate their

copy of the block (in an invalidation protocol), or update their copy of the block (in an update protocol). Upon a read miss, the cache issues the access on the bus. The requested block is provided by the main memory, if the main memory has the latest value, and by another cache, if the main memory's value is stale. Snoop-based cache protocols are not suitable for large systems since the shared bus can only support a limited number of processors (about 30) without degrading performance.

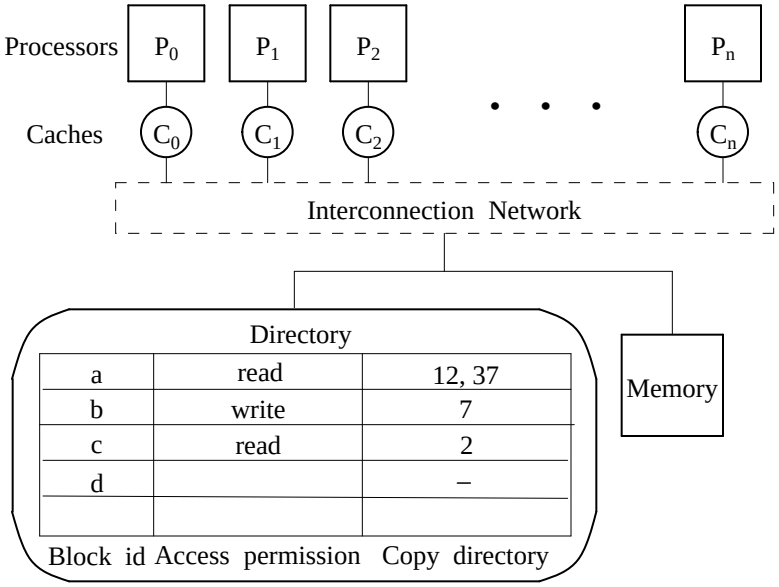


Figure 2.2: Centralized directory for directory-based coherence protocol.

Directory-based protocols [Tang76, Cens78, Agar88, Li89b] do not rely on a broadcast network such as a shared bus, and, hence are more suitable for large systems. Due to the absence of a snooping medium, directory-based protocols use a directory which maintains, for each block, the state of the block as shared or exclusive, and the set of nodes that have copies of the block (we call this part of the directory the *copy directory*). In the absence of a bus, point-to-point messages are

used for communication amongst the nodes and the directory. Upon a write to a shared block, the directory is looked up to determine the nodes that have copies of the block. These copies are invalidated (invalidation-based) or updated (update-based) to perform the write. The directory can be centralized at the memory (Figure 2.2) or at a specific node. The broadcast network (bus) is a performance bottleneck which is removed when we use a directory. However, the centralized directory is a performance and reliability bottleneck.

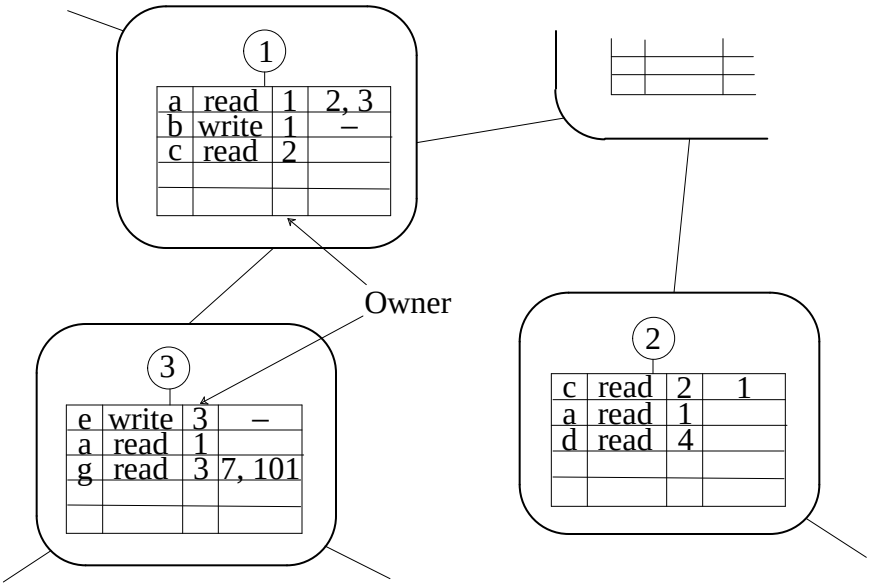


Figure 2.3: Distributed directory for directory-based coherence protocol.

Distributed directory-based coherence protocols alleviate the problem of a central directory becoming a bottleneck by distributing the directory among the nodes in the system. Li et al. [Li89b] have explored several different ways to distribute the directory. With the distributed fixed manager protocol, each node is statically assigned a set of blocks to manage. A node maintains the directory for

the blocks that it manages (Figure 2.3). In the distributed dynamic manager protocol, the manager of a block is dynamically determined. Typically, the node that last wrote the block maintains and manages the directory for that block. Since a block's manager is not fixed in the distributed dynamic manager protocol, an additional mechanism to locate the current manager of blocks is needed.

While directory-based coherence protocols scale better to large systems than snoop-based coherence protocols, they are more complex to design and verify. In a snoop-based system, the shared bus can serve as a mutual-exclusion lock. Since only one processor can access the lock (bus) at a time, all accesses are effectively ordered sequentially. In a system where processors are interconnected by a network, multiple paths are available. Consequently, a directory-based coherence protocol implementation for such a system must address issues of message races, deadlocks etc.

2.3. System Reliability through Fault Tolerance

Components of any system are inherently susceptible to faults. The failure of these components can lead to the failure of the system. Fault tolerance is an approach that improves system reliability by allowing the system to operate correctly despite the failure of some of its components. Fault tolerance mechanisms are essential for meeting the reliability requirements of critical special-purpose applications. Fault tolerance may also be necessary for general-purpose applications which require large multicomputer systems to continuously operate correctly for hundreds of hours. Large systems may be composed of several thousand chips, which have individual transient failure rates of 1000 - 6000

per billion part hours. If the entire system can fail due to the failure of a single chip, the system will have a mean time between failures of only tens of hours. Hence, without fault tolerance, large systems may be unable to meet their reliability requirements.

Error recovery techniques implement fault tolerance by restoring the system to a valid state following an error. In error recovery based on *checkpointing* and *rollback* [Rand78, Bari83, Tami84, Koo87, Borg83, Powe83, John87, Stro85, John88], an error-free state of the system (checkpoint) is periodically saved to stable storage. Upon an error in the system, the system recovers to a correct state by restoring (rollback) a previously checkpointed state. In a system where processes do not communicate with each other, process states are not dependent on each other. Hence, processes can be checkpointed and recovered independently in such systems.

When processes communicate with each other (as in DSM multicomputer systems), their states are interdependent. In such systems, checkpointing and rollback must guarantee that process states are always *consistent* [Merl78, Chan85] with each other. For example, in a message-passing system, process states are inconsistent if one process state indicates that a specific message was received while the states of other processes indicate that the message was never sent. If processes are checkpointed independently ignoring their interactions, recovery is entirely responsible for ensuring that the set of states to which the processes are rolled back is consistent. In such cases, recovery can lead to the domino effect [Rand78], where processes are rolled back to successively older checkpoints in the search for a consistent state, possibly even to their initial state. The domino

effect can be avoided by checkpointing processes at each interprocess communication. But, this approach is expensive. *Message logging* and *coordinated checkpointing* are two approaches that avoid the domino effect without incurring this high cost.

Message logging schemes [Borg83, Powe83, John87, John88, Stro85, Sist89] checkpoint process states periodically, and log all interprocess messages in sequence. A failed process is recovered to a consistent state by restoring its checkpointed state and replaying its logged messages. Coordinated checkpointing [Bari83, Tami84, Koo87] is based on checkpointing ‘‘together’’ consistent states of all the processes that have interacted since their last checkpoint. Recovery restores a consistent state by rolling back all processes that interacted with the failed process.

Both message logging and coordinated checkpointing were first proposed in the context of message-passing multicomputers. Message logging has the advantage that processes can checkpoint and recover independently. However, coordinated checkpointing incurs less overhead since messages are not logged [Fraz89]. In addition, unlike message logging, coordinated checkpointing does not require deterministic process execution.

Error recovery mechanisms have also been proposed for shared memory multiprocessor systems with broadcast interconnection networks [Bern88, Wu90b, Ahme90]. The Sequoia multiprocessor [Bern88] maintains a consistent state in failure-free main memory and periodically checkpoints the system through operating-system controlled cache flushes. The system tolerates single failures by invalidating the cache blocks, and re-executing from the main memory state. The

checkpoint-rollback mechanisms presented in [Wu90b] and [Ahme90] assume that the processor caches are reliable. In [Wu90b], independent checkpoint and rollback are facilitated by checkpointing a processor before it provides modified data to another processor. At checkpoint, processor registers and modified cache blocks are flushed to memory. Processors can recover independently by invalidating the cache contents and restoring the processor registers. Ahmed et al. [Ahme90] present schemes implementing coordinated checkpoint and rollback which take advantage of the ability to snoop in a bus-based shared-memory multiprocessor. The scheme uses three special bus lines which identify when data is shared, when a checkpoint is to be taken, and when processors should roll back. The line indicating shared data is used to identify interprocessor communication. The other two special bus lines are used to force coordinated checkpoint and recovery, respectively, of the processors that have communicated.

In order to implement error recovery in a distributed system, interprocess communication must be identified. In a message-passing system, processes communicate through readily identifiable messages. The task of identifying interprocess communication is more difficult in a DSM system, where processes communicate through accesses to the shared address space. Since, unlike bus-based multiprocessors, DSM multicomputers do not communicate through broadcasts, they cannot use snooping to identify interprocess communication. Unlike message-passing systems and bus-based shared memory multiprocessor systems, DSM multicomputers employ a coherence directory to manage the address space. This coherence directory is also susceptible to node failures. Since the coherence directory state is not a part of the state of any application process,

the directory state will not be recovered by the mechanisms recovering application process states. Hence, error recovery in DSM multicomputers must incorporate additional mechanisms to handle the failure of the distributed directory.

2.4. Storage Overhead for Coherency Management in DSM Systems

The support of a shared address space on a distributed memory multicomputer requires the maintenance of several tables and directories, as well as the use of a coherence protocol. Tables are required to identify which virtual addresses are mapped to physical memory. For the coherence protocol, coherency directories must maintain the state of blocks (shared, exclusive etc.) in the system as well as the location of copies of these blocks.

Conventional single-level tables incur a very high storage overhead in mapping the large virtual address spaces of large systems. These tables maintain an entry for every virtual page in the system. Hence, their size grows with the size of the virtual address space rather than the size of the physical memory. The size of these tables can be much larger than the physical memory in any node.

A related issue which is also important is the granularity at which shared data is managed. Larger granules require smaller mapping tables. Hence, with larger granules, larger address spaces can be mapped for a given mapping table size. However, since large granules contain more variables, more processors can concurrently require access to a single granule (possibly for different variables). This can lead to increased contention for granules between multiple nodes (false sharing). Also, larger granules incur higher network transfer latency.

Another important design issue is the organization of the copy directory,

which maintains the location of copies. If the copy directory maintains the identities of all nodes that possess copies, its size grows linearly with the number of nodes in the system. In large systems, the storage available per node may not be large enough to accommodate the copy directories of all blocks managed by the node. Hence, this straightforward copy directory organization may not be scalable to larger systems. While the copy directory can be eliminated by using broadcasts to invalidate (or update) the block at a write, this will degrade performance since broadcasts on point-to-point networks are usually expensive.

In summary, with a straightforward implementation, the size of the tables and directories required to manage the shared address space increases in proportion to the size of the address space and the number of nodes. However, the sizes of tables and directories that can be maintained is limited by the storage available per node. Since the storage available per node may not grow proportionately with the size of the system (the number of nodes and the size of the address space), the size of the system where DSM can be supported effectively may be limited. Thus there is a need for DSM support mechanisms that can scale up to large systems without requiring significant increases in size of mapping and coherence tables on each node.

2.5. Implementations of DSM Systems

The idea of implementing shared memory in a system with physically distributed memory is not new. Several implementations of distributed shared memory have been reported in the literature. Some implementations support DSM primarily through software, while others provide significant hardware support in

the implementation. We summarize the key aspects of some of these implementations here.

The Cm* multiprocessor [Full78] was probably the first system where several processors shared memory that was physically distributed. In the Cm*, when a processor referenced a location that is not available locally, a message is sent to the remote processor cluster where the memory being addressed is physically connected. However, data from the remote memories were not cached locally. While the absence of caching avoided the coherency problem, the remote accesses resulted in poor performance. More recently, several research efforts have investigated DSM as a vehicle for scalable high-performance.

Kai Li [Li86b] proposed the implementation of shared virtual memory on a loosely-coupled network of workstations. In this shared virtual memory, the virtual memory is shared by all workstations and the physical memory in each workstation serves as a cache of this virtual memory. A prototype shared virtual memory system, Ivy [Li88], was implemented, in software, on a network of Apollo workstations under the Aegis operating system. Coherence is maintained in Ivy using directory-based coherence protocols with invalidation. The prototype was used to investigate three directory-based protocols — with centralized managers, with distributed fixed managers, and with distributed dynamic managers. Memory is mapped and maintained coherent in Ivy at the granularity of 1 K byte pages. Sequential consistency is enforced by implementing atomic writes. Memory and process management are both handled in software. Process synchronization is achieved using the underlying eventcount mechanism in the Aegis operating system.

Clouds [Rama89] uses a distributed shared memory abstraction to develop an object-based distributed operating system. The distributed shared memory constitutes the virtual address space of all objects in the system. Each thread of computation in the object-based system progresses as a series of object invocations, each invocation executing in the virtual address space of the object invoked. The global shared memory abstraction provides a convenient mechanism to map the object's address into the address space of the executing thread. Clouds implements the distributed shared memory abstraction in software. The coherence protocol in Clouds differs from coherency protocols used in other DSM systems in two key aspects. First, the coherence protocol does not satisfy write requests by invalidating (or updating) read copies. Instead, the writing thread is suspended until threads with read copies voluntarily discard their copies. The Clouds kernel forces the discard of memory segments that are read-write shared when a critical section is exited. It also uses a deadlock detection algorithm to detect deadlocks due to undiscarded memory segments. Second, the synchronization mechanism is unified with the coherence scheme by transferring a memory segment only when the associated semaphore operation succeeds. The Clouds system also permits memory segments to be acquired without enforcing strong memory coherence.

Both the Paradigm [Cher91] and the DASH [Leno90, Ghar90, Leno93] projects implement DSM on a hierarchical system organized as a set of processor clusters interconnected by a switching network. Each cluster comprises several processors that communicate over a shared bus. The virtual memory in Paradigm is file-based and allows replication of files. The file copies are maintained coherent by software using operating system supported remote procedure calls. At

a lower level, Paradigm also supports several levels of caching which are maintained coherent by a directory-based protocol implemented in hardware.

The DASH multiprocessor implements DSM in hardware. The 64-processor prototype connects 16 4-processor bus-based clusters through a 2-D mesh. Coherence is maintained at the cache-line size of 16 bytes, using a snoop-based coherence protocol within each cluster, and using a distributed directory-based protocol across clusters [Leno90]. The directory for any block is maintained in a fixed *home node* of the block. The home node maintains a full-map copy directory (a bit-map with one bit for every node in the system) as well as storage for the block. The Dash system implements a weak model of consistency called release consistency [Ghar90]. This model allows memory accesses of a processor to complete out of order with respect to another processor. The ordering of accesses is only preserved before ‘‘release’’ synchronization operations, and other explicit ordering operations (‘‘fence’’). These operations are visible to the hardware, and the hardware blocks the processor for pending accesses only at such operations.

The Data Diffusion Machine (DDM [Hage92]) implements DSM on a system with a hierarchy of buses through extensive hardware support. In this system, the processor-memory nodes are connected to the lowest level buses. Each processor’s memory is organized as a cache which can dynamically store different sections of the address space. The higher level buses contain only directory information. The directory at each hierarchy tracks information about all hierarchies directly below it. Thus, the directory is distributed through the hierarchy. Coherence is maintained at the cache-line size of 16 bytes, using snoop-based protocols in each bus. A hierarchical search algorithm is used to locate any data item, since no

specific manager is assigned for any item. A similar architecture, based on a hierarchy of buses, is also proposed in [Wils87]

KSR1 [Burk92] is a commercial machine built by Kendall Square Research implementing DSM in hardware. KSR1 is similar to the DDM [Hage92] in that the processor's memories are organized as a cache. KSR1 is organized as a hierarchy of rings with the processors connected to the lowest level ring. The KSR1 manages its memory hierarchically by mapping and allocating memory in 16 KB pages, and allowing data to be shared and maintained coherent in 128 byte sub-pages. Unlike the DDM, each page in the KSR1 is associated with an home node, which maintains storage for the page. A hierarchical search algorithm is used to locate any sub-page, as in DDM.

The Alewife [Agar95] architecture supports DSM on a machine with up to 512 nodes are interconnected in a 2-D mesh. A 32-node prototype of this architecture has been constructed. Alewife implements DSM primarily in hardware with some software assist. The architecture uses a distributed directory-based coherence protocol to maintain each 16-byte memory line coherent. Alewife is similar to DASH [Leno93] in that each memory block is assigned a *home node* which maintains the directory and storage for the block. A distinguishing feature of Alewife is that the copy directory is extended in software [Chai91]. The software-extended scheme LimitLESS effectively implements a full-map copy directory by maintaining the copy directory in software when the number of copies exceeds the small number of entries (two to five) in the hardware copy directory. Alewife implements the sequential consistency model. The hardware provides support for fine-grain synchronization using full/empty tag bits with each memory

word, as well as for passing messages between processes. Multithreading and software prefetching are used to hide the latency of remote accesses.

The FLASH [Kusk94] architecture being designed at Stanford builds on their earlier DASH [Leno93] architecture. The integration of support for message passing is a key additional goal for the FLASH architecture. FLASH uses a 2-D mesh interconnection of nodes (not clusters). The DSM architecture in FLASH retains most of the features from DASH. A key difference is the use of dynamic pointer allocation in the copy directory storage. Unlike DASH, FLASH does not maintain the location of copies in each directory entry. Rather, a common store is used to maintain all the copy directories. FLASH also differs slightly from DASH in the low-level implementation of the coherence protocol.

Chapter Three
Hierarchical Management of DSM systems

Distributed shared memory has been implemented [Li89b, Leno93, Agar95, Kusk94], using directory-based cache coherence algorithms [Agar88, Leno90]. With single-level distributed shared memory schemes, there is a fixed size block, which is the unit of mapping, the unit of transfer, and, thus, the granularity of sharing. If the block is ‘‘large’’ (several thousand bytes), the scheme may be refered to as a page-level scheme. If the block is ‘‘small’’ (cache block size — tens of bytes), the scheme may be refered to as a block-level scheme. With a page-level scheme [Li89b], the large unit of sharing results in long delays for interprocessor communication even when the object being (logically) transferred is small. In addition, the large unit of mapping results in reduced system performance due to unnecessary data movement caused by ‘‘false sharing’’ [Bolo89]. Hence, page-level DSM schemes do not provide adequate support for fine-grain parallelism.

The difficulties with page-level coherency management motivate the use of block-level schemes. In these schemes, the unit of transfer is a block. Different blocks of a page can be simultaneously mapped to the memories of different nodes. With schemes that support sharing at the fine granularity of cache blocks [Chai90, Leno90] the number of blocks that must be managed is very large. The mapping tables required to manage the large number of blocks leads to unacceptably high storage overhead.

We propose hierarchical management as a solution to the problems associated

with page-level and block-level DSM schemes. Our scheme is based on the assumption, which is validated by execution-driven and trace-driven simulations, that at any one time only a small percentage of the address space of every process is actually shared. With hierarchical management, the mapping tables are partitioned into tables for strictly local, non-shared pages and tables for shared pages. Special fine-grain support is provided only in the tables for shared pages. These special tables make it possible to specify different locations for different blocks in the page. Since the shared pages are few, this reduces storage overhead significantly over providing fine-grain support for all pages. Pages are dynamically identified as private or shared. During normal execution, pages are dynamically moved between the different mapping tables, as the need arises. The proposed scheme introduces the use of *ownership* [Katz85] at both the page level and the block level in order to provide block-level sharing and also support dynamic page migration to reduce message traffic and the size of mapping tables.

The basic system organization and single-level management of DSM are discussed in Section 3.1. A brief review of Li's [Li89b, Li89a] page-level scheme is presented in Section 3.1.1. A simple block-level scheme is presented in Section 3.1.2. The sizes of tables required for block-level management of DSM are computed in Section 3.1.4. The proposed two-level management scheme is described in Section 3.2. The storage overhead of the scheme is evaluated in Section 3.2.3. Other schemes for multi-level management of DSM are discussed in Section 3.3.

3.1. Single-Level Management of DSM

A high-performance node of a message-passing multicomputer consists of: an application processor, local memory, cache memory to speed up access to the local memory, a memory management unit (MMU) that manages translation and protection for the local memory, and a communication coprocessor that handles most of the tasks involved with sending, receiving, and forwarding messages [Atha88, Chow88].

Distributed shared memory can be supported with single granularity of management on a multicomputer with a number of these nodes. When an address requested by the processor is not available locally, the node should send a message to a remote node requesting a copy of the data. When a node receives such a remote request, the granule of data that was addressed should be accessed from local memory and sent to the requesting node. Mapping information must be maintained for each granule of data. Also, the nodes must implement a coherence protocol to maintain each granule of data coherent.

DSM can be supported by relying on the application processor to perform all the necessary operations [Li89b, Bolo89]. However, for maximum system performance, the application processor should not be involved in the ‘‘emulation’’ of shared memory — special-purpose hardware should service remote requests for pages and convert local accesses to non-local pages into request messages to remote nodes. Just as the communication coprocessor handles the various tasks involved in message passing, a dedicated coprocessor, called a memory-to-messages / messages-to-memory transducer (MMT) will provide support for shared memory emulation. The resulting organization of the multicomputer node is

shown in Figure 3.1. The virtual cache satisfies most of the local processor's memory requests. The MMU performs the address translation in the event of a cache miss or a remote request. A translation lookaside buffer (TLB) in the MMU is used to speed up the translation.

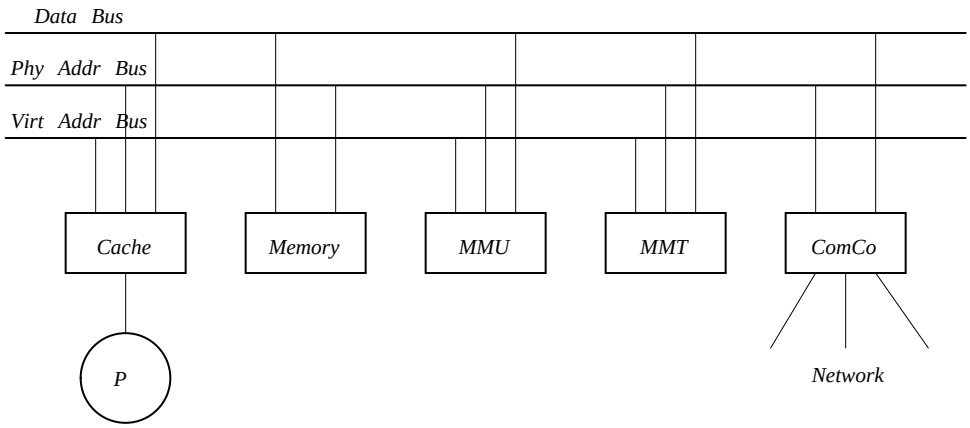


Figure 3.1: A multicomputer node.

Upon request from the MMU, the MMT composes messages and forwards them to the ComCo for transmission to the network. On receipt of a message from the network, the ComCo writes the message either into physical memory or into the MMT. Messages that are responses to requests sent out earlier into the network, are written into a physical memory location specified by the MMU at the time of the request, using DMA. Unsolicited messages such as write-backs, invalidates, remote requests are written to the MMT. The MMT coordinates with the MMU in servicing messages passed on by the ComCo, and writes/reads the physical memory using DMA. The virtually addressed cache snoops on the transactions between the MMU and MMT to maintain its data coherent with those in physical memory.

Each node may be time-shared between multiple processes. A set of processes that share their address space forms a *task*. The system may be executing multiple tasks simultaneously. Unique task identifiers partition the total virtual system space into disjoint task virtual spaces. Processes belonging to different tasks execute in disjoint address spaces and cannot share data.

Single-level management of DSM can be implemented with large granules the size of a ‘‘page’’ or with smaller granules the size of a ‘‘cache block.’’ As discussed earlier, the performance of single-level managed DSM and the storage overhead associated with it are dependent on the size of the granule. We discuss single-level management at the granularity of ‘‘pages’’ and at the granularity of ‘‘cache blocks’’ in the next two sub-sections.

3.1.1. Management at the Granularity of ‘‘Pages’’

Distributed Shared Memory where sharing and coherency are managed at the single granularity of *pages* (thousands of bytes) can be implemented using the normal page fault mechanism of the MMU [Li89b, Bolo89]. Li [Li86a, Li89b] has implemented coherent shared virtual memory on networked workstations and on a multicomputer [Li89a]. The scheme is directory-based [Agar88] where, with the most promising of the proposed alternatives, the directory is distributed among the nodes. Nodes must obtain exclusive ownership of a page before modifying it, invalidating all other copies of the page. Multiple read-only copies can be distributed throughout the system. With the ‘‘distributed fixed manager algorithm’’ each virtual page is initially managed by a *default manager* whose identity is determined by a simple function of the virtual page number.

3.1.2. Management at the Granularity of ‘‘Cache Blocks’’

As discussed earlier, the use of pages as the unit of mapping and the unit of transfer can result in low performance due to the long latencies and the bandwidth requirements for transferring pages and due to false sharing. A simple solution to this problem is to use the same basic algorithms with much smaller pages, henceforth referred to as *blocks*, as the unit of mapping, transfer, and ownership [Chai90, Leno93, Kusk94]. Such a scheme requires address translation to map any virtual block number to a local block frame, a remote node, or secondary storage. Unfortunately, in a large multicomputer, the total number of blocks in all the virtual address spaces of all the tasks in the system will be too large for address translation based on conventional page tables (block tables).

The above problem can be solved by using an *inverted table* [Chan88] for translating virtual addresses to local physical addresses. The inverted block table on each node, called the *present block table* (PBT), contains one entry for each block frame in the local physical memory (Figure 3.2). Hence, the size of the PBT on each node is proportional to the size of the physical memory, rather than virtual memory, on the node. Hashing is used to provide fast associative lookups in the PBT [Chan88]. If a local or remote memory access does not ‘‘hit’’ in the PBT, the default owner (manager) of the block is computed and a remote request for the block is sent to that processor. The default owner of a block may refer to a Migrated Block Table (MBT) to determine whether the block has migrated to another processor or if it is on disk.

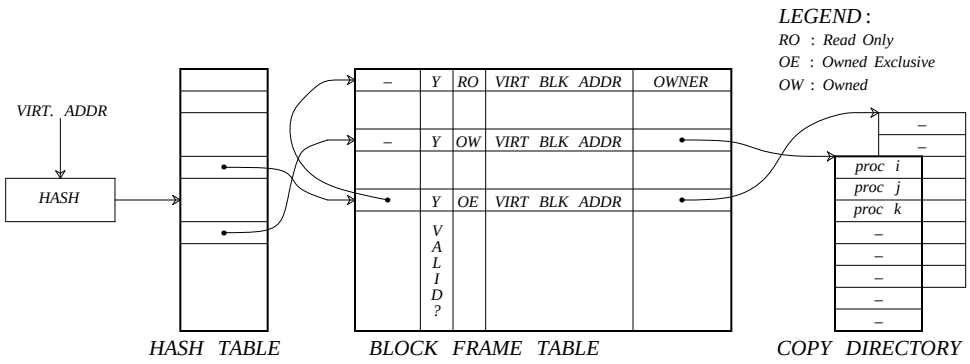


Figure 3.2: Address translation tables for the block-level (one-level) scheme.

3.1.3. Limited Copy Directories

If a block is in read-only state, its PBT entry specifies the block owner. If the block is owned non-exclusively, the same field in the PBT is used to point to the *copy directory*, that contains the identities of the processors that possess read-only copies of that block [Li89b]. This information is used for invalidating all read copies prior to any write to the block. A very large copy directory may be needed to accommodate blocks that are read shared by many processors. There are several approaches to limiting the size of the copy directory. With broadcast protocols, if the number of read copies exceeds some fixed bound, a system wide broadcast is required to invalidate a block [Agar88]. The disadvantage of this approach is that many multicomputer interconnection networks do not provide efficient support for broadcasts. An alternative to broadcast protocols is to invalidate one of the read-only copies of the block whenever a new read request is received by the block owner and the size of the copy directory has already reached its limit. With such an *eviction* policy [Chai90], upon receipt of a read request that would ‘‘overflow’’ the copy directory, the block owner picks a victim from the copy directory,

invalidates it, and uses the freed slot in the copy directory for the new request.

The performance impact of limiting the size of the copy directory has been evaluated based on the access-burst model [Dubo88]. Critical section accesses and the locality of references in shared blocks are modeled by assuming that accesses by one processor to a shared writable block are done in independent uninterrupted bursts called access bursts. An access burst can modify the block with a fixed write probability w. If a copy directory overflow requires a block to be invalidated, the performance degradation due to the limited number of entries in the copy directory can be measured as the increase in the frequency of invalidates over an infinite copy directory. For an unlimited copy directory, the average number of access bursts before an invalidate is $\sum_{i=1}^{i=\infty} i(1-w)^i w$, which reduces to $(1 - w)/w$.

With the limited copy directory, invalidations can be caused by write access bursts and by copy directory overflows. To derive an expression for the average number of access bursts before an invalidate for the limited copy directory, we partition the problem into three cases:

1) when the number of access bursts before invalidate is *less* than the maximum number of entries in the copy directory,

2) when the number of access bursts before invalidate is *greater than or equal* to the maximum number of entries in the copy directory and the invalidate is caused by a write access burst, and

3) when the number of access bursts before invalidate is *greater than or equal* to the maximum number of entries in the copy directory and the invalidate is

caused by a copy directory overflow.

In case 1, the invalidate is only due to a write access burst. Hence, the contribution of case 1 to the average number of access bursts before invalidate is $\sum_{i=1}^{i=v-1} i(1-w)^i w$ where v is the maximum number of entries in the copy directory. The invalidate in case 2 is also caused only by a write access burst. However, since the number of access bursts in case 2 is greater than the maximum number of entries in the copy directory, this case is possible only when utmost v processors participate in the access bursts, irrespective of the number of access bursts. The contribution of case 2 to the average number of access bursts before invalidate can be expressed as

$$\sum_{i=v}^{i=\infty} i(1-w)^i \left[\sum_{j=1}^{j=v} P(j,i,N) w \right]$$

N is the number of processors sharing the block and $P(p,b,N)$ is the probability that exactly p out of the N processors have copies of the block after b read bursts. The inner summation in this expression is the probability that utmost v processors participate in a sequence of i access bursts. Case 3 accounts for the only other scenario possible — when the invalidate is caused by a copy directory overflow. For this case, exactly v processors must have already participated in the access bursts when a read access burst is initiated by a new processor. The contribution of case 3 to the average number of access bursts before invalidate can be expressed as

$$\sum_{i=v}^{i=\infty} i(1-w)^i P(v,i,N)(1-w)\frac{N-v}{N}$$

$P(v,i,N)$ is the probability that the copy directory is fully occupied after i access

bursts. $(1-w)\frac{N-v}{N}$ is the probability of a read access burst by a processor not in the copy directory.

The average number of access bursts before an invalidate for a limited copy directory B is the sum of the contributions of the three cases. Hence, for N processors in the system, and a write burst probability of w, the average number of access bursts before an invalidate for a copy directory with v entries $B(N, w, v)$ is

$$\sum_{i=1}^{i=v-1} i(1-w)^i w + \sum_{i=v}^{i=\infty} i(1-w)^i \left[\sum_{j=1}^{j=v} P(j,i,N)w + P(v,i,N)(1-w)\frac{N-v}{N} \right]$$

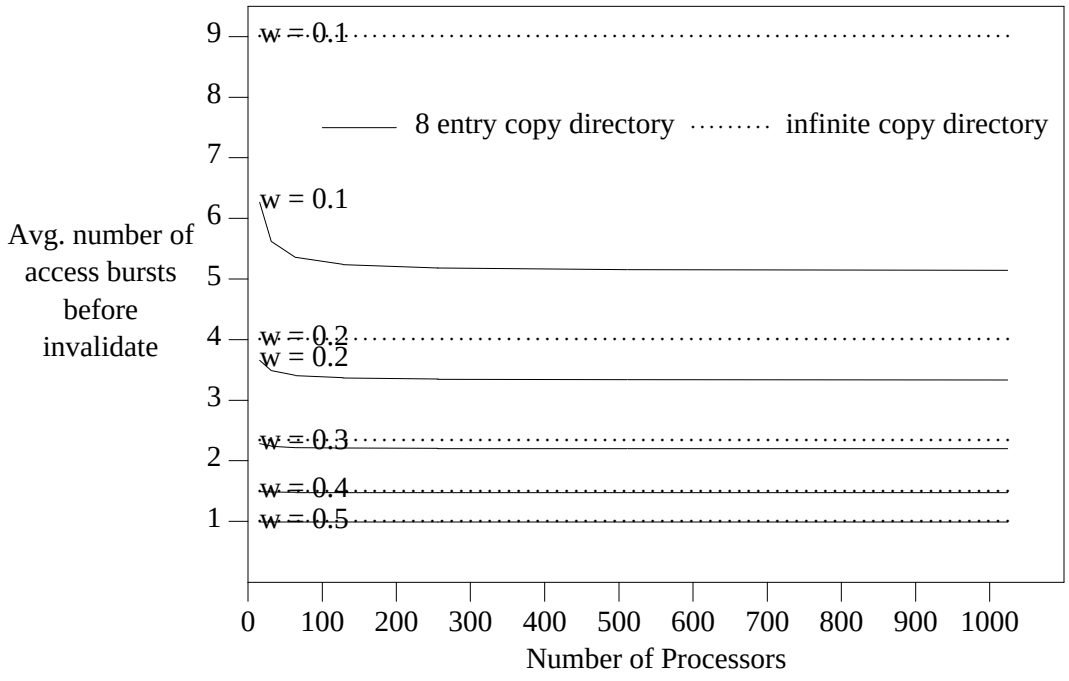


Figure 3.3: Number of Access Bursts prior to invalidate for the infinite and the finite copy directories.

Figure 3.3 shows the relationship between the number of processors and the average number of access bursts before an invalidate for various values of w, for

both and infinite copy directory and for a copy directory limited to eight entries. For write probability ranges observed in practical applications (on the order of 0.3) [Bayl89, Dubo88], the figure indicates that a copy directory with a maximum of eight entries performs nearly as well as the infinite copy directory.

3.1.4. Storage Overhead Example

As discussed above, large mapping tables and directories are a critical problem with single level management at a fine granularity. As an example of the sizes of the mapping tables and directories, consider a multicomputer with 1024 nodes, each with 128 MBytes of physical memory. The virtual address space is 1 TB (2^{40}). A page in this system is 1 KB and a block 32 bytes. The PBT on each node has 4 M entries, each about 82 bits wide. Hence, the PBT requires 40 MB of the local memory. With 8 entries in the copy directory of each of the 4 M blocks in the PBT, the total size of the copy directory in one processor would be approximately 40 MB. The MBT does not have to be resident in local memory but its total size can become unmanageable.

3.2. Hierarchical Management of Distributed Shared Memory

With uniform block-level sharing, all ownership and mapping information is at the granularity of blocks. For every block in physical memory, space is allocated to maintain its access rights and copy directory. The hierarchical scheme proposed in this section improves upon the one-level scheme by optimizations based on the following two observations. First, a large portion of physical memory is used exclusively by the local processor for long periods of time. Hence, the

one-level scheme is wasteful in allocating storage space for access rights and copy directory for each block in the local memory. Second, private segments are likely to occur in contiguous chunks much larger than a block. A private page, that would require only one mapping entry in a page level scheme, would need as many entries as the number of blocks per page, in a block level scheme. Thus, the block level scheme also results in an unnecessarily large number of mapping and directory entries for private segments of the memory.

3.2.1. The Two-Level Management Scheme

The hierarchical DSM management scheme distinguishes between private and shared segments (pages) in physical memory and allocates space for coherency state information and mapping only for the shared pages. The private segments are managed at a page level granularity. Pages dynamically change state, on demand, between private and shared status, transparent to the application. To eliminate the overhead that is caused by false sharing, ownership of shared segments is maintained at both the page and block levels.

A private page located on its default owner (manager) does not require storage of any extra mapping information. On a remote request for a block of a privately owned page, the page changes to the shared state. For each shared page in the system, information is maintained regarding two levels of ownership: page level and block level. The processor owning the shared page maintains the block ownership information for all blocks of that page. The semantics of ownership of a block of a shared page are the same as ownership in the one-level scheme. Specifically, the processor owning a block can provide read-only copies, invalidate

read-only copies, or transfer ownership of the block to another processor. If the block owner provides read-only copies of a block, it must maintain the copy directory for that block.

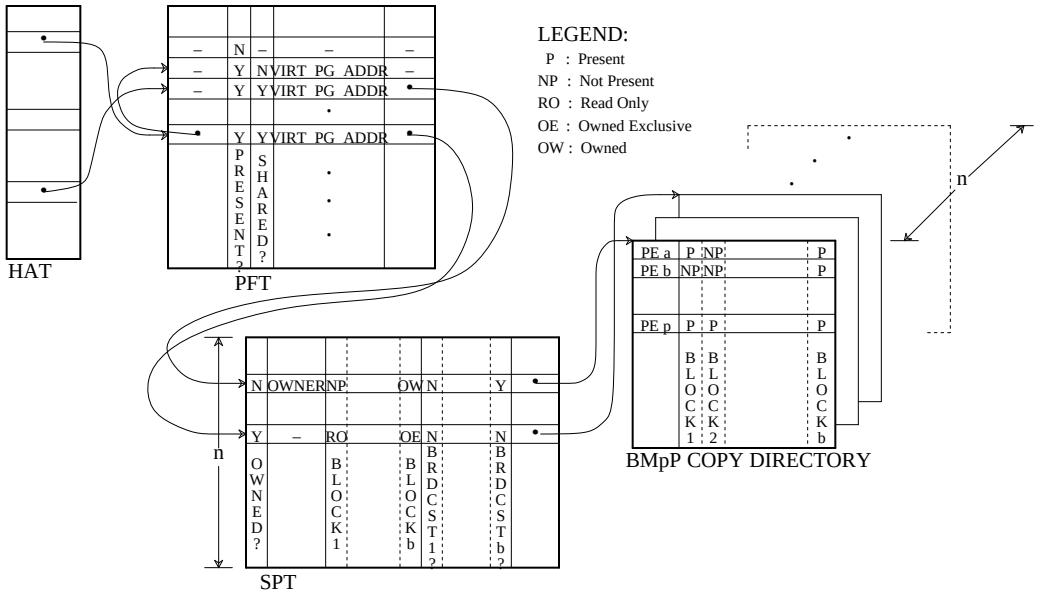


Figure 3.4: Address translation tables for the two-level scheme.

Initially, all pages are private. A remote request for a block of the page changes its state to shared. When a page enters the shared state, all of its blocks are owned by the page owner. Ownership of individual blocks may be transferred to other nodes that need to modify the block. Hence, the owner of a shared page does not necessarily possess copies of all blocks belonging to that page. As with sector mapping in cache memories [Lipt68], space for the entire page must be allocated in the local memory of every processor that has any blocks from that page.

The address translation tables are organized in a two-level hierarchy (Figure 3.4). The first level table indicates which virtual page is mapped in each

page frame of the physical memory and whether the page is shared. On a memory request, the first level table identifies if the page containing the block being requested is present in local memory. If the page is private, the block is guaranteed to be present. If the page is shared, the second level table indicates whether the requested block is present with sufficient access permission. For every owned shared page, the second level table maintains the identity of each block owner. If the page is present and shared, but the block is absent, the request is forwarded to the page owner, or to the block owner if this processor is the page owner. As in the one-level scheme, on a page miss the request is forwarded to the default owner. A Migrated Page Table (MPT) maintains the identity of the current owner of default pages that have been migrated.

The first level table, the *present pages table* (PPT), is organized as an inverted table, consisting of a hash anchor table (HAT) and a page frame table (PFT) [Chan88]. Since the PPT maps pages rather than blocks, it is smaller than the PBT in the one-level scheme. For each page in the physical address space, the PFT maintains the virtual page number, a bit to indicate if the page is shared or private and, if the page is shared, a pointer to a second level table entry. The address translation for a private page need not access the second level table.

The second level table, *shared page table* (SPT) maintains information about shared pages. The SPT entry maintains the access rights to each block in the page as well as a flag indicating whether the page is owned locally. If the page-owned bit is not set, the page owner field holds the identity of the processor owning the page. Two state bits for every block in a shared page indicate the access rights to the block: invalid, read only, owned non-exclusive, and owned exclusive (write

permission) [Katz85]. Each SPT entry includes a field which may contain a pointer to the copy directory. If the page is owned, the copy directory must indicate the owners of each block which is not owned locally. For every block owned non-exclusively, the copy directory indicates all the nodes that have read-only copies of the block. If neither the page nor the block are owned by the local node, messages destined to the block owner are sent to the the *page* owner, specified in the SPT. The page owner may then forward such messages to the block owner. As in the one-level scheme, the copy directory size can be limited by following a policy of eviction or broadcast. The organization of the copy directory is discussed in Subsection 3.2.2. If the broadcast policy is used, each SPT entry includes one *broadcast bit* per block (Figure 3.4).

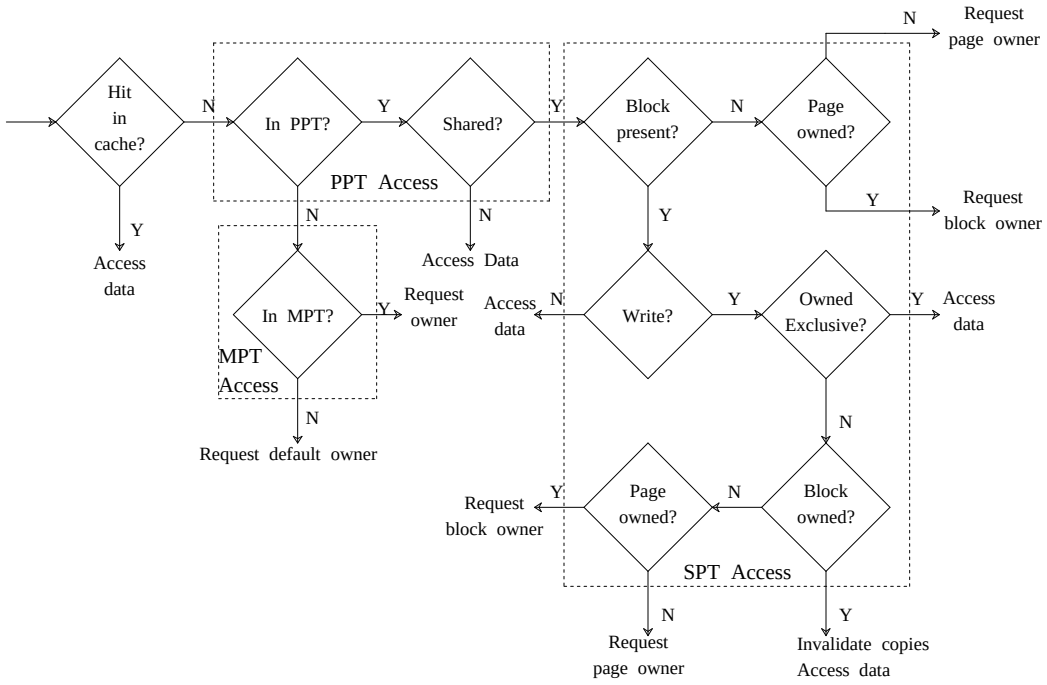


Figure 3.5: Address translation for local requests.

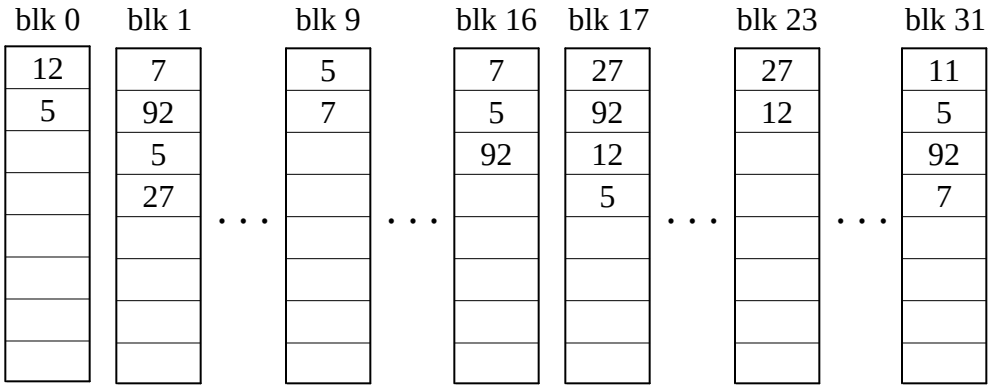
A key to the efficiency of the two-level scheme is that the required number of

entries in the SPT is significantly smaller than the number of entries in the PPT. In many parallel applications the proportion of references to shared storage is less than 15% [Bayl89]. Furthermore, only part of the shared space is actually shared during each ‘‘phase’’ of the program. Hence, an SPT with significantly fewer entries than the PPT should be sufficient to accommodate the ‘‘shared working set’’ of most applications. When the SPT ‘‘overflows,’’ an entry can be freed up by flushing an un-owned page back to its owner or by ‘‘recalling’’ all the blocks of an owned page and making the page private. If page migration is supported, an SPT entry may be freed by migrating an owned page to another node which already owns some of its blocks. We expect an SPT which can accommodate about 10% of the working set of the process, roughly 10% of the entries of the PPT, to be sufficient (see Section 5.1.3 in Chapter 5).

Figure 3.5 shows the sequence of actions of the address translation mechanism for local requests. Remote requests are handled similarly but without the initial lookup in the local cache. The messages exchanged between the nodes to maintain coherency are presented in Section 4.3.3 in Chapter 4.

3.2.2. The Copy Directory

As discussed in Section 3.1.2, in most cases the maximum number of entries in the copy directory for each shared block can be bounded with little performance degradation. With the two level scheme, there may be a need to maintain copy directory entries for every block in every shared page. Using a conventional organization of a copy directory [Agar88], a limited number of processor identifiers is maintained for each block of the page (Section 3.1.2). We call this organization

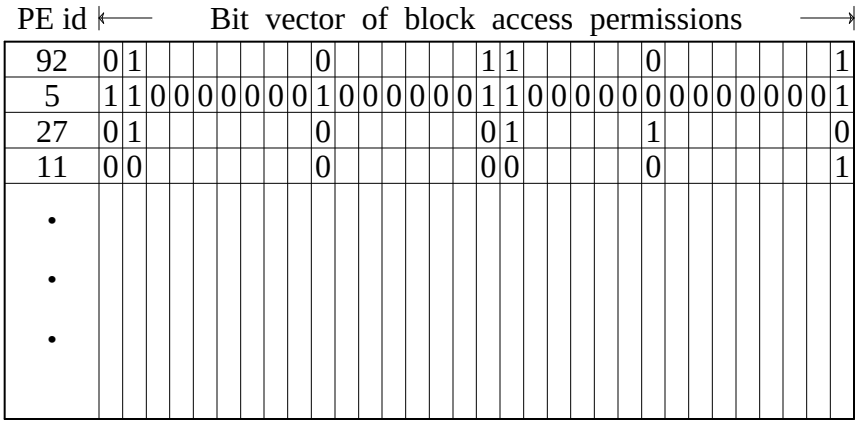


Figures 3.6: Processor List per Block (PLpB) copy directory organization. Each page has 32 blocks, and 8 copy directory entries are maintained for each block.

the *Processor List per Block* (PLpB) organization. This organization is illustrated in Figure 3.6.

For the hierarchical coherency management scheme, we propose a new organization for the copy directory, called the *Block Map per Processor* (BMpP) organization (Figure 3.7). This organization minimizes the required storage for the copy directories based on the assumption that two nodes that share one block of a page are likely to share other blocks from that page. For each shared *page* there is a limited number of BMpP entries. Each entry identifies one remote processor that shares one or more blocks of that page. In each entry there is a bit vector (map) with one bit per block indicating whether the particular remote node has a copy of the corresponding block. The maximum number of entries in the copy directory per shared page is the maximum number of nodes that may share a page at the same time.

With the PLpB organization, if a processor shares several blocks of a page,



Figures 3.7: Block Map per Processor (BMpP) copy directory organization. Each page has 32 blocks.

the processor identifier is stored multiple times for that one page (for example, node 5 in Figure 3.6). Since every entry in the BMpP copy directory keeps track of all the blocks of the page, the processor identifier is stored once so the total storage needed for good performance is expected to be significantly smaller than with the PLpB copy directory (see Section 5.1.4 in Chapter 5). As with the limited PLpB organization, when a BMpP copy directory is full and a new entry is needed, an existing entry can be evicted by recalling all the blocks in the associated remote node. If necessary, it is possible to use a broadcast bit per block (Figure 3.4) with the limited BMpP directories to deal with blocks which are shared by a large number of processors [Agar88].

3.2.3. Storage Overhead of the Two-Level Management Scheme

Coherency Scheme	Size of Mapping Tables per Node			
	PPT/PBT	SPT	Copy Dir.	Total
Conventional One-Level Scheme	40 MB	-	40 MB	80 MB
One-Level scheme (10% shared blocks)	40 MB	-	6.75 MB	46.75 MB
Hierarchical Scheme (10% shared blocks)	1.28 MB	272 KB	2.75 MB	4.25 MB

Table 3.1: Comparison of the storage overhead in the one-level and two-level schemes. The comparison is shown for a system with 1024 nodes, 1 TB of virtual address space, and 128 MB of physical memory per node. The one-level scheme uses 32-byte blocks. The hierarchical scheme uses 1 KB pages and 32-byte blocks.

With the system parameters discussed in Section 3.1.2, the Present Pages Table (PPT) has just 128 K entries as opposed to 4 M entries in the Present Blocks Table (PBT) discussed in Section 3.1.2. As a result, the total size of the PPT is only 1.28 MB. As discussed in Subsection 3.2.1, the number of entries in the second-level Shared Pages Table (SPT), and thus the number of BMpP copy directories can be significantly smaller than the number of PPT entries. With the broadcast bit shown in Figure 3.4, each SPT entry requires 134 bits of storage. Hence, approximately 272 KB will be required for 16 K entries.

With the proposed Block Map per Processor (BMpP) organization of the copy directory, given a fixed memory size and page size, the total storage needed for the copy directory is sensitive to the size of the block. Larger blocks lead to smaller entries (fewer blocks per page), and thus reduce the memory used for copy

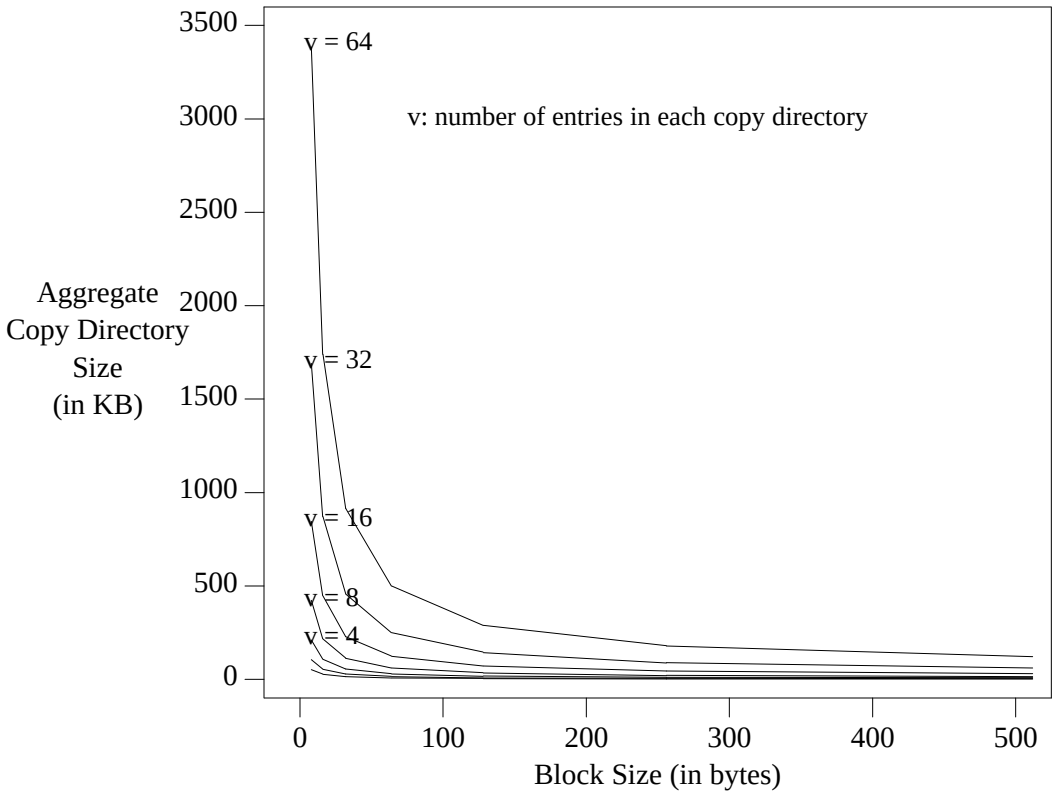


Figure 3.8: Total size of all the copy directories with the BMpP organization on a node versus the granularity of sharing. 1K entries in the SPT. 4 KB pages. 32 MB local memory. The parameter *v* is the number of BMpP copy directory entries per shared page. The total memory required for the copy directories increases rapidly for block sizes below 64 bytes.

directory storage. However, small blocks are needed to minimize false sharing. For the example system, Figure 3.8 shows the size of the copy directory per node versus the block size for different number of entries in the copy directory per shared page. With a block size of 32 bytes and 32 entries per copy directory, the total copy directory size per node is about 2.75 MB.

In the one-level scheme, if all the blocks are allowed to be shared, the total size of memory resident tables — PBT and the PLpB copy directory — is

approximately 80 MB (62.5% of local memory) If 10% of the blocks may be shared, the total size of tables is approximately 46.75 MB (36.5% of local memory) (see Table 3.1). With the two-level scheme, the total size of memory resident tables — PPT, SPT, and the BMpP directory — is about 4.25 MB, consuming less than 4% of the local memory.

3.3. Related Work

Two of the key ideas in our proposed scheme have also been proposed by O'Krafka and Newton [O'Kr90]: 1) managing coherency at two levels of granularity, block and sub-block, and 2) exploiting the fact that not all the blocks are actually shared at each point in time. Their sectored directory scheme [O'Kr90] manages data at two levels — a block and a sub-block, similar to sector-mapping in uniprocessor caches. Ownership is associated with each sub-block. The identity of the owner and a modified bit are maintained for each sub-block. To minimize the directory storage overhead, the full-map copy directory is maintained only for each block (instead of each sub-block, as in our scheme). As a result, the sectored directory scheme does not fully exploit the idea of multi-level management since it requires invalidating all the sub-blocks of a block when only one sub-block needs to be invalidated. This limitation results in poor performance [O'Kr90].

The second scheme proposed in [O'Kr90] avoids maintaining a copy directory for every memory block. Instead, copy directory information is maintained in dedicated associative caches (''tag caches'') only for blocks that are present in processor caches. This reduces directory storage overhead over maintaining directory information for all blocks in memory. As an additional optimization, two

different caches are used to maintain copy directory information: a large cache with a limited copy directory in each entry, and a small cache with full-map entries. A block is initially allocated an entry with a limited copy directory. If the number of copies exceeds the number of copies supported in the limited copy directory, a full-map entry is allocated, and the limited copy directory entry is freed.

The second scheme in[O'Kr90] requires dedicated associative copy directory caches to exploit the fact that not all blocks are shared at each point in time. Through the use of simple pointers in the PPT, we accomplish the same functionality with simpler hardware that should result in faster operation. Since the pages are large, the number of entries in the PPT is relatively small and the storage overhead for the additional pointers is insignificant (see Subsection 3.2.3). There is no discussion in [O'Kr90] of how the two ideas for reducing the storage requirements for mapping tables can be combined in one efficient scheme, as described earlier in this section. In particular, as discussed in Section 3.2.2, the two-level management scheme allows a new efficient copy directory organization to be used.

Gupta et al. [Gupt90] propose using caches of directory information, which they call *sparse directories*, instead of maintaining a directory entry for every memory block. The sparse directory maintains only entries associated with blocks that are present in the caches. This sparse directory approach is similar to our approach of using inverted mapping tables. However, Gupta et al. [Gupt90] maintain copy directory information for each entry in the sparse directory. They do not explore the benefits of dynamically identifying private and shared data. Our

hierarchical scheme identifies dynamically shared data and maintains copy directory information only for shared pages. This refinement leads to additional savings in directory storage overhead.

With both the copy directory organizations discussed in Section 3.2.2, the copy directory is placed at the owner. Since local storage for the copy directory is limited, performance may degrade with such organizations in the presence of heavy sharing in large systems. In the Stanford FLASH multiprocessor [Kusk94] this problem is alleviated using a technique called *dynamic pointer allocation*. In FLASH, copy directory entries for all blocks are allocated from a common bank of copy directory memory. This accommodates the need for large copy directories for heavily shared blocks while not wasting storage for the small copy directories sufficient for lightly shared blocks.

Another solution to the problem of copy directory overflows is to implement the copy directory of a block as a linked list of entries (a *chained directory*) distributed among the processors sharing the block [IEEE93, Gust92, Thap90, Chai90]. The chained directory scales well to larger systems, but incurs additional latency during invalidations in scanning the linked list.

Alewife employs a LimitLESS [Chai91] copy directory organization which handles copy directory overflows in software. Alewife maintains a limited copy directory in hardware. When this limited hardware copy directory overflows, a full-map copy directory is maintained in software. All subsequent requests for this block are serviced by software. Our scheme can also take advantage of such copy directory extensions to improve performance under heavy sharing.

Chapter Four

An Implementation of Hierarchically Managed DSM

The memory local to a node in a DSM system maintains only a part of the entire shared address space. The local memory functions as a cache for the entire distributed address space. When a processor issues an access, the datum referenced by the access is not guaranteed to be resident locally. As described in Section 3.2.1 in Chapter 3, local memory management tables are used to determine whether the access can be completed locally. If it can, the various tables locate the datum in local memory. If the datum is not available locally or its state does not permit the access, a request must be sent, through the interconnection network, to a remote node that can provide the datum with the appropriate access permission. Requests received from remote nodes must be processed to provide the required data. The inter-node interactions involved in moving data to where it is needed are governed by the coherence protocol.

We have investigated hardware implementation of the address translation mechanisms and the coherency protocol required to support hierarchically managed (and one-level managed) DSM. The address translation functionality of the Memory Management Unit (MMU) is extended to manage distributed shared memory. The design of the MMU also integrates support to handle the inter-node interactions required for maintaining coherency. We discuss our design of the MMU in Section 4.1. Designing a distributed directory coherency protocol must address several issues such as deadlocks and race conditions. Section 4.2 discusses

these issues, and our solutions to these issues. Our distributed directory coherence protocol is completely described in Section 4.3. Section 4.4 describes our implementation of hierarchically managed DSM for trace-driven simulation and execution-driven simulation.

4.1. Memory Management Unit design

The Memory Management Unit (MMU) provides most of the hardware support for the DSM implementation. Figure 4.1 shows the block diagram of the MMU. To support address translation for shared accesses, the MMU implements the mapping tables Present Pages Table (PPT), Shared Pages Table (SPT) and the Block Map per Processor (BMpP) copy directory. The MMU also implements the coherence engine which ensures that all requests are serviced according to the coherence protocol.

The MMU services requests from the local processor as well as requests from remote nodes. To satisfy local and remote requests simultaneously, the MMU maintains independent datapaths for local and remote requests. Since both local and remote requests may need to look-up the Present Pages Table (comprised of the Hash Anchor Table, HAT, and the Page Frame Table, PFT) and the Shared Pages Table, the PPT and the SPT are multi-ported. In addition, the access to the Hash Anchor Table (HAT Acc. in Figure 4.1), access to the Page Frame Table (PFT Acc. in the figure) and access to the Shared Pages Table (SPT Acc. in the figure) are performed in pipeline sequence, both for local requests and remote requests. This allows multiple local requests and multiple remote requests to be serviced concurrently in a pipelined manner.

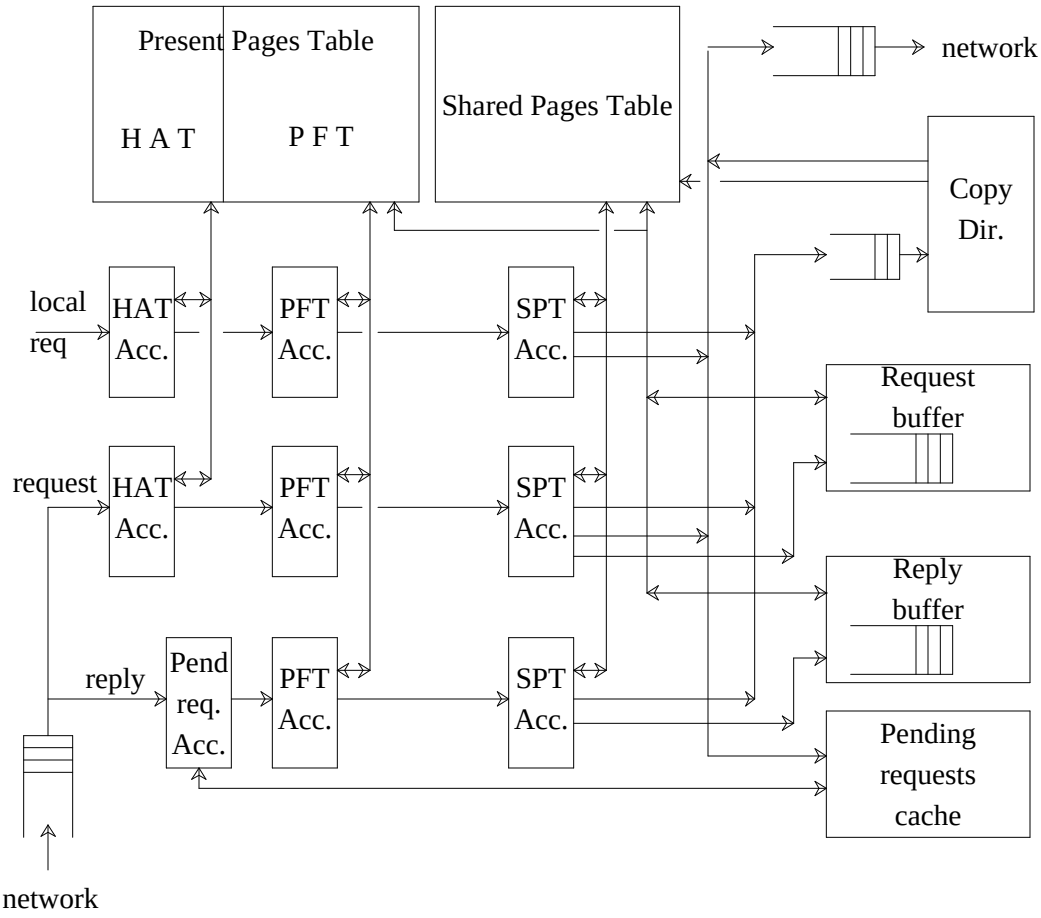


Figure 4.1: Memory Management Unit Design.

Local and remote request servicing may also require copy directory lookup. The copy directory access is internally pipelined. However, unlike the PPT and SPT, only one request can be issued to the copy directory at a time. Since requests that require copy directory access typically also incur costly remote access latencies (messaging), more sophisticated support to parallelize copy directory access may not be beneficial.

Local processor requests and remote requests can generate other network

messages which require replies from other nodes in the system (e.g., invalidation on a write request). Such local and remote requests must be suspended until their replies arrive, to allow the MMU to continue servicing other local and remote requests. In addition, replacement events to free up entries in the PPT, SPT or the copy directory in the local node may also generate network requests which require replies (e.g., invalidation to free up a copy directory entry). The state of local and remote requests that are suspended waiting for remote replies as well as eviction events which are waiting for remote replies are maintained in the Pending Requests Cache (Figure 4.1).

The state maintained in the Pending Requests Cache allows replies received for the messages sent to be serviced with a minimum number of additional mapping table lookups. The state information maintained in the Pending Requests Cache identifies the suspended request. It indicates the type of the request (read, write or local replacement event), the address accessed by the request and the source of the request (local node or identity of remote node). In addition, the entry for the request maintains the page frame number of the page accessed by the request (PPT entry number), the type of message(s) that were sent out, the remote node to which the message was sent, and the number of replies (e.g., number of invalidation acknowledgments) expected. Note that all fields in the entry may not be used in all cases. For example, if many invalidations were sent (to perform a write or a replacement), the field identifying the node to which the message was sent is not used. The field identifying the page frame number may not be used when the entry corresponds to the eviction of a copy directory entry.

When a reply message is received from the network, the Pending Requests

Cache is looked up associatively (Pend req. Acc. in Figure 4.1) by matching with the address field of the reply. If the reply allows the suspended request to proceed, (e.g., all acknowledgements have been received) the MMU removes the suspended request from the Pending Requests Cache and completes servicing it. The MMU processes replies concurrent with local requests and remote requests. Reply servicing is also pipelined in the MMU. If the PPT and SPT need to be updated to service the suspended request, the updates are performed in a pipelined manner.

To service a remote request or a reply to a local request, an entry may need to be allocated in the Shared Pages Table or in the copy directory. If an entry is not available for allocation, service to the request or reply must be suspended until an entry becomes available (mechanisms to free entries will be discussed shortly in Section 4.1.1). The request or reply is queued. As shown in the block diagram (Figure 4.1), replies and requests are queued in independent buffers. When an SPT or copy directory entry becomes free, any request or reply that has been suspended and queued is serviced. The reply queue is prioritized higher than the request queue so that data can be provided to the local processor quickly.

The functionality of the MMU and MMT (Section 3.1) chips is similar to the functionality of the directory controller (DC) and the reply controller (RC) boards in the DASH prototype [Leno93], of the CMMU chip in the Alewife architecture [Agar95], and of the MAGIC chip in the Stanford FLASH multiprocessor [Kusk94].

4.1.1. Present Pages Table and Shared Pages Table

The MMU refers to mapping and coherency information when the local processor issues a memory access due to a miss in its cache and when a request is received from a remote node. As discussed in Section 3.2.3 of Chapter 3, the total size of the mapping tables (PPT and SPT) is on the order of a few hundred thousand bytes. Hence, it may be possible to store them on chip with the MMU in order to speed up accesses. The tables can also be maintained off-chip in external memory. The physical organization of the PPT and SPT is straightforward [Chan88] and similar to the logical structure shown in Section 3.2.1 (Figure 3.4). The MMU can exploit locality by caching this information in a Translation Lookaside Buffer (TLB). As in conventional systems, the TLB provides fast translation of virtual to physical page numbers. Here, it also provides the full SPT entry for shared pages. With two read ports and one write port, the TLB can service local and remote requests concurrently. We have not investigated the impact of the size and organization of the TLB, and do not show it in the block diagram in Figure 4.1.

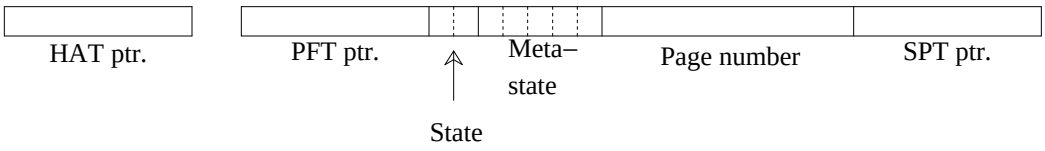


Figure 4.2: Present Pages Table (PPT) entry format

The PPT entry format is shown in Figure 4.2, which is similar to the logical structure shown in Figure 3.4. Each entry maintains two pointer fields used in the inverted table organization, one part of the Hash Anchor Table (HAT) and the

other part of the Page Frame Table (PFT). In the inverted table organization, each table entry is the start (i.e., anchor) of an hash bucket, and the entry itself may dynamically be part of another hash bucket chain. The HAT pointer field identifies the first PPT entry in the hash bucket anchored at this entry. The PFT pointer field points to the next PPT entry in the hash bucket chain to which this entry belongs. When a PPT entry is not allocated in any hash bucket, the PFT pointer is re-used to maintain a list of free PPT entries. The usage of each PPT entry is indicated by the state bits maintained in the entry. Two bits of entry state identify if the entry is invalid, valid, or if the entry is allocated pending data from a read request or write request. When a page entry is valid, additional information about the page is maintained by a few meta-state bits to assist in situations such as page replacement. Two of the five meta-state bits identify if a valid page is private or shared, and if the page is clean or dirty. The other three meta-state bits identify whether the page is pinned from being replaced, whether the page is being replaced to free up its PPT entry and whether the page is being replaced to free up its SPT entry. If the entry is valid, the page number field identifies the virtual page number of the page mapped to this entry. If the page is shared, a pointer to this page's entry in the second level Shared Pages Table (SPT) is maintained.

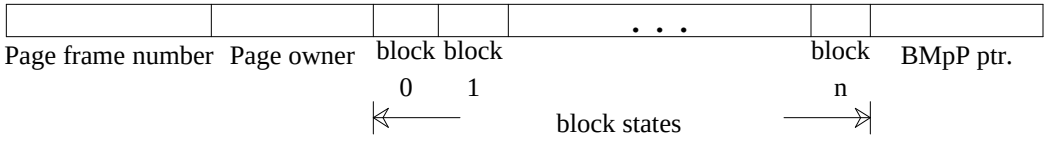


Figure 4.3: Shared Pages Table (SPT) entry format

Figure 4.3 shows the SPT entry format. Each entry identifies the owner of the

shared page. It also identifies the physical page frame to which the page is mapped. When the SPT overflows and an SPT entry is evicted, the state of the page in its PPT entry must be appropriately modified. The page frame number maintained in the SPT entry serves as a back pointer to the PPT, allowing the identification of the PPT entry used by the page. The SPT entry also maintains the state of each block of the page, and potentially whether the blocks are dirty or clean. The state of each block identifies the type of accesses (read or write) that are permitted for the local processor. If a request has been issued to the network to service an access to the block, the block state also identifies the access that is pending on the block. We discuss the actual states of blocks along with the discussion of the coherence protocol in Section 4.3.4. The SPT entry also maintains a pointer to its BMpP copy directory. If the SPT entry is not currently used, this pointer is used to maintain a chain of available SPT entries.

An entry is allocated in the PPT when an access by the local processor misses in the PPT. A read or write request is sent out to the network and the PPT entry state is updated to indicate that read or write access has been requested. No entry is allocated in the SPT at the time the request is sent out. Rather, an SPT entry is allocated when the reply arrives from the network, if the reply indicates that the page is shared. At that time, the state of the PPT entry, the SPT pointer field in the PPT entry, and the state of the block in the SPT entry are all suitably modified. The PPT entry state bits are set to valid, and the page is marked as shared in the meta-state bit of the PPT entry. The SPT pointer field of the PPT entry is set to point to the SPT entry allocated for the page. The state of the block in the SPT entry is updated to identify the access permission granted by the reply message.

An SPT entry is also allocated when a request from a remote node makes a page which is currently private, shared.

If no entry is available in the PPT when a local processor access requires the allocation of a new entry, the local processor is blocked. The processor is unblocked when a entry becomes available due to invalidation or replacement. If no free SPT entry is available when required by a reply from the network (to a request previously issued from this node) or when a request from a remote node requires one, the reply or the request is buffered. The buffered request or reply is serviced when an SPT entry is freed by eviction or invalidation. To minimize blocking and queuing, we employ premature eviction, with both the PPT and the SPT, to maintain some free entries. When the usage of these tables goes above a threshold, we pick a victim and initiate its replacement.

When a PPT entry must be freed, the victim page must be migrated to another node or disk. If page ownership is not allowed to change, owned pages can only be evicted to disk. Since this incurs significant performance impact, it is preferable to evict pages that are not owned locally. However, if the PPT entry of an owned page must be freed, it is preferable to evict private pages rather than shared pages. When the victim page is a shared page, the corresponding SPT entry must also be freed which additionally requires interactions with other nodes.

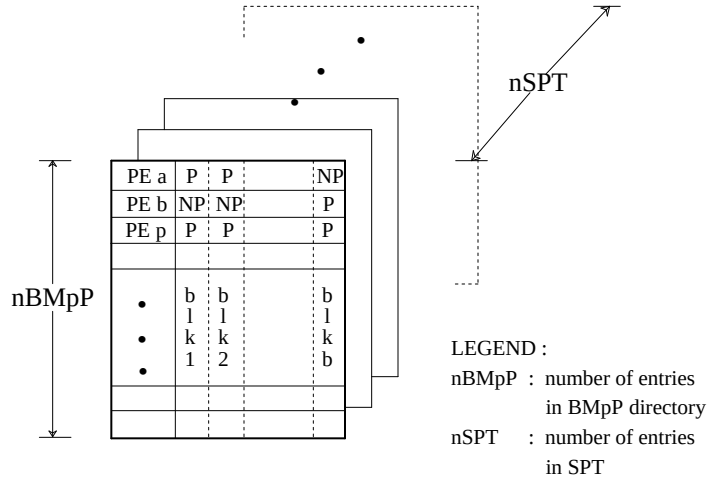
It may be necessary to evict an SPT entry due to the eviction of the corresponding local PPT entry or in order to allocate the SPT entry to a different page. As described previously, if the ownership of the page cannot change, it is preferable to evict a page that is not owned locally to avoid disk access. The SPT entry of a page not owned by the local node, can be freed by ‘‘pushing’’ away the

page. All blocks of the page that are owned locally can be written back to the page owner, and all blocks that are held read-only can be dropped. If the SPT entry of a page owned locally must be freed, all blocks of the page must be invalidated from all other nodes in the system. All blocks of the page can be ‘‘pulled’’ from the remote nodes, by commanding all nodes that own blocks of the page to write-back the blocks.

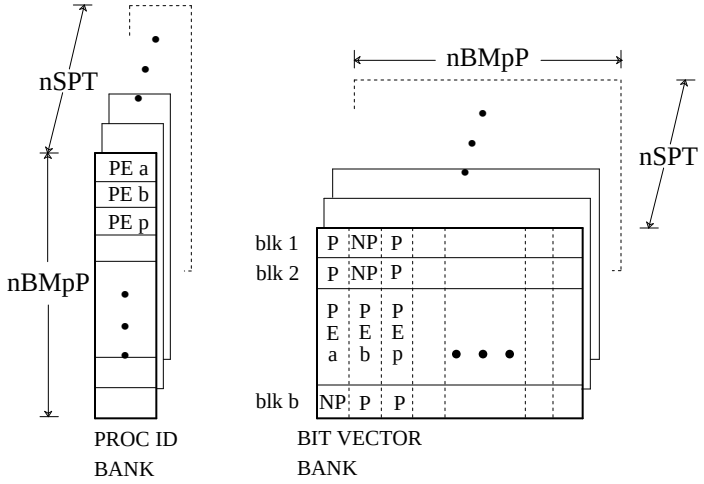
4.1.2. Block Map per Processor Copy Directory

A lookup in the Block Map per Processor (BMpP) copy directory may be performed in order to respond to a remote request or as part of an access initiated by the local MMU. A remote request may request a copy of a locally-owned block or send a notification that it has deleted its copy of a locally owned block. The local MMU may require a copy directory lookup in order to find the locations of all the copies of a locally-owned block or in order to find the owner of a block that is not locally-owned. With the first type of copy directory access, the directory must be searched sequentially based on the processor identifier. Once a match is found (or a new entry is allocated), the corresponding bit vector is updated. For the second type of access, the directory is accessed directly using the block offset within the page. In this case the desired result is a column bit vector from the table shown in Figure 3.4, i.e., from each directory entry, the one bit that corresponds to the specified block. Once the bit vector is read, it is used to obtain the processor identifiers corresponding to ‘‘1’’ entries.

The physical organization of the BMpP copy directory can be geared to the above access requirements by organizing it as two separate banks — one



a) Logical organization



b) Physical organization

Figure 4.4: BMpP copy directory organization.

maintaining the processor identifier field of all copy directory entries and the other maintaining the bit vectors (Figure 4.4-b). Both types of copy directory accesses discussed earlier require access to the processor identifier and the bit vector banks. Since these banks are physically independent, the banks can be concurrently accessed. This also allows copy directory accesses to be pipelined.

The bit vector bank is also transposed with respect to the logical structure of the copy directory (Figure 4.4-b versus Figure 4.4-a). Each word in this bank (or multiple words, depending on the word length of the bank and the number of entries in the copy directory) maintains the set of bits associated with a *specific* block of a page (a row in Figure 4.4-b and a column in Figure 3.4). Unlike the logical organization, this organization maintains the bit map associated with any block in contiguous bit positions. Hence, copy directory accesses to locate copies of the block can be performed efficiently by reading fewer words than if the logical structure was used.

Copy directory accesses can be further speeded up by special hardware support such as associative lookup of the processor identifier bank and pipelining in the bit vector bank access. However, since a memory reference that involves a copy directory access must also involve network communication, the copy directory access latency may be overshadowed by the network communication latency. Thus, additional hardware support to speed up copy directory access may not be justified.

If a fixed number of BMpP entries are allocated per shared page, the copy directory may not be able to simultaneously accommodate all the processors that are accessing a particular block. As with other limited copy directory schemes [Agar88], contention for BMpP entries may be resolved by eviction of occupied entries. Where such a policy is inefficient (e.g., for read-only shared pages), an unlimited number of simultaneous copies may be supported by requiring system-wide broadcasts to invalidate all the copies. A *broadcast flag* in the SPT entry can be used to indicate that broadcasts are required.

Techniques such as the dynamic pointer allocation [Kusk94] scheme used in the Stanford FLASH multiprocessor can also be employed here. BMpP entries can be allocated in batches from a memory bank of BMpP entries shared by all pages. As implemented in [Kusk94], BMpP entries can be chained, and BMpP entries that are freed can be returned to the free list in the memory bank. We did not investigate the implementation of the dynamic pointer allocation scheme for BMpP copy directories.

The Alewife machine designed at MIT [Chai91, Agar95] allows the copy directory to overflow into software. When the number of copies of a block goes over the static bound of five, the copy directory is maintained by software, and all remote requests to the block are handled by software. Such software support schemes are also applicable for the BMpP copy directory scheme, but have not been investigated.

4.2. Distributed Directory Coherence Protocol Design Issues

We have investigated the implementation of a distributed directory coherence protocol for maintaining the node memories coherent. Following most current DSM implementations [Leno90, Kusk94, Agar95], we chose an invalidation-based ownership protocol. Nodes must obtain exclusive ownership of the block before modifying it, by invalidating all other copies of the block. Multiple read-only copies can be distributed throughout the system.

As we discussed earlier in Section 2.2, the design of distributed directory coherence protocols is more complicated than the design of snoopy protocols. In a shared bus multiprocessor using snoopy coherence protocol, the bus can be used by

only one processor at a time, while all processors can snoop on the bus activity. As a result, the shared bus serves as a mutual exclusion lock ordering accesses by different processors.

In the multicomputer system, multiple processors can simultaneously send requests out to the network. These requests can arrive out of order at the destination if the network does not guarantee FIFO delivery of messages between any pair of nodes. Even when the network guarantees pair-wise FIFO delivery of messages, such race conditions can occur if the coherence protocol requires requests to be forwarded by intermediate nodes (such as when a request is forwarded by the page owner to the block owner).

Distributed directory coherence protocol design must also address the potential for deadlock. Since the directory information is distributed, accesses can require several request and reply messages to be exchanged. Hence, accesses are not inherently atomic and they progress by allocating distributed directory resources incrementally. Directory resource requirements of conflicting accesses can, therefore, lead to a cycle of dependencies resulting in deadlock.

In the rest of this section, we discuss the issues in implementing a distributed directory coherence protocol and our solutions to these issues. In our discussion, we refer to specific states, and specific message types used in the coherence protocol. While the semantics of the states and the message types used here will be clear from the context of the discussion, they are formally defined in Sections 4.3.2 and 4.3.3.

4.2.1. Deadlock

The potential for deadlocks exists in any system where there are multiple operations in progress concurrently and where these operations require use of shared resources. Specifically, deadlocks may occur when operations can obtain exclusive access to a subset of the resources they require for completion and maintain this exclusive access until completion. The problem occurs when the additional resources required for completion are held by another operation which, in turn, is waiting for shared resources to be freed. The design of the system must prevent the possibility of a *deadlock* where multiple operations are unable to progress due to a cycle of resource dependencies [Silb94].

In the DSM system, requests serviced by the coherence protocol compete for logical and physical resources. Requests compete for logical resources such as read access permission and exclusive access permission. For example, a write request can compete for exclusive access to a block with other read requests and write requests to the same block. To complete the write request, the node issuing the request must invalidate the access permissions of all other nodes in the system. Since these logical read access permissions are distributed through the system, they can be invalidated only incrementally. When concurrent writes to the same block compete for control of all read access permissions (to invalidate them), a cycle of dependencies can manifest leading to deadlock.

Similar cycle of dependencies can also occur when requests compete for physical resources such as table entries and buffer space. When a request is generated in a node, storage may be allocated on that node to maintain information associated with that request. When this protocol request is received at the

destination node, other child protocol requests may need to be generated (e.g., a writeback request may initiate generation of invalidate messages). This requires allocation of storage in the destination node also. The parent request progresses by incrementally acquiring storage in several nodes, and its completion depends on the availability of sufficient storage in all these nodes. Competition for the distributed storage resources amongst concurrent coherence protocol requests can also lead to cyclic dependencies and deadlock.

We integrate several mechanisms into our coherence protocol to address the problem of deadlock in DSM systems. We prioritize invalidate commands and ensure that invalidate messages are always consumed. As a result, invalidate messages do not participate in any deadlock. This is similar to the approach in the DASH [Leno90] and FLASH [Kusk94] directory coherence protocols where reply messages are always consumed. Our coherence protocol also prioritizes the owner over other nodes. When requests require conflicting access to the same block, all nodes uniformly grant the request issued by the owner. In addition, our coherence protocol uses negative acknowledgements during contention or resource scarcity to avoid deadlocks. When a request cannot be serviced, a negative acknowledgement is sent to the requesting node and resources allocated for the request are freed. Our mechanisms for handling deadlock may not be sufficient to handle all deadlock situations that are possible in DSM systems. When new deadlock situations arise in a full-scale implementation, additional mechanisms may be necessary to handle them. In the rest of this subsection, we discuss two examples illustrating the occurrence of deadlock in DSM systems and our method of handling them.

Consider the example shown in Figure 4.5. Two processors A and B are

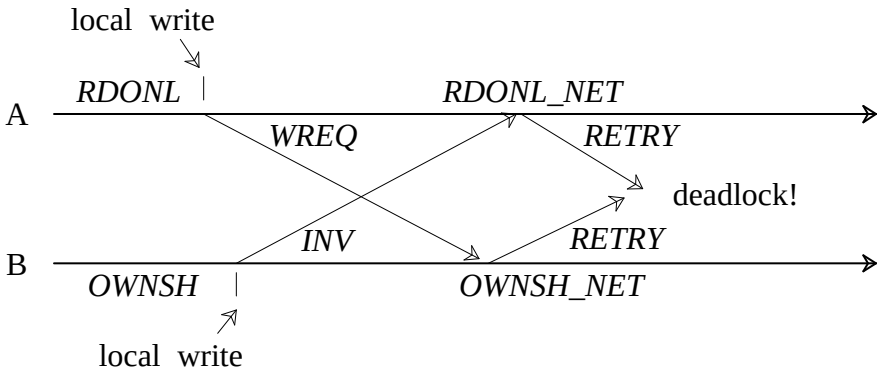


Figure 4.5: Deadlock due to competing writes. Processor *A* does not service *B*'s invalidate request (inv) due to its pending write, and processor *B* does not service *A*'s write request (wreq) due to its pending write resulting in deadlock.

attempting to write to the same block. Initially, processor A has a read access permission to the block, and processor B is the owner of the block. With the invalidation-based ownership protocol, exclusive access permission and ownership must be obtained by invalidating all other read copies before writing the block. In this case, processor A issues a write request (*WREQ*) to the owner B to obtain exclusive ownership for its write. Simultaneously, processor B sends an invalidate message (*INV*) to processor A to proceed with its own write. Suppose each processor refused to service the other processor's request (using a *RETRY* message) due to its own pending request. In this case, each processor retains its access permission and must wait to acquire control of the other's access permission. This leads to a cycles of dependencies. Hence, neither write access would progress, and the accesses would be deadlocked.

To avoid this deadlock, one of the two competing accesses should be allowed to progress. This can be accomplished by giving higher priority to the current

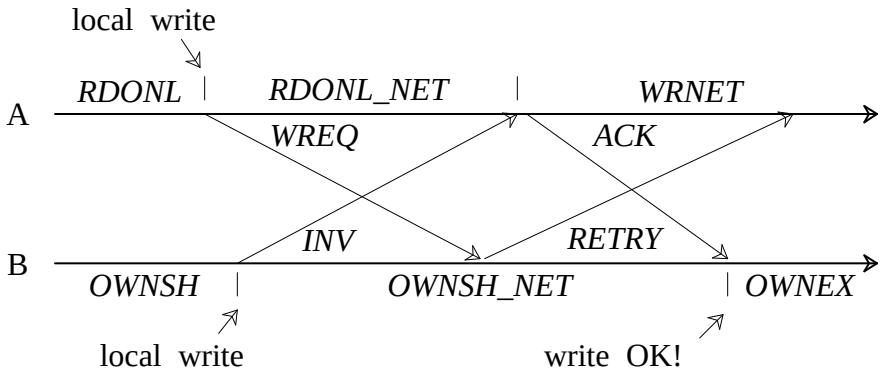


Figure 4.6: Deadlock avoided by ‘‘sink’’ing invalidate. Deadlock is broken by requiring that invalidate messages (*INV*) cannot be denied service. Processor *B*’s write access will succeed first. Subsequently, processor *A*’s write will be retried and will complete successfully.

owner of the block. Specifically, in our implementation, *invalidate* request messages from the owner are handled and ‘‘obeyed’’ without delay. The modified handling of the example is shown in Figure 4.6. Processor A invalidates its read copy and sends an acknowledgement to the owner B, while still maintaining its pending write. Processor B will, hence, complete its write access, and it will subsequently service processor A’s write request.

Dependencies may also be created between processors due to resource constraints, which may lead to deadlocks [Leno90]. For example, consider the following scenario summarized in Table 4.1. Processor B owns page *p1* containing block *b1*. No block from *p1* is present in processor A. Processor A owns page *p2* containing block *b2*. Processor B owns block *b2*. Processor A possesses a read-only copy of *b2*. Processor A executes a load from block *b1*. As a result a read only copy of block *b1* is requested from processor B. Processor A receives a read copy of block *b1*, which it buffers since no free SPT entry is available. Page *p2* is

picked for eviction to free up an SPT entry. To evict this page a message is sent to processor B, which owns block *b2* of the page, requesting it to write-back block *b2*. Concurrently, processor B issues a write access for the block *b1*, and sends an invalidate message to processor A. Processor B records its pending write to block *b1* in its pending request cache in order to collect invalidation acknowledgements. To service the write-back request from A associated with the SPT eviction, processor B must invalidate copies of block *b2*. This invalidation can proceed only if there is space in the pending request cache of processor B since the pending write-back of block *b2* must be recorded in the pending request cache of processor B (again, to collect invalidation acknowledgements). Table 4.1 lists the sequence of events at each node, the action initiated by the event and the dependency created by the action.

If the invalidate for block *b1* in processor A cannot be processed until its SPT eviction is completed and the freed SPT entry is allocated to the buffered block *b1*, a cycle of dependencies can manifest as follows. The invalidate for block *b1* cannot be processed by processor A until the SPT eviction of page *p2* is completed (dependency 1 in Table 4.1). The SPT eviction cannot progress until space is available in the pending request cache of processor B for its invalidate of block *b2* (dependencies 2 and 4 in Table 4.1). Space will not become available in this pending request cache until A services the invalidate of block *b1* (dependencies 3 and 5 in Table 4.1). This cycle of dependencies (1→2→4→3→5→1 in Table 4.1) leads to a deadlock. In our implementation, we break this dependency by allowing the invalidate message to invalidate the buffered reply without waiting for the SPT eviction to complete.

Node	Event	Consequent Action	Resulting Dependency
A	*b1* read copy is received; buffered due to full SPT	eviction of page *p2*'s SPT entry is initiated	*b1* copy buffered until completion of SPT entry eviction (1)
A	eviction of page *p2*'s SPT entry	request node B to writeback block *b2* of page *p2*	*p2*'s SPT entry freed when *b2* is written back by B (2)
B	local write request for *b1* must invalidate A's copy	invalidation is allocated storage in Pending Requests Cache	Pending Requests Cache storage freed when A sends acknowledgment (3)
B	request to writeback *b2* received — must invalidate copies of *b2*	invalidation is suspended due to full Pending Requests Cache	*b2*'s writeback can progress if space freed in Pending Requests Cache (4)
A	invalidate request for *b1* received from B	invalidation is suspended until *b1* is unbuffered	A can acknowledge *b1*'s invalidation when *b1* is unbuffered (5)

Table 4.1: Deadlock due to contention for physical resources. The sequence of events, consequent action, and resulting dependency at each node are shown. Deadlock occurs due to the cycle of dependencies 1→2→4→3→5→1. The deadlock is avoided in our implementation by allowing invalidate messages to invalidate buffered read copies. This breaks the cycle by eliminating dependency 5.

4.2.2. Handling of Race Conditions

In a system that supports multiple concurrent processes, the outcome of the execution may depend on the order of accesses to shared resources. This order may vary from one execution to the next, depending on the access times to the memory system (which may vary due to caching) or variations in the communication latencies through the network. A *race condition* is present if the outcome can vary from one execution to the next due to these causes. The system must either ensure that all possible outcomes are ‘‘correct’’ or use explicit locking and synchronization to prevent incorrect or undesirable ordering of accesses to

shared resources.

Race conditions can occur in a shared bus multiprocessor with snoopy caches [Katz85]. In systems based on snoopy caches, the ‘‘races’’ of concern are between accesses to the cache from the local processor and remote accesses across the bus. Many more races are possible in a system where the nodes are interconnected by a point-to-point network. Specifically, in such a system messages may be delivered in a different order than they were sent. If the network provides multiple paths between pairs of nodes and does not guarantee pairwise FIFO (First-In-First-Out) delivery of messages, it is obvious that messages can race in the system. Even when the network guarantees pairwise FIFO ordering of messages, coherence protocols can themselves introduce race conditions. Coherence protocols (such as in [Leno90], for example) may require different messages to be sent through different sequences of nodes. For example, some messages may be sent directly to the servicing node, while other messages may be processed by intermediate nodes before reaching the servicing node. The underlying network is oblivious to the coherence protocol-level routing. Hence, even when the network guarantees pairwise-FIFO, messages from the same node can reach the servicing node in different order than in which they were generated. We discuss this race problem associated with coherence protocols in more detail below.

Coherency information is distributed among several nodes with the distributed directory coherence protocol. As a result, a node that requires access to a block may not be able to identify the owner of the block from its local fragment of the coherence directory. To acquire access to the block the node must contact the node

that can identify the block owner. Traditional coherence protocols employ a technique called *forwarding* to facilitate this step. With the use of forwarding, the request message is first sent to the node that maintains the identity of the owner. This message is subsequently *forwarded* by this intermediate node to the node that owns the block. However, protocols typically do not always use the forwarded route. For example, when a message must be sent replying to a message from the owner, the reply message may be sent directly to the owner without forwarding through additional nodes. These protocols selectively determine whether a protocol message uses the forwarded route or the direct unforwarded route. We refer to such protocols as protocols based on *selective forwarded-routing*. Since requests from one node to another may follow different paths, protocols with selective forwarded-routing introduce race conditions.

To illustrate the use of selective forwarded-routing, consider the handling of a read request and an acknowledgement in the DASH multiprocessor [Leno90]. In DASH, when a cluster has a read miss, the cluster sends a read request to the home cluster of the block. If the block is owned dirty by a remote cluster, the home cluster forwards the request to the dirty, owning remote cluster (use of forwarded-routing). This owning cluster sends the data directly to the requesting cluster (see [Leno90] for a definition of home and remote clusters). However, when the cluster with the read copy receives an invalidate message from the owning cluster, the acknowledgement reply is sent directly to the owning cluster. Unlike the original read request, the acknowledgement message is not forwarded through the home cluster (forwarded-routing is not used).

Whether the coherence protocol selects the forwarded route or the

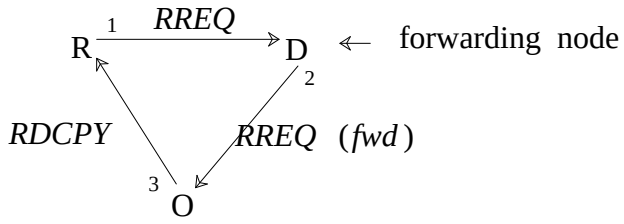


Figure 4.7: Forwarding in Protocol. Node *D* forwards node *R*'s request to node *O*.

unforwarded route can also depend on the state of the protocol rather than on the type of the message. For example, forwarding can be used only when a node initiating a request does not know the owner of a block. In Figure 4.7, the node *R* does not know the current owner of a page, and, hence, sends its read request to the default owner *D*. The default owner *D* knows the current owner *O*, and it forwards the request to *O* who sends a reply directly to *R*. For better performance, *R* can cache the identity of the owner when it receives the reply to its first request, and subsequent requests can be directly sent to the owner, without using forwarding (selective forwarded-routing).

We illustrate the occurrence of races in DSM systems due to the use of forwarding (since the use of the forwarded-route is selective in all protocols that we are aware of, we simply refer to these protocols as protocols that use forwarding) in the next sub-section.

4.2.2.1. Race Conditions in DSM Systems

We have previously observed that read requests and acknowledgements can follow different paths in the DASH multiprocessor [Leno93]. Hence, in DASH, a race condition is possible between a read request and an acknowledgement message sent from a node. Suppose a cluster *R* has a read copy of a block, which gets invalidated due to a write by cluster *O*. Subsequently, cluster *R* requests a read copy of the block. The home cluster of the block is *H*. As we discussed previously, in DASH [Leno90], *R* sends its acknowledgement for the invalidate directly to *O*. However, *R*'s subsequent read request is sent to *H*, which can be forwarded to *O* for service. Since the acknowledgement and the read request follow different paths from *R* to *O*, the read request can race the acknowledgement to *O*. When the read request is received ahead of the acknowledgement, the situation is equivalent to the case where a read request is received while a write is in progress. Hence, the race condition can be handled by queuing the read request or denying service (negative acknowledgement) as a read request received during a write would be handled.

A race condition such as the one discussed above can also occur in our DSM system if forwarding is used. We describe a situation in our DSM system that is slightly modified from the previous example. In Figure 4.8, node *R* initially requests and receives a read copy of a block from owner *O*. Subsequently, node *R* drops its read copy (replacement) and sends a copy invalidated message *CPINV* (message types are discussed in Section 4.3.3) to update the owner *O*. When node *R* subsequently needs to read the block, it does not know the current owner and, hence, sends the read request (*RREQ*) to the default owner *D* who forwards it to the

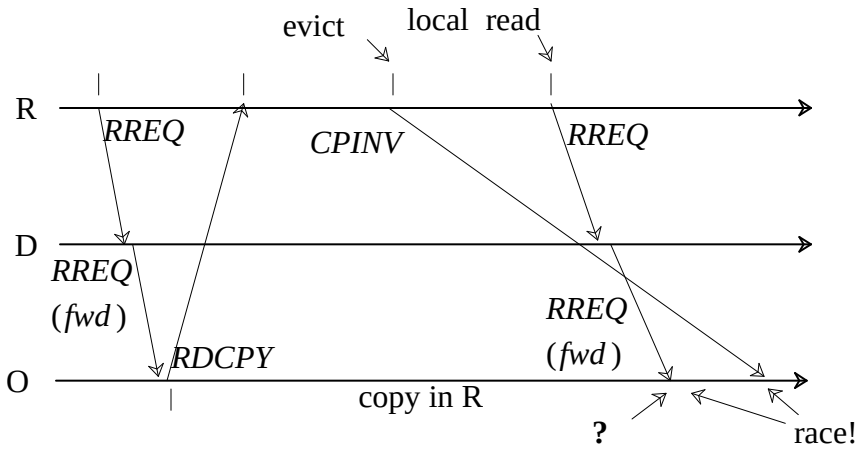


Figure 4.8: Races in protocols with forwarding. Node *R*'s read request is forwarded by node *D* to node *O*. The read request sent after the copy invalidated message *CPINV* can arrive at node *O* earlier since it followed a different path due to forwarding.

owner *O*. As with the example in DASH, the messages *CPINV* and *RREQ* follow different paths from node *R* to node *O*. Consequently, the *RREQ* can race the *CPINV* message, and arrive first at *O*. If this occurs, node *O* would receive a read request from a node that, in its view, already has a copy of the block. The arrival of a read request message from a node that already possesses a copy of the requested block can pose a problem for the coherence protocol.

In the example scenario of Figure 4.8, it may be possible for *O* to identify the race condition, and act appropriately. If *O* maintains a copy directory, it can identify the race condition by observing that the *RREQ* was sent by a node *R* that is already present in the copy directory. Node *O* can resolve the race by denying the service for the read request (force a retry using a *RETRY* message). In general, however, it may not be possible to identify a race condition purely from the local state.

As an example of a race condition which cannot be correctly resolved based on local information, consider a modification to the example shown in Figure 4.8. In the modified scenario, shown in Figure 4.9, node *O* is forced to evict node *R* from its copy directory (due to copy directory overflow). To perform this eviction, node *O* sends an invalidate message (*INV*) to node *R*, and drops node *R* from the copy directory. Since node *R* is no longer identified in the copy directory, when node *O* receives node *R*'s read request *RREQ* ahead of its copy invalidate message *CPINV*, node *O* cannot identify that a race has occurred. From its local state, node *O* cannot distinguish if the *RREQ* raced another message, or if it is a request from a new node. Node *O* will provide a read copy to the read request. This can cause the system to enter an incorrect state, as discussed below. The messages in transit at this point are the copy invalidated message (*CPINV*) from node *R* to node *O*, and the invalidate (*INV*) and read copy (*RDCPY*) messages from node *O* to node *R*. Upon receiving the copy invalidated message, node *O* will (erroneously) drop node *R* from its copy directory. Meanwhile, node *R* will acknowledge (*ACK*) the invalidate message, and subsequently receive the read copy. After node *O* receives the acknowledgement (*ACK*), there are no messages in transit. Since, node *R* possesses a read copy, and node *O* does not record this in the copy directory, the copy directory state is incorrect. This can allow future writes to proceed without invalidating the read copy of node *R*, violating the write exclusive protocol. This specific race condition may not be a problem in the coherence protocols used in [Leno93, Agar95, Kusk94] since they employ a full copy directory without need for eviction. Such race conditions have not been discussed in their literature. As will be discussed in Section 4.2.2.2, our coherency protocol avoids this problem by using ‘‘bilateral exchanges.’’ With this approach, the read request message REQ

from node *R* cannot race its earlier copy invalidate message *CPINV*.

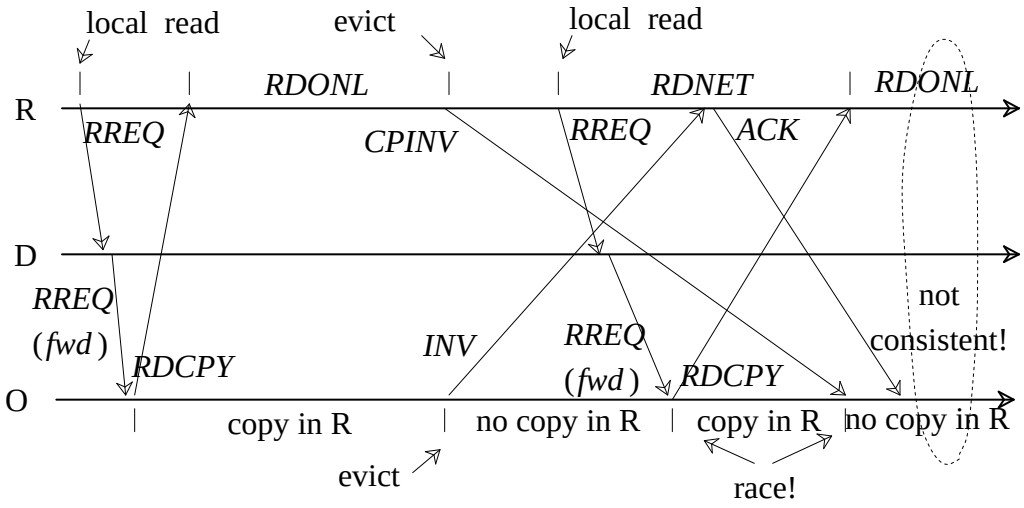


Figure 4.9: Race condition that cannot be locally identified. Node *O* evicts node *R* from its copy directory, and, hence, cannot locally determine that *R*'s read request raced its copy invalidated message. In the end, although *R* has a read copy, the copy directory in owner *O* does not reflect this. The copy directory state is incorrect.

The designers of the DASH prototype [Leno90] also discuss race conditions caused by the use of two independent networks in their system. DASH uses two networks, one for requests and one for replies. Since requests and replies use different networks, races between request and reply messages are possible. For example, when a node sends a read copy reply to another node and subsequently sends it an invalidate request, the invalidate requests can race the preceding read copy reply. In DASH, such race conditions are handled by designing the protocol engine to identify race conditions and providing special mechanisms to process them. To handle the race between an invalidate request and a read copy reply, an additional state is added to the coherence protocol. When an invalidate is received for a block that has not yet been received, the block is changed to a state which

identifies that the block must be invalidated when it is received. When the read copy is received, it is discarded. The block is changed to the invalid state, and an acknowledgement reply is sent to the owner.

In summary, protocols that employ forwarding involve race conditions. This possibility of races (even in a network with FIFO service discipline) presents an obvious problem in the design of correct distributed directory coherence protocols. The protocol design needs to identify possible race conditions and develop mechanisms to avoid them, or develop mechanisms to detect and handle them. This makes the protocol design and implementation complex. Races can also mask implementation bugs and runtime error conditions, which makes design verification and error detection difficult. We discuss a new protocol technique that addresses the race problem in the next sub-section.

4.2.2.2. DSM Coherency based on Bilateral Exchanges

As discussed in the previous sub-section, many race conditions are introduced by the use of forwarding. The problems caused by these race conditions provide the motivation for developing coherency protocols that avoid forwarding. In this sub-section, we introduce a coherency protocol where all exchanges between nodes are *bilateral*, i.e., do not involve any forwarding. In a protocol based on *bilateral exchanges*, if a node cannot service a request itself, it will send an appropriate reply message back to the requester, and not forward the request to any other node. For example, consider the example where the default owner forwards a read request to the current owner node. This protocol interaction is modified to use bilateral exchanges as shown in Figure 4.10. Since the default owner *D* cannot

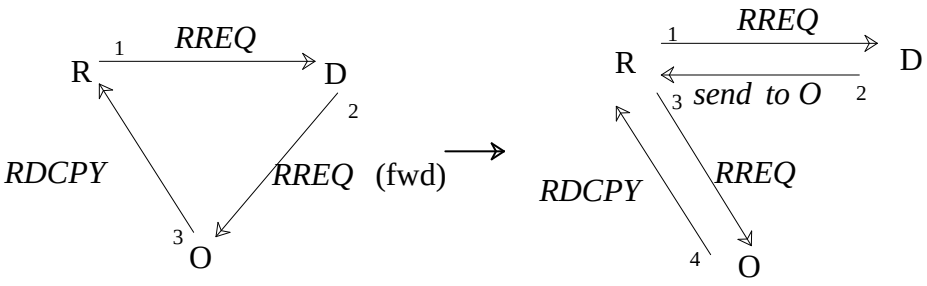


Figure 4.10: Protocol based on bilateral exchanges. The protocol exchange on the left uses forwarding. The modified protocol exchange using bilateral exchanges is shown on the right. All protocol exchanges between nodes are bilateral. Node *D* identifies the owner *O* to the requester *R*, instead of forwarding the request. *R* requests a read copy from the owner *O* bilaterally.

service node *R*'s read request itself, it sends a reply back to node *R* identifying the current owner *O*. The exchange between node *R* and node *D* is bilateral. After receiving the response from the default owner, node *R* has another bilateral exchange with the current owner of the block *O* to receive the read copy.

The bilateral exchange technique is designed to avoid races in networks that guarantee pairwise FIFO ordering of messages. Since all exchanges in the protocol are bilateral, i.e., pairwise, protocol messages exchanged between any pair of nodes will always arrive in order. Figure 4.11 shows how a race situation such as the one described earlier in Figure 4.8 is not possible in a protocol based on bilateral exchanges. Since node *O* receives the copy invalidated message *CPINV* and the read request message *RREQ* directly from node *R*, the network's FIFO service discipline guarantees that they will be received in the exact order they were sent.

The absence of message races with bilateral exchanges makes the design and implementation of the protocol simpler. Verification of the design and detection of

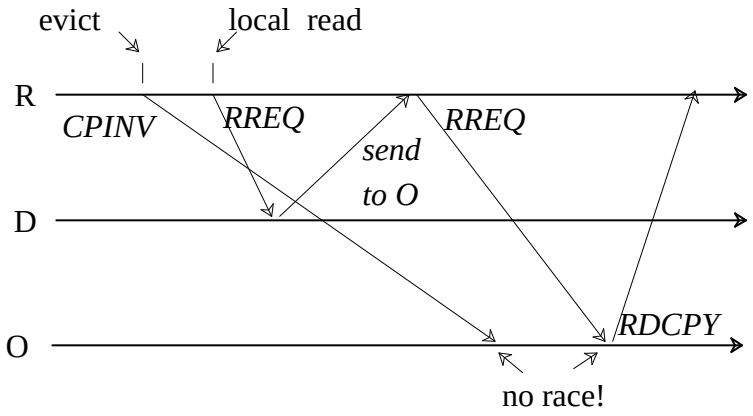


Figure 4.11: Protocol based on bilateral exchanges avoids races. The copy invalidated message and the read request from *R* follow the same path to *O* due to the bilateral exchange. With the network guaranteeing pairwise FIFO, these messages will arrive at *O* in the order they were sent.

runtime errors will also be simplified. However, while the protocol based on bilateral exchanges is simpler, it is less efficient. Comparing Figures 4.8 and 4.11, it is clear that the number of messages and the number of host processing delays will be higher with the usage of bilateral exchanges. In our protocol implementation (described later in this chapter), we use an hybrid mechanism to reduce the inefficiency due to bilateral exchanges.

4.2.3. Necessity of Transient Block States

The coherence directory in each node must maintain the state of the blocks in that node. The state of each block identifies whether the block is invalid, readable or writable. To change the state of a block, a request must be sent to a remote node requesting the new access permission. The block state can be changed only after a reply message granting the request has been received. Since this request-reply

protocol exchange is not atomic, additional block states may be needed to maintain the transient state of the blocks.

Maintaining transient block states clearly provides several benefits. Since transient states identify outstanding requests, they allow the coherence protocol to verify whether a reply message that is received is a valid reply. Second, if the reply message received denies the requested access (negative acknowledgement), the transient block state allows the coherence protocol to determine the type of the request message that must be re-issued. Transient block states are also useful in systems where multiple processes can execute on a node. When a second process requests similar access to the block, the coherence protocol can identify that a request for the access has already been issued to the network. A second request need not be sent to the network. Thus, the transient state allows the coherence protocol to filter requests to the network.

Transient block states are also a compulsory requirement in some cases. Certain accesses to a block, such as a write access, require that concurrent requests received from other nodes for competing or conflicting accesses be denied service. A transient block state that identifies the specific access in progress for the block is necessary to allow the coherence protocol to refuse service to competing accesses. Transient block states are also required in the coherence protocol if messages that cannot be denied service can be received at a time when they cannot be immediately serviced (see the second example later in this subsection). In such cases, a transient state must be maintained indicating that the message has been received. This allows the coherence protocol to subsequently service the message when service can be granted. Transient block states are also required to handle

some race conditions, as previously discussed in Section 4.2.2.

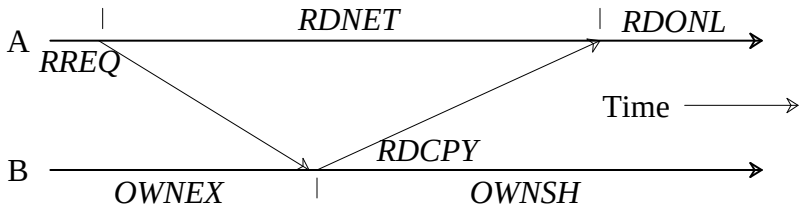


Figure 4.12: Protocol actions for a read access. Node *A* requests and receives a read copy to service local processor's read access. The block is maintained in the read network state *RDNET* while the request is outstanding.

In the rest of this sub-section, we discuss some examples showing the use of transient block states. Our first example illustrates the use of block states to identify outstanding requests. Consider the flow of a read access in the coherence protocol. Figure 4.12 illustrates the sequence of actions. Node *A* requires read access to a block. To service the read access, node *A* sends the read request (*RREQ*) to node *B*, which is the owner of the block. To indicate that a read request is outstanding for the block, the local state of block in *A* is marked to be in the transient read network state *RDNET*. This transient read network state *RDNET* is necessary in our design for at least three reasons. First, the state allows a memory frame to be reserved for the block at the time the request is issued. The alternative approach of allocating a memory frame for the block only when the reply arrives is more complex. For example, the scenario when no memory frame is free when the reply is received must be handled without the possibility of deadlock. Second, if another thread running on the same processor accesses the block again while the request is in progress, the transient state prevents a second request from being issued. Allowing multiple requests for the same block to be issued from a node

adds complexity to the design for no benefit. Such a design must address additional correctness issues associated with the copy directory. For example, suppose the block received in reply to the first request is discarded when the reply to the second request is in transit. In this case, the copy directory for the block at the owner must continue to indicate the presence of a read copy at the requester even after the owner receives notification that a copy was discarded. Third, if a retry reply (*RETRY*) is received for the request, the transient state identifies that a read request must be re-issued (in most cases the necessary information can be obtained from the Pending Requests Cache (Section 4.1) without accessing the directory). In general, the coherence protocol and the system must be carefully studied to determine if the transient read network state can be eliminated from the design.

When node *B* receives the read request, node *A* is added to the copy directory, and a read copy (*RDCPY*) is sent to *A*. The block which was previously owned exclusively (*OWNEX*) in node *B* is moved to the *OWNSH* state to indicate the presence of remote copies. When node *A* receives the read copy, it verifies that this reply corresponds to the read network state *RDNET* of the block. Hence, the block is updated to the read-only state *RDONL*, and the processor is allowed to complete the read access. A similar transient block state *WRNET* is used to identify outstanding write requests. Read or write requests for the block from other nodes are denied service when the block is in the *WRNET* state.

To illustrate another situation where transient block states are necessary, consider the addition of a replacement event to the read access scenario. In our coherence protocol, a node that replaces a block must send a message updating the

owner with this information. The node evicting the block will reclaim the space freed by the eviction without waiting for any confirmation from the owner. When the owner receives the replacement update, it does not have the option to deny service to the message.

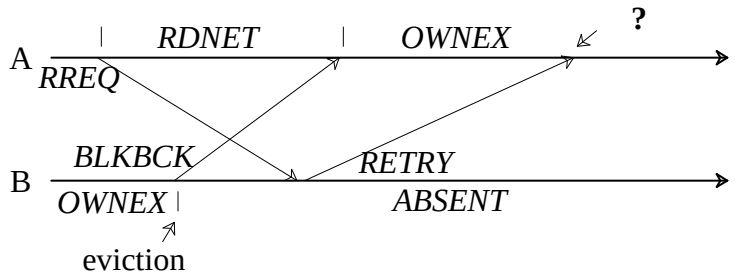


Figure 4.13: Replacement during read access. Node *B* which is the owner of the block replaces the block, and writes the block back to node *A*. Node *A* must process the writeback, and does so by changing the block to the exclusively owned state *OWNEX*. The subsequent retry message *RETRY* is unexpected in the *OWNEX* state.

The example including a replacement event during a read access is shown in Figure 4.13. Node *B* evicts the block due to local memory contention. Node *B* writes the block data back (*BLKBCK*) to node *A* while a read request to the block is in progress in node *A*. (The block will be written back to node *A* when node *A* is either the the default owner of the block or the owner of the page to which the block belongs.) After the write-back, node *B* does not possess the block. Hence, when it receives node *A*'s read request for that block, a retry reply message (*RETRY*) is sent to *A*. When node *A* receives the block data back (*BLKBCK*) it must service it. *A* recognizes that there are no copies of the block in the system from the *BLKBCK* message and the local state of the block is changed to the exclusively owned state *OWNEX*. The read access of the processor on node *A* can

be completed at this time. However, when node *A* receives the retry message *RETRY* subsequently, the local state of the block indicates no outstanding request. The only options that are available are to flag an exception or error in the protocol, or to ignore the retry message. The first option is unsuitable since it would halt the system for a perfectly normal (possibly common) event. While the second option may be safe in this specific case, it has the potential to mask other exception conditions, implementation or runtime errors, thus compromising the robustness of the protocol.

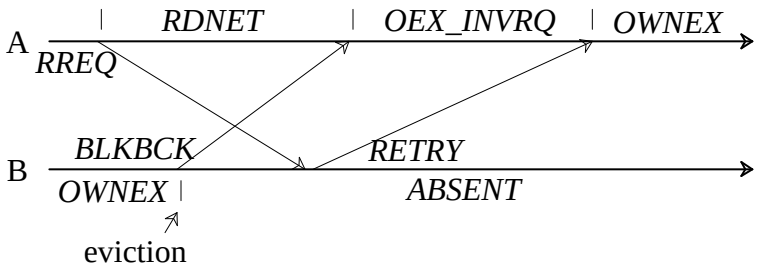


Figure 4.14: Addition of a transient protocol state to handle replacement. On receipt of the writeback, node *A* changes the block to a new *OEX_INVRQ* state which identifies the outstanding request. When the retry message *RETRY* is received, the block moves to the exclusively owned state *OWNEX*, and the message can be safely dropped.

The solution to this problem is to modify the coherence protocol design to preserve the information about the outstanding request even after the block back (*BLKBCK*) update message is received. Specifically, a new state can be added to the coherence protocol for this purpose. Upon receiving the writeback message in the read network state *RDNET*, the block is moved to a new state *OEX_INVRQ* which indicates that the block is owned exclusively with a request outstanding (*OEX_INVRQ* is a mnemonic for ‘‘owned exclusive with a pending invalid

request’’). As before, the processor’s read access on node *A* can be completed at this point. When the retry message is received, the block can be transferred to the owned exclusive state *OWNEX*, and the retry message safely dropped. The modified coherence protocol flow is depicted in Figure 4.14.

Similar situations can occur during a write access, and similar solutions can be used. The handling of remote read and write requests in the *OEX_INVRQ* state (when a reply is pending) is an interesting issue. Although it is possible to service these remote requests while continuing to maintain the information about the outstanding request, it can lead to an increase in the number of protocol states and complicate design and verification. A viable alternative is to deny or postpone service to these remote requests until the reply for the outstanding request is received. This is the solution that we use in our protocol implementation.

We have previously discussed (Section 4.2.2) the use of transient block states to handle race conditions. As we discussed earlier, in the DASH protocol [Leno90], an invalidate request message can race a previously issued read copy reply since they use different networks. The invalidate cannot be denied service, but cannot be serviced until the read copy is received. A transient block state is maintained to track the outstanding invalidate. When the read copy is received subsequently, the transient block state indicates that the block must be invalidated and an acknowledgement must be sent.

4.2.4. Handling of Write Requests

A write miss occurs if the block to be modified is either not present locally or is present without write permission. To resolve the write miss, the block data and/or write permission must be acquired from the current block owner. This is accomplished by sending a write request to the block owner. When handling write requests, an important issue is whether or not the block data has to be sent to the requesting node over the network. If the block data is not present in the node and only part of the block is to be modified, then, generally, a block data transfer is necessary. However, it may not be necessary to transfer block data for a write request in other cases. For example, if the requesting node already possesses a read copy of the data, it is not necessary to re-send the data to the requester. It is sufficient to send a message granting write access (i.e., granting ownership). This approach can also be used if the requesting node will overwrite the entire data without reading the previous contents. While these cases can also be handled correctly by sending the data, the elimination of data messages (which are typically larger than control messages) can reduce the latency of the access and the load on the network.

It may not be possible to accommodate this optimization, however, for reasons discussed below. First, it is typically not possible to identify the case where a processor intends to write the entire data without reading it prior to the write. The size of the data being written may be smaller or larger than the size of data to be transferred (block or page) and cannot be determined by the coherence protocol. Also, the coherence protocol cannot typically distinguish between pure writes and read-modify-writes of the processor. Second, the node servicing the

write request may not be able to determine if the requester has a read copy of the data. For example, this will be the case if the coherence protocol allows data to be shared in the broadcast state. In this case, the owner of the data may not maintain a copy directory and cannot determine if the requester already has a read copy.

It is reasonable to consider requiring the requester to indicate whether it possesses a read copy, to solve this problem. But, this is not sufficient when the copy directory is not maintained by the node servicing the write request. In this case, a concurrent write by the owner of the block can invalidate the read copy of the requester, after the requester indicated that it possessed a read copy. Note that, from the discussion in Section 4.2.1 on deadlocks, invalidate messages must be immediately serviced. The requesters write would be granted after the write of the block owner, and if the data is not sent to the requester, the last copy of the data would have been discarded from the system.

With both the one-level and the hierarchical schemes, the copy directory may not be maintained by the node servicing write requests. In the one-level scheme, the copy directory is maintained by the block owner, but write requests are serviced by the default owner. In the hierarchically managed system, the write access to the block will be granted by the node owning the page (to which the block belongs), but the copy directory for the block is maintained by the owner of the block (who may be different from the owner of the page). In the hierarchical scheme, additional coordination between the block owner and the page owner would be needed for the page owner to identify if the requester possesses a read copy. Such coordination could be achieved, for instance, by requiring the block owner to send the copy directory of the block to the page owner, when the block owner

relinquishes ownership of the block. The page owner can determine whether the node requesting the write currently has a copy of the block by looking at this copy directory.

In our DSM implementation, we maintain the flexibility to allow blocks to be shared in the broadcast state to investigate support for heavily shared data. For this reason and the additional complexity associated with determining if data has to be transferred to the requester, we transfer the data on all write requests in our coherence protocol implementation. Protocol implementations for DASH [Leno90] and FLASH [Kusk94] also transfer data on all write requests.

4.3. A Coherence Protocol for Hierarchically Managed DSM

We present our distributed directory coherence protocol in detail in this section. While our coherence protocol is an invalidation based ownership protocol similar to the protocols implemented in DASH [Leno90], FLASH [Kusk94], and Alewife [Agar95], it differs in its support for multiple levels of ownership. We sketched our approach to resolving issues in designing distributed directory coherence protocols in Section 4.2. Deadlocks are avoided by ensuring that invalidate and reply messages are always sunk. Special block states are used to indicate pending operations. Data is always transferred for write requests. Many potential race conditions are avoided by using bilateral exchanges rather than forwarding. Specifically, we employ a hybrid approach, which uses bilateral exchanges only for write requests (which are fewer than read requests) and forwarding for read requests. We describe our protocol in detail in the following sub-sections.

4.3.1. Protocol Ownerships

The hierarchical scheme supports ownership of data at both the page and block levels. Each page that is in the physical memory of some node in the system is owned at both the page and block levels. Initially, a page is owned by the node (*page owner*) that reads the page from disk. Subsequently, the page ownership can be transferred by migrating the page to another node or by swapping the page to disk. Each block of the page can also be independently owned at the block level. When a page is read from disk, the page owner is the *block owner* of all blocks of the page. When a node other than the current block owner writes to the block, the block ownership is transferred to the writing node. The page owner coordinates these block ownership transfers, and maintains the identity of the block owner of each block of the page.

Each page in the virtual address space is also statically assigned a *default owner*. The default owner maintains the identity of the current page owner in the Migrated Page Table (MPT). If a page is not indicated in the MPT, the page resides only on disk. The default owner and its MPT are updated whenever the page is migrated or written back to disk. It is important to note that while both the page owner and the block owner maintain storage for the page in their physical memory, the default owner does not maintain storage for the page in its physical memory (unless it is also accessing the page).

Directory information about a shared page is distributed among the nodes in the system as follows. The default owner of the page identifies the current page owner. The page owner maintains the identity of the block owner of each block of the page, as well as the location of read only copies of any blocks that it owns

itself. The block owners maintain the location of read only copies of the block. Any node in the system that has at least one block of the page in its physical memory maintains the identity of the page owner, but it does not maintain the identity of the block owners. All nodes in the system know the identity of the default page owner of every virtual page.

The responsibility for handling access requests to a shared page is distributed as follows. The block owner services read requests for the block by providing read only copies of the block. The block owner is also responsible for invalidating these read only copies during a write. The page owner services write requests by coordinating the transfer of block ownership from the previous block owner to the new writer. The default owner services read and write requests by identifying the page owner if it is resident in some physical memory in the system, and by issuing the request to disk, otherwise. Any node can send read and write requests to the page owner, if it knows the page owner, and to the default owner, otherwise.

4.3.2. Page and Block States

A page can be in one of the following protocol states — invalid (*INVALID*), valid private (*PRIVATE*), valid shared (*SHARED*), read requested (*RDNET*) or write requested (*WRNET*). As previously mentioned in Section 4.1.1, this state of the page is maintained by the PPT entry in 2 PPT state bits and 5 PPT meta-state bits. The two PPT state bits identify whether the page is invalid, valid, read requested or write requested. When the state bits indicate that the page is valid, one of the meta-state bits identifies if the page is private or shared. The other meta-state bits maintain additional information about the page when the page is

valid. One meta-state bit indicates if the page is clean or dirty. The other three meta-state bits maintain information relevant in page replacement. They indicate if the page is pinned from being replaced, if the page is being evicted to free the PPT entry of the page and if the page is being evicted to free the SPT entry of the page respectively.

access_permit	Implication for local processor
ABS	Absent; no access permitted
RDO	Only reads permitted
OEX	Owned exclusive; reads and writes permitted
OSH	Owned shared; read permitted; invalidate for write

Table 4.2: *access_permits* of a block.

When a page is shared, the protocol also maintains a state for each block of the page. The block state has two components: *access_permit* and *access_waitlist*. The *access_permit* component specifies what accesses to this block are permitted for the local processor. The *access_waitlist* component identifies what access is pending on the block. There are four possible values for access_permit (Table 4.2). The *ABS* state indicates the block is absent and the local processor has neither read nor write access permission. The local processor is allowed read access but no write access if the access_permit is either *RDO* (read only) or *OSH* (owned shared). The *OSH* state differs from the *RDO* state in two respects. First, the *OSH* state indicates the local processor owns the block. Second, it indicates the processor can secure write access by invalidating all read copies. The local processor is permitted both read and write access when the access_permit is *OEX* (owned exclusive).

If the access_permit state does not allow the local processor's access, access

access_waitlist	Access in progress
NONE	None
RDNET	Local read request
WRNET	Local write request
LOCK	Remote write request

Table 4.3: *access_waitlists* of a block.

permission needs to be obtained from the block owner or the page owner. Since this is not an atomic operation, there is a need to keep track of pending operations on the blocks using the access_waitlist state (Table 4.3). The access_waitlist can indicate that no request is outstanding (*NONE*), a local processor's read request is outstanding (*RDNET*), a local processor's write request is outstanding (*WRNET*), or that a remote processor's write request is outstanding (*LOCK*). At most one outstanding request is allowed per block. An outstanding request for a block blocks all other requests for the block. The semantics of access_waitlists *NONE*, *RDNET*, and *WRNET* are obvious. The *LOCK* access_waitlist indicates that, in response to a remote request, exclusive block ownership is being acquired, and the block ownership should be transferred as soon as it is acquired. As we will see in the discussion of protocol messages (Section 4.3.3), exclusive block ownership is acquired by invalidating read copies, if the local node is the block owner, or by requesting a write back from the block owner, if the local node is the page owner (but not the block owner).

The two components of the block state, access_permit and access_waitlist, can each be represented in 2 bits (for a 4-bit block state). The block state is shown as a composition of the access_permit and the access_waitlist in Table 4.4. The semantics of most of the block states and the state transitions follow easily from the

		access_waitlist (2 bits)			
		NONE	RDNET	WRNET	LOCK
access _permit (2 bits)	ABS	ABSENT	ABS_RDNET	ABS_WRNET	ABS_LOCK
	RDO	RDONL	X	RDONL_NET	RDO_LOCK
	OEX	OWNEX	OEX_INVRQ	OEX_INVRQ	X
	OSH	OWNSH	X	OWNSH_NET	OWNSH_XFR

Table 4.4: block states — composition of access_permit and access_waitlist

above discussion of access_permit and access_waitlist. We elaborate on the less intuitive block states. *OWNSH_NET* and *OWNSH_XFR* both signify that invalidation is in progress, for a local write access in the former case, and for a remote write request in the latter case. When invalidation is complete and exclusive ownership is acquired, the local processor can complete its write (*OWNSH_NET*), or the ownership can be transferred to the remote requester (*OWNSH_XFR*). The block states *ABS_LOCK* and *RDO_LOCK* are valid only in the page owner, when it is coordinating the transfer of block ownership from the current block owner to a new requester. The block state *OEX_INVRQ* is also valid only in the page owner. This state indicates that the block is owned exclusive and there is a pending invalid request involving this block which has been sent from this node. As discussed in Section 4.2.3, this state occurs when the block is written back to the page owner by the previous block owner (due to a replacement in the block owner) when a local request for the blocks was outstanding at the page owner. This protocol state is necessary to handle the retry response that will be received for the pending invalid request. No remote requests are serviced in this state until a retry message is received for the outstanding request (see

Section 4.2.3). When the retry message is received, the block state is changed to *OWNEX*.

4.3.3. Protocol Messages

Message	Function	Class
RREQ	Read request	Request
WREQ	Write request	Request
WRBCK	Request write-back of block	Request
INV	Invalidate block read copy	Command
PGINV	Invalidate page	Command
PAGE	Page data, that is currently not shared	Reply
RDCPY	Block data for read access	Reply
WDATA	Block data for exclusive ownership	Reply
ACK	Acknowledge invalidation	Reply
PGOWN	Identification of page owner	Reply
RETRY	Require retry	Reply
BLKBCK	Write back of block data	Reply/Update
CPINV	Copy invalidated notice	Update
PGBCK	Write back of page data	Update

Table 4.5: Protocol messages and their function.

Table 4.5 lists the messages used in our coherence protocol. We classify the messages into four categories — requests, commands, replies, and updates. Request messages solicit a reply message indicating either that the service was provided or that it was not provided. Command messages also solicit a reply, but cannot be denied service. As we discussed in Section 4.2.1, this ensures that protocol deadlocks are broken. Reply messages are sent in response to a prior request. If the reply does not satisfy the requested service, the request will be re-issued. Update messages are caused by replacements, do not expect a reply, and

cannot be denied service. We describe the functionality of some non-trivial messages below.

The request message *WRBCK* is used by the owner of the page to request the current block owner to write back the block to the page owner, when the block ownership needs to be transferred. If a local write access is not in progress, the block owner writes back the block with the *BLKBCK* message. If a request cannot be immediately satisfied, a *RETRY* response can be sent in response to any of the requests (negative acknowledgement). The requester can subsequently re-issue the request. The *PGOWN* reply is used by the default page owner to implement the bilateral exchange on a read or write request. The default owner does not forward the request to the page or block owner. Rather, the requester is informed of the identity of the page owner using the *PGOWN* message.

When a node replaces a page (and the blocks belonging to that page) from its physical memory, the default page owner, the page owner, and/or the block owners must be informed. The nodes which must be informed, and the message types that must be used to inform these nodes depend on the state of the evicted page and the state of the blocks of the page. When a node replaces a page that it owns, the contents of the page (all blocks) are written back to the default owner with a *PGBCK* update message. When the page is not owned by this node, any blocks that are owned locally are each written back to the page owner using *BLKBCK* messages. If the read copy of any blocks of the page was discarded, the owner of each block is notified using a *CPINV* message.

The *PGINV* message is a performance optimization for the hierarchical scheme. When a node replaces a page that it owns, the node ‘‘pulls’’ all blocks of

the page to itself (see Section 4.1.1) so that the page data may be written back. This can be accomplished by individually requesting the writeback of each block of the page that is owned by a remote node and by individually invalidating each read only copy of blocks of the page which are owned locally. The *PGINV* message accomplishes this task with lower message count and lower node processing by implicitly commanding the writeback of any blocks of the page that are owned by the recipient, and the invalidation of any read copies held by the recipient. A single *ACK* response along with the appropriate *BLKBCK*s signals the completion of the page invalidation in the recipient.

Messages *RDCPY*, *WDATA*, and *BLKBCK* carry one block of data. Messages *PAGE* and *PGBCK* carry one page of data. All other messages are control messages. In our current implementation, the *PAGE* message is used only when the page is read from disk. *PAGE*, *PGBCK* and other newer messages can be used to support page migration, but this has not been addressed in the current implementation.

4.3.4. The Flow of Protocol Actions During Reads, Writes and Replacements

The discussion of our coherence protocol for the hierarchically managed DSM system thus far has described the assignment of ownerships in the protocol, the various states of pages and blocks in the protocol and the various messages that are used in the protocol. The ownership and state information associated with the blocks and pages in the system are distributed amongst the nodes in the system. At each node, they are maintained in the tables used by the hierarchical scheme: the Present Pages Table (PPT), the Shared Pages Table (SPT), the Migrated Pages

Table (MPT) and the copy directory (described in Chapter3). When a processor access misses in a node or when a page is replaced from a node, different protocol messages are exchanged between several nodes and necessary tables are accessed at these nodes to look-up and update protocol ownership and protocol state information. The messages that are exchanged, the tables that are looked up and the transitions in protocol state and ownership all naturally depend on the transaction (read, write or replacement) and on the state and ownership distribution at that time. In the following three sub-sections, we describe in detail the flow of protocol actions used to service read requests (Section 4.3.4.1), to service write requests (Section 4.3.4.2), and to service replacement events (Section 4.3.4.3).

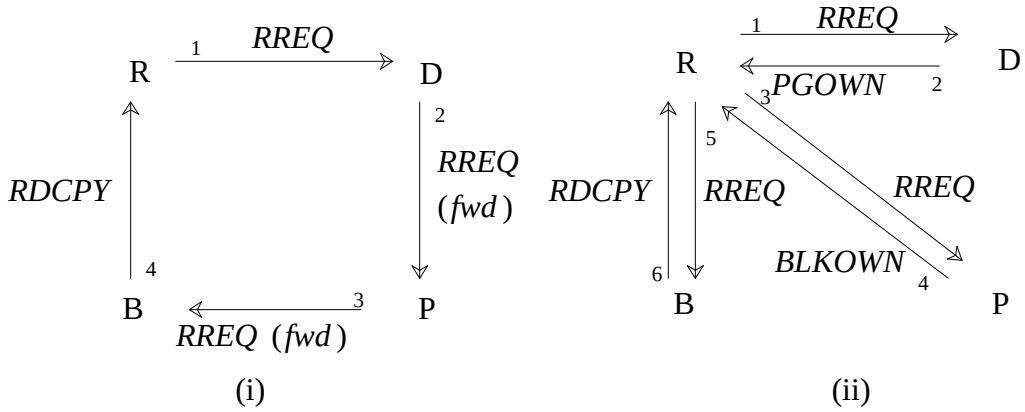


Figure 4.15: Comparison of bilateral exchange and forwarding for the hierarchical scheme. When a node making a request does not know the page owner, the default owner, the page owner and the block owner will all participate in servicing the request. The default owner identifies the owner of the page to the requester (using *PGOWN*), and the page owner identifies the owner of the block to the requester (using *BLKOWN*). With bilateral exchanges (case ii), six messages are sent and processed, as compared to four when forwarding (case i) is used.

Before we describe these flows of protocol actions, we first describe a

modification to our protocol to address the inefficiency of using bilateral exchanges. As we mentioned in our discussion in Section 4.2.2.2, protocols based on bilateral exchanges incur higher message count than protocols based on forwarding. To highlight this inefficiency, consider a specific example on the hierarchically managed system. In this example, a node that does not know the owner of a page requests access to a block. Also, the default page owner, the owner of the page and the owner of the required block are three different nodes. In this case the request must first be sent to the default page owner, then to the page owner, and finally to the block owner. The comparison of messages exchanged in the protocol based on forwarding and in the protocol based on bilateral exchanges is shown in Figure 4.15. With the use of bilateral exchanges, such a request would require six messages to be exchanged and processed. Using forwarding, the same request can be satisfied with four messages.

The higher number of messages used with bilateral exchanges will increase the latency of accesses and the load on the network. However, if forwarding is used, several race conditions must be handled. Hence, we use a hybrid protocol that minimizes the inefficiency due to bilateral exchanges. The hybrid protocol uses forwarding for efficiency where race conditions are easy to resolve and uses bilateral exchanges elsewhere to avoid the race condition problems discussed in Section 4.2.2.

In our read access protocol we use bilateral exchange at the default owner but use forwarding at the page owner. Since the page owner must be contacted to request a read copy of any block of the page, this approach is likely to reduce message count significantly. Since the default owner is contacted much less

frequently (only when a node has no blocks of the page), the reduction in message count by also forwarding at the default owner is likely to be marginal. The small improvement may not justify the additional complexity required to handle race conditions caused by forwarding at the default owner. With this approach, when the default owner receives a read request, it replies to the requester with the *PGOWN* message identifying the current page owner. The requester subsequently directly requests the page owner for a read copy of the block. When the page owner receives a read request, it is forwarded to the block owner, who directly responds to the original requester. This hybrid approach reduces the number of messages for the example shown in Figure 4.15 to five (from six messages with the use of pure bilateral exchanges).

Write accesses, which are less frequent than read accesses, are serviced using the bilateral exchange approach. The penalty for using bilateral exchanges instead of forwarding is smaller for write accesses since, even with forwarding, a message must be sent from the block owner back to the page owner to record the change in ownership.

The read and write protocols are described in detail in the following sub-sections. The use of forwarding in read accesses makes races possible. The races created by the use of forwarding are discussed in conjunction with the description of the read protocol.

4.3.4.1. Anatomy of a Read Request

A read request is initiated when a local processor's read access misses in the node's (*R*) cache and memory. We describe the flow of the protocol actions used to service the read request under different situations. The cases that we discuss include: 1) *R* has other blocks of the page, 2) *R* has no other blocks of the page, and the page is only on disk, 3) *R* has no other blocks of the page and the page is owned by the default page owner itself and 4) *R* has no other blocks of the page and the page is owned by a node other than the default page owner.

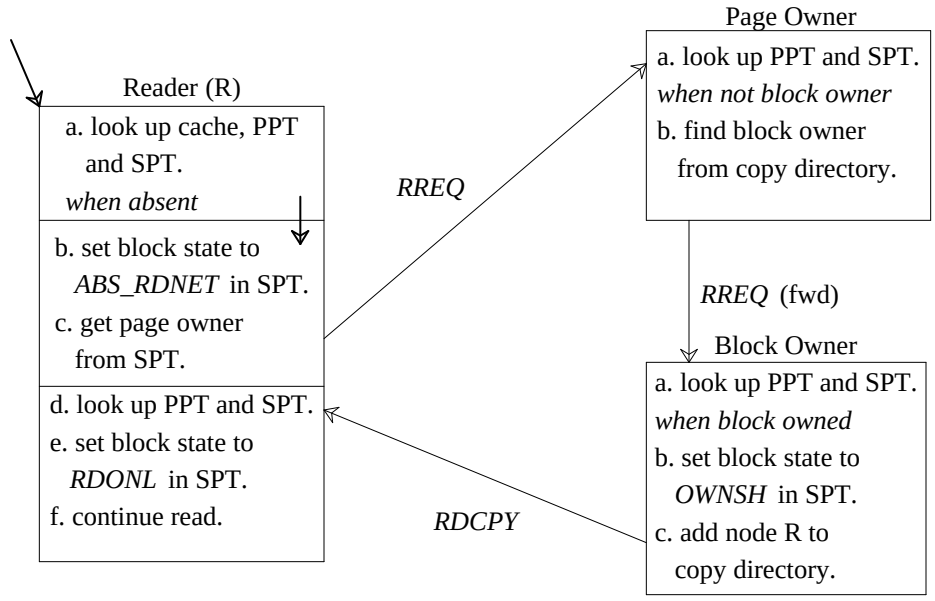


Figure 4.16: Anatomy of a read request. This is one possible flow of the protocol to satisfy a read miss by the local processor on node *R*. *R* does not have the block required by the read access, but has other blocks of the page. Hence, *R* knows the identity of the page owner. The block is owned by some node other than the page owner.

First consider the case where other blocks of the same page are present in node *R*. Since *R* has other blocks of the page, the Shared Pages Table (SPT) entry

for the page on node *R* identifies the owner of the page. *R* sends a read request *RREQ* to the page owner (*P*) identified in the SPT entry. *R* changes its local state of the block to *ABS_RDNET* to mark the pending read request. On receipt of the *RREQ*, the page owner checks its SPT to determine if it owns the requested block. A block is owned locally if the local state of the block has access_permit *OEX* or *OSH*. If the block is owned by another node, the page owner looks up the identity of the block owner in its copy directory and forwards the request to the block owner (*B*). When the block owner receives the request, it checks its SPT to verify that it owns the block, updates the local state of the block to *OWNSH*, adds *R* to its copy directory, and sends the block data in a *RDCPY* message to *R*. When *R* receives the *RDCPY*, it updates the local block state to *RDONL* and allows the local processor to complete the read access. This flow of the read request is shown in Figure 4.16.

If the page owner *P* owns the block itself when it receives the read request from *R*, it can service the request itself. In this case, *P* does not forward the read request to any other node. *P* adds *R* to its copy directory and sends a read copy of the block (*RDCPY*) to *R*.

If *R* does not have any block of the page in its node memory, it cannot identify the owner of the page. In this case, an initial phase is needed to query the default owner for the page owner identity. *R* allocates a PPT entry for the page and marks its state to indicate a read request is in progress. *R* sends the *RREQ* to the default owner *D* of the page (the default page owner is a static function of the page number). When the default owner receives the *RREQ*, it looks up the page in its PPT. If the page is not present in the default owner (no PPT entry) it looks up its

Migrated Page Table (MPT) to see if the page is owned by any node. If the page is present in the default owner (PPT entry present) the SPT entry for the page is accessed to identify the page owner. Three outcomes are possible from the access of the MPT or the SPT — the MPT may indicate that the page is not owned by any node, the SPT may indicate that the page is owned by the default owner itself, or either the MPT or the SPT may indicate that the page is owned by some node other than the default owner. When the page is not owned by any node, the page is available only on disk. In this case, the entire page is read from disk by the default owner, and sent in a *PAGE* message to the requester *R*. The default owner also updates its MPT to indicate that *R* owns the page. On receipt of the *PAGE* data message, *R* changes its PPT entry state to private. When the default owner is also the page owner, the protocol continues as previously described (Figure 4.16).

Figure 4.17 describes the case when *R* does not have other blocks of the page, and the page is owned by some node other than the default owner. The default owner determines the identity of the page owner from its MPT or its SPT and sends the identity of the page owner node *P* in a *PGOWN* message to the requester *R* (bilateral exchange). When *R* receives the *PGOWN* message, it checks its PPT entry which indicates that a read request is in progress. *R* then sends a *RREQ* request message to the node *P* identified by the *PGOWN* message. Starting from this point, the transaction progresses as shown in Figure 4.16 up to the point where the block owner sends the block in a *RDCPY* message to *R*. When the *RDCPY* reply arrives at *R*, an SPT entry is allocated (since the page is shared) for the page in *R*. *R* sets the state of this block to *RDONL* and sets the state of all other blocks to *ABSENT* in the SPT entry. *R* also records the identity of the page owner in the

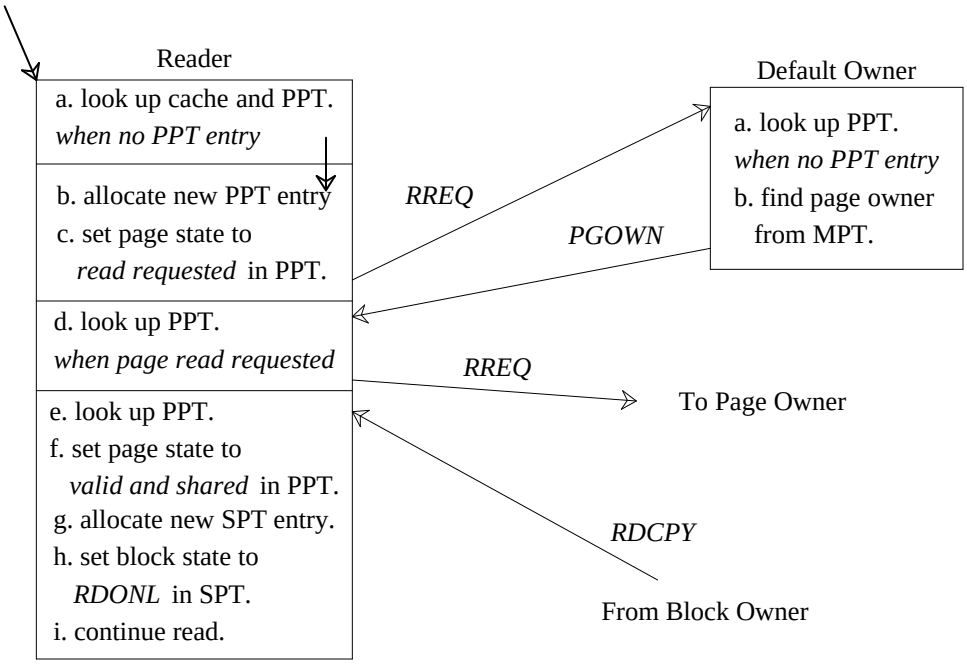


Figure 4.17: Read access when page owner is not known locally. This shows the flow of the protocol to satisfy a read miss by a local processor on node *R*, when *R* has no block of the page. *R* does not know the page owner, and requests the default owner who send the page owner identity in a *PGOWN* message. Subsequently, the protocol can flow as shown in Figure 4.16. When the *RDCPY* is received, an SPT entry is allocated in *R*. The page owner is marked in the SPT entry, the state of the block set to *RDONL*, and the states of all other blocks of the page set to *ABSENT*.

SPT entry.

When the page owner itself owns the block, clearly the request need not be forwarded anywhere. The page owner services the request and sends a *RDCPY* with the requested block to *R*. We illustrate this in Figure 4.18. It is possible for the page or block ownership to move while the read request is in transit. In this case, a read request can arrive at a node that was previously the owner after the

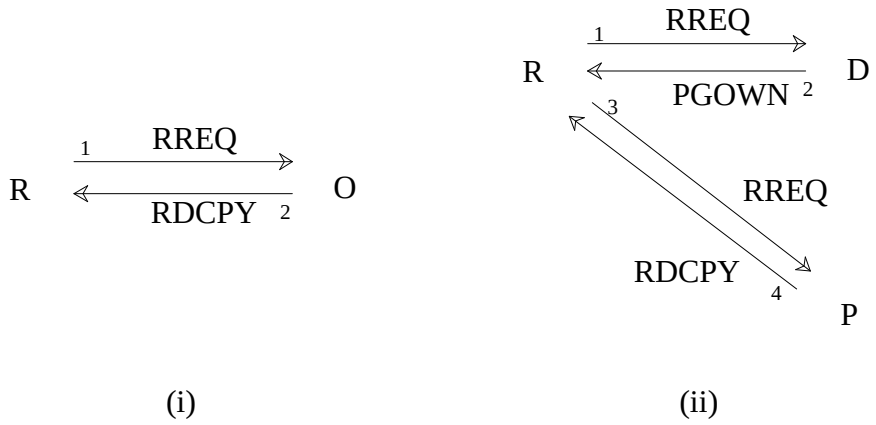


Figure 4.18: Additional protocol flow sequences for read requests. *R* is the read requester, *P* is the page owner, *D* is the default owner, and *O* represents either a page owner or a block owner. Protocol can flow as in case (i) in two situations — when *O* is the page owner and the block owner; or when *R* is the page owner and *O* is the block owner. Case (ii) is possible when *R* has no block of the page, and the page owner is also the block owner. Other flows are also possible, including those when the ownerships move.

node relinquished ownership. In such situations, the node receiving the request forwards the request to the default owner and the protocol flow is repeated. A pathological case where a read request is endlessly re-issued trying to locate the current owner of the block is possible. This can occur if the ownership of the block or page moves repeatedly in such a manner that the read request always reaches a node after it relinquishes ownership. However, this pathological case is unlikely (a similar scenario is also possible in DASH [Leno90] and FLASH [Kusk94]).

In our discussion thus far, we have discussed read requests which are serviced by the page owner and block owner as soon as the request is received. The page owner or block owner may not be able to service the read request immediately if a write access is in progress for the block, or if the page is being replaced. A write

access in progress is indicated by the block states *OWNSH_NET*, *OWNSH_XFR* and *ABS_LOCK*. A page that is being replaced is indicated by the PPT meta-state bits associated with the page replacement mechanism (Section 4.3.2). The read request cannot be satisfied in both these cases since it conflicts with the write access and the page replacement. The write access will complete only after exclusive ownership of the block is acquired. The page replacement will complete only after exclusive ownership of all blocks of the page (including the block being requested) is acquired. If a read copy is sent, these transactions will not complete since all read copies would not have been invalidated. Under these situations, the owner sends a *RETRY* message to *R* requiring it to retry the request.

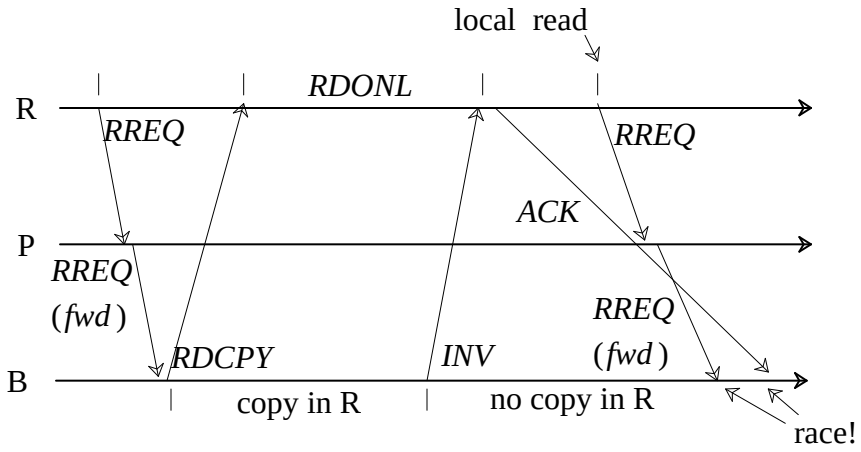


Figure 4.19: Race between *ACK* and *RREQ*. With the use of forwarding in servicing *RREQ*, a *RREQ* can race an *ACK* sent previously since they follow different paths.

The forwarding of the read request by the page owner introduces race conditions in the protocol. Consider the case where node *R* has its read copy of a block invalidated by the block owner *B* and subsequently requests a read copy of the block. Figure 4.19 shows the paths taken by the acknowledgement message

ACK and the read request *RREQ*. The read request *RREQ* is forwarded by the page owner *P* while the acknowledgement message *ACK* is sent directly to the block owner *B*. While the *ACK* message is sent from *R* before the *RREQ*, the *RREQ* can arrive at *B* before the *ACK*. (With bilateral exchanges, *P* would identify *B* to *R*, and *R* would directly send the *RREQ* to *B*, which cannot race the previous *ACK*, since they follow the same path).

In our implementation, the block owner handles the race situation between the *ACK* reply and the subsequent *RREQ* request by refusing to service any *RREQ* requests (by sending a *RETRY* reply) received when it is waiting for acknowledgements. As we described in Section 4.1, the information that the block owner is waiting for acknowledgements is maintained in its Pending Requests Cache. Since the Pending Requests Cache does not maintain the identity of nodes that must send acknowledgements, this mechanism cannot identify whether a *RREQ* request that is received is sent from a node that must send an acknowledgement (which indicates a race). When the *RREQ* is indeed sent by a node that must send the acknowledgement, this mechanism handles the race condition. This solution to the race problem may appear to be an unnecessary overhead whenever the *RREQ* request is received from a node other than one of the nodes that must send acknowledgement. However, even in such situations, it is not an overhead when the block owner had issued the invalidations to satisfy a previous write request. Read requests received while a write is in progress would anyway be denied service to allow the write access to complete. The mechanism incurs unnecessary overhead only when the block owner had invalidated read copies to evict them from the copy directory and a request is received from a node

other than the nodes whose copies were evicted. This is likely to be an infrequent event.

A race condition is also possible between an *ACK* message and a *CPINV* message. Suppose a node *R* evicts its read only copy of a block when the owner of the block, *B*, is concurrently invalidating all read copies of the block to satisfy a write access. A *CPINV* message notifying the block owner of the replacement will be sent by *R* to the page owner since *R* does not know the identity of the block owner. As with read requests, when the page owner receives the *CPINV* message, it forwards it to the block owner *B*. When *R* receives the *INV* message from *B* after it had sent the *CPINV* message, *R* sends an *ACK* message acknowledging the invalidate directly to *B* (in our implementation, an *ACK* message is sent to an *INV* message even when the block is not already present in the node — to support invalidations that are broadcasted). Since the *CPINV* and the *ACK* follow different paths to *B*, the *ACK* message can race the *CPINV* message and arrive at *B* first. However, this race is not a problem for our protocol. The *CPINV* messages requires that *R* be dropped from the copy directory for the block in *B*, which was already accomplished by the the invalidation. Hence, the *CPINV* is ignored. In general, when a block owner receives a *CPINV* message from a node that is not indicated in its copy directory, the *CPINV* message is ignored.

4.3.4.2. Anatomy of a Write Request

A processor's write access is a hit in the local node if the access_permit component of the local state of the block (Section 4.3.2) is *OEX*. The processor's write access will miss in the local node if the local state of the block being accessed does not permit write access, or if the block being accessed is not available locally. Write access must be requested from remote nodes in these cases to service the write miss.

The complete flow of protocol actions needed to satisfy a write request naturally depends (as with read requests) on several factors such as whether the requesting node *W* knows the identity of the page owner and whether the page owner is also the owner of the required block. In the rest of this sub-section, we discuss the flow of protocol actions in several cases including: 1) when *W* is also the owner of the block, 2) when *W* is also the page owner but not the owner of the block, 3) when *W* has other blocks of the page, 4) when *W* has no other blocks of the page and the page is on disk and 5) when *W* has no other blocks of the page and the page is owned by some node.

The write-miss scenario is simplest when the node that issues the write access is also the owner of the block. A node that owns a block will miss on a write access to the block if the block is owned shared (access_permit component of the local state of the block is *OSH*). In this case, the node must invalidate all read copies of the block in the system to service the write. The block owner first changes the local state of the block in its Shared Page Table (SPT) to *OWNSH_NET* indicating that a local write is in progress. The block owner then looks up the copy directory and an invalidate message *INV* is sent to each node that

has a read copy of the block. The write request is marked in the Pending Requests Cache on *W* (see Section 4.1). The Pending Requests Cache entry identifies the number of acknowledgements that must be received before the write access can be granted. When a node holding a read-only copy of a block receives an invalidate (*INV*) message, the local read copy is invalidated by setting the access_permit of the block state to *ABS*. An *ACK* reply is sent by the node receiving the invalidate (*INV*) message to the block owner, which sent the invalidate. Once all *ACK* messages (as indicated by the entry in the Pending Requests Cache) have been received by the block owner *W*, *W* sets the local state of the block to *OWNEX* and allows the write access to proceed.

When the node issuing the write *W* is not the owner of the block, the write request must be serviced by the page owner. When *W* is itself the owner of the page (but not the owner of the block), *W* can service the write request itself. First, in the SPT entry of the page in *W*, the block is marked to be in state *ABS_WRNET* if it was previously absent, and to be in state *RDONL_NET* if a read copy was present previously. The page owner *W* then uses its copy directory to determine the identity of the block owner. *W* then sends a write-back request *WRBCK* to the block owner (*B*). When the block owner *B* receives the write-back request, if the block is owned exclusively (*OWNEX*), *B* can immediately write the block back to the page owner *W* using a *BLKBCK* message. *B* then changes its local state of the block to *ABSENT*. When *W* receives the *BLKBCK* message, the access_waitlist component of the local block state indicates that a local write is in progress. Hence, the local block state is set to *OWNEX* and the local write access is allowed to continue.

In the scenario discussed above, where the node *W* issuing the write access is the page owner but not the block owner, the state of the block in the block owner *B* is *OWNEX*. The state of the block in the block owner *B* can also be owned shared (*OWNSH*). In this case, unlike the previous scenario, the *BLKBCK* message cannot be sent by the block owner until all read copies of the block have been invalidated. When the block owner *B* receives the *WRBCK* message, it looks up the copy directory and sends an invalidate message *INV* to each node that has a read copy of the block. The local state of the block in *B* is modified to *OWNSH_XFR* to indicate that a remote write is in progress and that the ownership of the block will be transferred away once all acknowledgements are received. Once all *ACK* messages have been received (identified using the Pending Requests Cache as previously discussed) by *B*, *B* writes the data of the block back to the page owner *W* using the *BLKBCK* message. At this point, *W* sets the state of the block to *OWNEX* (as before) and allows the write access to continue.

When node *W* is neither the block owner nor the page owner, the write request must be sent to the page owner for service. If some other blocks of the page being written to by *W* are present locally on *W*, the identity of the page owner is available in *W*'s Shared Pages Table (SPT). In this case, a write request is sent to this page owner *P*. In the SPT entry of the page in *W*, the block is marked to be in state *ABS_WRNET* if it was previously absent, and to be in state *RDONL_NET* if a read copy was present previously. When the page owner receives the write request, it looks up the local state of the block in its SPT. If the page owner possesses the block in the exclusive owned state (*OWNEX*), the block and ownership are immediately sent by a *WDATA* message to *W*. The page owner then sets its copy

directory to indicate that *W* is the block owner and changes the local block state to *ABSENT*.

If the page owner *P* possesses the block in the owned shared state (*OWNSH*) when the write request is received from *W*, read copies of the block must be invalidated before ownership is transferred. At the page owner, the access_waitlist of the block state is set to *LOCK*, indicating that a remote write is in progress. The page owner looks up its copy directory and invalidate messages *INV* are sent to all nodes with read copies. An entry is also added to the Pending Requests Cache in the page owner to indicate the number of acknowledgements expected for the invalidation. The information about the original remote write request including the identity of the requester (*W*) is recorded in this entry. Once all acknowledgements are received, the page owner sends the block and its ownership to the requester *W* with a *WDATA* message (the page owner determines that node *W* issued the write request from the information previously recorded in the Pending Requests Cache). The page owner then sets its copy directory to indicate that *W* is the block owner and changes the local block state to *ABSENT*. As before, when *W* receives the *WDATA* message, the local state of the block in *W* is set to *OWNEX* and the write access is allowed to complete.

Now, consider the scenario when the node issuing the write *W* sends the write request to the page owner *P* and the block accessed by the write is not owned by the page owner. In this case, when the page owner receives the write request, it uses its copy directory to determine the identity of the block owner. The page owner then sends a write-back request *WRBCK* to the block owner (*B*). At the page owner, the access_waitlist of the block state is set to *LOCK*, indicating that a

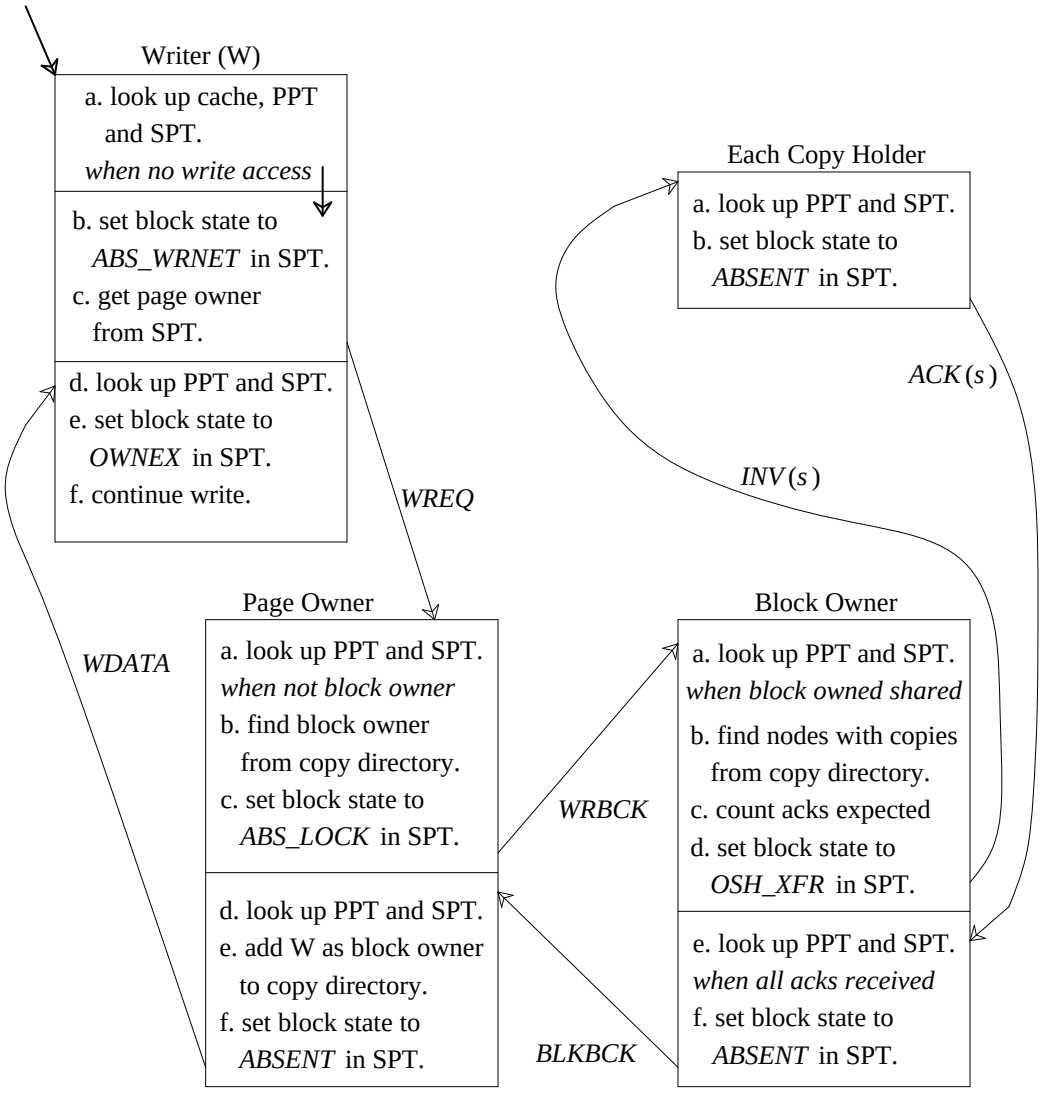


Figure 4.20: Anatomy of a write request. This is one possible flow of the protocol to satisfy a write miss by a local processor on node *W*. *W* does not have the block required by the write access, but has other blocks of the page. Hence, the SPT entry in *W* identifies the page owner. The block is owned by a node other than the page owner. Read copies of the block are present on other nodes.

remote write is in progress. An entry is added to the Pending Requests Cache in the page owner indicating that a response is expected for the write-back request. As before, the information about the original write request including the identity of the requester (*W*) is recorded in this entry. When the block owner receives the *WRBCK* request, the block must be written back to the page owner. As discussed in the previous cases, if the block is owned exclusively (*OWNEX*), the block owner can write the block back to the page owner immediately. If the block is owned shared (*OWNSH*), the block owner writes the block back to the page owner after all read copies have been invalidated. When the page owner receives the *BLKBCK* message from the block owner, since the access_waitlist component of the local block state is *LOCK*, the block and its ownership are sent to the requester *W* with a *WDATA* message. The page owner determines that node *W* issued the write request from the information previously recorded in the Pending Requests Cache. The page owner then sets its copy directory to indicate that *W* is the block owner and changes the access_waitlist component of the local block state from *LOCK* to *NONE*. When *W* receives the *WDATA* response, it changes the block state in its SPT to exclusively owned (*OWNEX*), allowing the local processor to complete its write access. This flow of a write access is shown in Figure 4.20. It should be noted that all the inter-node exchanges in this case are bilateral.

When the requester *W* does not have any block of the page, *W* sends the write request to the default owner *D*. *W* allocates a PPT entry for the page. *W* sets the state of the PPT entry to *WRNET* to indicate that a write request has been issued (Section 4.3.2). When the default owner receives the *WREQ*, it looks up the page in its PPT. If the page is not present in the default owner (no PPT entry) it looks up

its Migrated Page Table (MPT) to see if the page is owned by any node. If the page is not owned by any node, the default owner reads the page from disk and sends the entire page in a *PAGE* data message to node *W*. The default owner updates its MPT to indicate that *W* owns the page. When *W* receives the *PAGE* message, it marks the page private in its PPT.

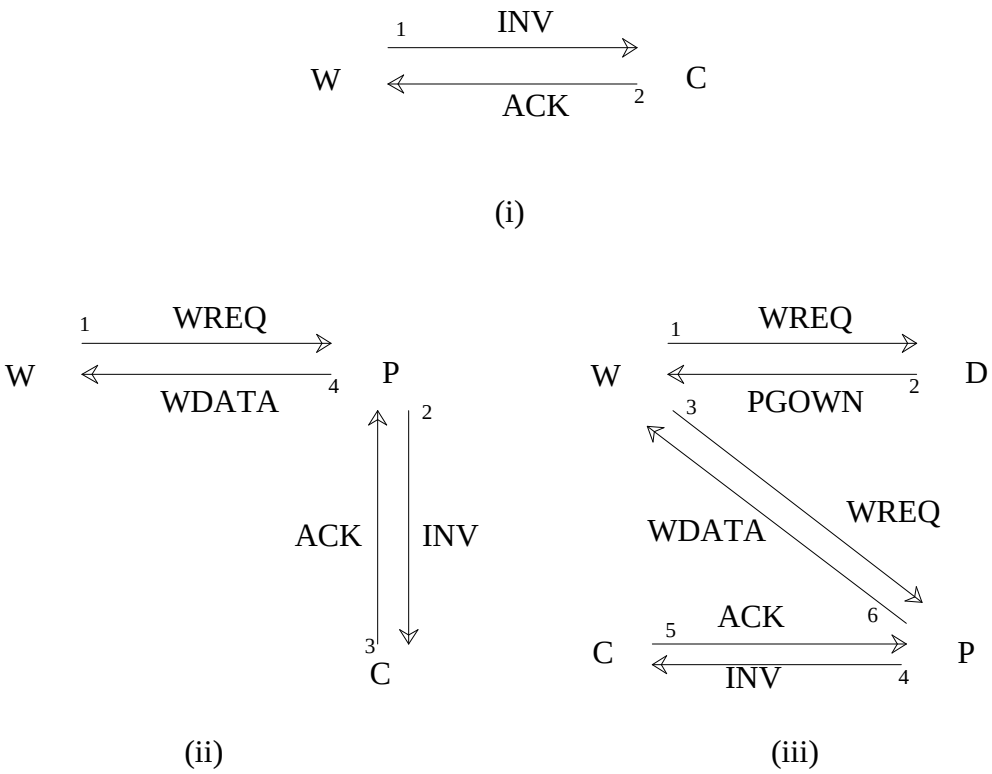


Figure 4.21: Additional protocol flow sequences for write requests. *W* is the write requester, *P* is the page owner, *D* is the default owner, and *C* is some node that has a read copy of the block. Protocol can flow as in case (i) when the *W* is the block owner. Case (ii) will occur when the page owner is also the block owner. Case (iii) also occurs when the page owner is also the block owner, but when *W* has no block of the page. Other flows are also possible, including those when the ownerships move.

Another possible write-miss scenario is where the requester *W* does not have

any block of the page and the page is owned by some node other than the default page owner. In this case, the default page owner sends a *PGOWN* reply to *W* identifying the page owner *P*. When *W* receives the *PGOWN*, it checks its PPT entry. Since the PPT entry indicates a pending write request, *W* sends a *WREQ* to the page owner *P* identified in the *PGOWN* message. After this point, the protocol flows exactly as described in Figure 4.20 (where the node requesting the write knows the identity of the page owner) up to the point where *W* receives the *WDATA* message. When *W* receives the *WDATA* message, it allocates an an SPT entry for the page. *W* sets the state of the block in the SPT to *OWNEX*, and sets the states of all other blocks of the page to *ABSENT*. *W* also marks the identity of the page owner in the SPT entry. This flow of the write access and a few other flows are shown in Figure 4.21.

In the above discussion, we have covered cases where the write request is serviced once it is received. Under some situations, it may not be possible to service the write request when it is received. A write request can be denied service if the page owner has an outstanding read request for the block. When a read request for the block is in progress in the page owner, the access_waitlist component of the local state of the block in the page owner is set to *RDNET*. Since we support only one outstanding access on a block at a time, the page owner will send a *RETRY* message to the node requesting write access. A write request can also be denied service when another write request is already in progress for the block (access_waitlist of *WRNET* or *LOCK* in the local block state), or if the page is being replaced in the page owner (identified by the PPT meta-state bits associated with the page replacement mechanism). As we discussed in the context

of read requests that are denied service (Section 4.3.4.1), the write request will be denied service so that the on-going write access or on-going replacement can complete. A write request is denied service by the current block owner if the current block owner is writing to the block. In this case, the current block owner sends a *RETRY* message to the page owner. The page owner must retry the *WRBCK* request later. All other cases when a write request is denied occur only at the page owner. The page owner sends a *RETRY* message to the node issuing the write request *W* in these cases. The write request must be retried by *W*.

4.3.4.3. Anatomy of a Replacement

A replacement occurs when there is a need to release resources allocated to a previous operation in order to allow a new operation to proceed. For the coherency protocol, the relevant resources are entries in the various tables and the local memory on each node. The tables where entries are dynamically allocated include the Present Pages Table (PPT), the Shared Pages Table (SPT) and the copy directory. An entry is allocated in the PPT when an operation requires a new page frame to be allocated (e.g., an access to a new page). An operation allocates an entry in the SPT when the operation causes the sharing of a new page. When a new node is allowed to share a page, a new entry must be allocated for the node in the copy directory. These operations can be blocked when entries needed by the operations are not available for allocation in these tables. To allow these operations to continue, space must be freed from these tables by replacing existing entries. We discuss the mechanism of PPT entry replacement, SPT entry replacement, and copy directory entry replacement in this sub-section.

When a node *N* requires a page frame to allocate a new page, a page must be replaced from its local memory. This frees up the PPT entry for use by the new page. The page to be replaced is chosen randomly. If the page randomly chosen is pinned from being replaced (e.g., due to outstanding requests), another page is chosen. The page that is picked for replacement can be private or shared (see Section 4.1). When a page is picked for replacement, its PPT entry is marked by setting the appropriate meta-state bit to indicate that the page is in replacement (see Section 4.1.1). From this point, the mechanism of replacement must clearly be different for replacing private and shared pages. In our implementation, the mechanism of replacing a shared page also depends on whether the page is owned by a remote node or by the local node *N*. We discuss the mechanism of replacement for each of these three cases: replacing a private page, replacing a shared page that is not owned by the node replacing the page and replacing a shared page that is owned by the node replacing the page.

Page replacement is simple when the page being replaced is currently private to the node *N*. The private page is written back to the default owner using a *PGBCK* message and the PPT entry is immediately reclaimed. When the default owner receives the *PGBCK* message, it writes the page to disk. The default owner also updates its Migrated Page Table (MPT) to indicate that the page is not owned by any node.

When a node *N* replaces a page that is owned by a remote node, *N* will relinquish the ownership of any blocks that it owns as well as any local read copies of blocks of the page. We discuss the mechanism of relinquishing ownership and then discuss the relinquishing of read copies. To relinquish block ownership, the

node must write the block back to the page owner. *N* writes back every block that it owns exclusively (*OWNEX*) to the page owner using a *BLKBCK* message immediately. Node *N* can write back any block that is locally in the owned shared state only after it acquires exclusive ownership of the block. The local state of each block that is currently in the owned shared state (*OWNSH*) is changed to *OWNSH_XFR* to indicate that exclusive ownership is being acquired to return the block to the page owner. For each of these blocks, node *N* sends invalidate messages (*INV*) to nodes that have read copies indicated in the copy directory. Once all acknowledgements for the block are received by *N*, *N* writes the block back (*BLKBCK*) to the page owner and changes the local state of the block to *ABSENT*.

Node *N* relinquishes its read copies of blocks of the page being replaced by changing the local state of such blocks to *ABSENT*. For each of these blocks, *N* sends a *CPINV* message to the page owner indicating that the local copy of the block was invalidated. When the page owner receives the *CPINV* message from *N*, the page owner determines the owner of the block (from the local state of the block or by accessing the copy directory). When the block is owned by the page owner itself, the page owner removes *N* from the copy directory for the block. If the block is not owned by the page owner, the page owner forwards the *CPINV* message to the owner of the block. When the block owner receives the *CPINV* message, node *N* is removed from the copy directory.

All blocks of the page are relinquished by *N* when the local state of each block of the page is *ABSENT*. This completes the replacement of the page (which was shared but not owned by the node replacing the page) At this point, the PPT

(and the SPT entry) entry associated with the replaced page is free to be reused.

Now, consider the scenario when the page being replaced to free up a PPT entry is owned by the node replacing the page (*N*) itself. In this case, the replacing node *N* first makes the page private to itself by evicting the page from all nodes that share the page. Subsequently, *N* can free the PPT entry by writing the entire page back to the default owner. *N* evicts the shared page from all nodes sharing the page by acquiring exclusive ownership of all blocks of the page. We describe the mechanism below.

When replacement of a shared page owned locally is initiated in node *N*, it modifies the local state of each block of the page to indicate that exclusive ownership is being acquired for the block. The local state of blocks in the *ABSENT, RDONL* and *OWNSH* states are changed to *ABS_WRNET, RDO_WRNET* and *OWNSH_NET* respectively. *N* can acquire exclusive ownership of these blocks by sending individual *WRBCK* messages to the owner of each block of the page that is not owned locally, and by individually invalidating (*INV*) each read copy of every block that is owned shared locally. As we mentioned in Section 4.3.3, to optimize performance, our scheme employs an alternative mechanism using the page invalidate (*PGINV*) message.

When *N* replaces a shared page that it owns (i.e., *N* is also the page owner), a page invalidate (*PGINV*) message is sent to each node that is indicated in the copy directory for the page in *N*. The nodes indicated in *N*'s (page owner's) copy directory are nodes that own any block of the page and nodes that possess a read copy of any block owned by the page owner *N*. Node *N* maintains the count of acknowledgements expected in its Pending Requests Cache. When a node receives

a *PGINV* message, it performs three actions. First, it drops its read copies of any block of the page — the local state of each block of the page that is in the *RDONL* state is changed to *ABSENT*. Second, the node receiving the *PGINV* message acquires exclusive ownership of any blocks of the page that it owns so that the block may be written back to the page owner *N*. The node writes back all blocks which are locally in the *OWNEX* state to the page owner using *BLKBCK* messages and changes the local state of these blocks to *ABSENT*. The node changes the local state of blocks in the *OWNSH* state to *OWNSH_XFR*, and sends invalidate messages (*INV*) to all nodes with read copies of the block. When the node has received all acknowledgements (*ACK*) for the invalidation, it writes back the block to the page owner (*BLKBCK*) and changes the local state of the block to *ABSENT*. Lastly, when the node receiving the *PGINV* message has dropped all its read copies and written back all blocks that were owned locally, it sends an acknowledgement message (*ACK*) to the page owner *N* confirming the invalidation of the page.

The shared owned page being replaced by *N* is private to *N* when it has received all acknowledgements for its page invalidation and all blocks of the page have been written back (in a FIFO network, the *BLKBCK* messages would all be received before the acknowledgements are all received). *N* can free the PPT entry associated with the page by writing the page back to the default page owner using the *PGBCK*. As discussed previously (in the context of replacing private pages), the default owner will write the page to disk and update its MPT to indicate that the page is not owned by any node.

An SPT entry must be evicted from a node when no free SPT entries are available on that node, and an operation requires a new SPT entry to be allocated in

that node. To free an SPT entry in node *N*, a page that is currently shared by *N* must be replaced. A shared page is chosen for replacement at random (by choosing an SPT entry to evict at random). When a candidate SPT entry has been chodes, the corresponding PPT entry of the page is identified from the page frame number field in the SPT (see Section 4.1.1). The appropriate meta-state bit in the PPT entry of the page is set to mark that the page is being replaced to free the SPT entry. As we described in our discussion of PPT replacements, the shared page chosen for replacement may be owned by the local node that is replacing the page (*N*) or may be owned by other remote node. When the page being replaced is not owned locally, *N* relinquishes ownership and read copies of all blocks of the page. The replacement mechanism in this case is exactly as we described for PPT replacements.

When the page being replaced to free up an SPT entry is owned by *N*, *N* will evict the page from all other nodes by acquiring exclusive ownership of all blocks of the page. The page is evicted from other nodes using the same mechanism described for PPT replacements. However, once *N* acquires exclusive ownership of all blocks of the page, the page is not discarded. The state of the page is changed to private instead of writing the page back to the default page owner (the PPT meta-state bits identify that the page is being replaced to free an SPT entry and not to free the PPT entry). Since an SPT entry need not be maintained for a private page, the SPT entry previously used by the page is free for reuse.

As we mentioned earlier, entries of the copy directory may also need to be evicted. When a new node shares blocks of a page from a node *N*, it must be indicated in the copy directory for the page. When no entry is available to add the

node in the copy directory of the page on *N*, an entry must be evicted. In our Block Map per Processor (BMpP) copy directory, each entry of the copy directory identifies a specific node and the set of blocks of the page shared by that node. When a specific entry is chosen for eviction (randomly), the victim node identified by the entry (*V*) is forced to relinquish the blocks indicated in the entry. This frees the copy directory entry.

To force *V* to relinquish the blocks, node *N* must first determine which of the blocks marked in the copy directory entry of *V* are owned by *V* and which blocks are owned by *N* itself. This is determined by accessing the local state of these blocks in *N*. For each of these blocks that is not owned by *N* (i.e., owned by *V*), *N* forces *V* to relinquish the ownership of the block by requesting it to write the block back. The access_permit component of the block state in *N* is set to *WRNET* to indicate that ownership is being acquired, and a *WRBCK* message is sent to *V*. For each of the blocks that is owned by *N* (i.e., not owned by *V*), *N* forces *V* to relinquish the read copy by sending it an invalidate message (*INV*). In this case, the local state of the block in *N* is not changed. The state associated with the *WRBCK* and *INV* messages that are issued is indicated in the Pending Requests Cache on *N*. The copy directory entry is free to be reused at this point. However, eviction is complete when *N* receives all blocks that must be written back and all acknowledgements that must be received.

4.4. Simulation of Our DSM System Design

Distributed shared memory systems and other large scale systems are composed of several complex sub-systems including processors, caches, coherence engines and interconnection network. The performance of these complex systems cannot be adequately evaluated using analytical models. Analytical models of these large systems are unwieldy and they cannot accurately capture the intricate interactions between the component sub-systems. Actual implementation or detailed simulation is necessary to realistically evaluate and characterize the performance of distributed shared memory systems.

Hardware-supported distributed shared memory systems have been implemented both in industry [Burk92] and in academia [Leno93, Agar95, Kusk94]. These implementation efforts have provided valuable experience in the design of DSM systems and serve as good platforms for studying the use of DSM systems. Such actual system implementations allow architectural ideas to be rigorously tested. However, they take multiple man-years to complete and are expensive, especially when the system being designed is a large scale multiprocessor.

These hardware implementations also have limited flexibility in accommodating multiple designs. Most of the functionality in these hardware implementations is built into hardware components that cannot be easily changed. For example, Alewife [Agar95] implements its base protocol in a hardware finite state machine. The base coherence protocol can be modified only by an expensive re-design of this chip. The MAGIC chip used in the FLASH architecture [Kusk94] supports some degree of flexibility by implementing the protocol code in user-

handlers which can be modified to change the protocol.

System-level simulations can be implemented more rapidly than hardware prototyping and offer greater flexibility in accommodating multiple designs. These simulations allow the system to be characterized over a range of system parameters (such as table sizes and queue sizes). Unlike prototype implementations, system simulations are not constrained to design for technology that is available at the time of evaluation. Simulations can be used to evaluate system designs for future technology, modeling components which are beyond the capability of current technology (e.g., faster memory or network, sophisticated processors etc). Simulations also ease the process of verifying and refining the design. Even hardware prototyping efforts use simulations in their initial phase to refine the design (e.g., the FLASH [Kusk94] implementation).

Simulation models can be based on trace-driven simulation or execution-driven simulation. With trace-driven simulation, the system is evaluated using address traces from a prior execution of the workload. The trace of addresses issued by the workload are collected prior to simulation when the workload executes on a different architecture. Execution-driven simulation is based on the actual execution of the workload on a simulated virtual machine. The virtual machine that is simulated is a close representation of the architecture being evaluated.

Trace-driven simulation is generally more efficient than execution-driven simulation since the workload is not executed. Since the workload is not executed during a trace-driven simulations, building these simulators is also easier than building execution-driven simulators. However, trace-driven simulations are likely

to be less accurate than execution-driven simulations for several reasons. Firstly, the address traces are influenced by the architecture of the machine on which the traces were generated. Second, the ordering of the accesses in the workload may be perturbed by the process of collecting the traces. Third, if the trace is simply a sequence of addresses, there is no information about time between accesses. In the absence of such timing information, trace-driven simulation cannot measure the real indicators of system performance such as application execution time, processor utilization or network throughput. Trace-driven simulation also has the disadvantage that the storage needed for maintaining large address traces is quite high.

We first implemented a simple trace-driven simulator of our DSM design, allowing rapid evaluation of some of our preliminary designs. Subsequently, we have implemented an accurate execution-driven simulator of our design that served both for performance evaluation and for experimental validation of our protocols. We describe these implementations in this section.

4.4.1. Implementation of a Simple Trace-Driven Simulator

We have implemented a simple trace-driven simulator of our hierarchical scheme to obtain initial validation of our claims about the effectiveness of the scheme. The simulator performs all accesses atomically. This is accomplished by atomically performing all the state changes throughout the entire system required for a particular block access. This simple design allowed the simulator to be built quickly without the complexities associated with the implementation of a complete distributed directory coherence protocol, or of an interconnection network. The

simulator implements sufficient detail to measure the hit ratio, number of coherence messages generated, and the usage of various mapping tables.

The simulator accurately implements all the mapping tables required by the hierarchical scheme: Present Pages Table (PPT), Shared Pages Table (SPT), Migrated Pages Table (MPT) and the copy directory. PPT and SPT entries have the same format shown in Section 4.1.1. However, since all accesses are atomic, we do not maintain access_waitlist as part of the block state in the SPT (Section 4.3.2). The simulator supports both Block Map per Processor (BMpP) and the Processor List per Block (PLpB) copy directory organizations. When there are no available entries in the PPT, SPT or the copy directory, a victim is picked randomly and evicted.

For each address in the input trace, the local mapping tables in the processor issuing the address are looked up. If the access can be satisfied locally, then the access is a ‘‘hit.’’ If the access cannot be satisfied locally, the mapping tables in all relevant remote nodes (page owner, block owner, read copy holders) are looked up and modified appropriately. The simulator does not implement the coherence protocol, but accurately counts the number of coherence messages that would have been sent for servicing remote accesses and evictions.

4.4.2. Implementation of an Execution-Driven Simulator

The execution-driven simulator of the hierarchically managed DSM was implemented on top of the Proteus [Brew92a] simulation environment. Proteus is a parallel architecture simulator that can simulate any generic MIMD multiprocessor. Different parallel architectures can be modeled and studied using the Proteus

simulation framework. We describe the Proteus simulation environment in Section 4.4.2.1. We also discuss the writing of applications for this environment as well as the compilation and execution of the applications. Subsequently, in Section 4.4.2.2, we discuss the implementation of hierarchically managed DSM on Proteus.

4.4.2.1. The Proteus Simulation Environment

Proteus models the system as a set of nodes interconnected by some network. Each node in the system is viewed as a collection of threads. The threads correspond to the different components of the node: processor, memory and network interface. The processor is ‘‘simulated’’ by a thread of application code executing directly on the simulation host hardware. The memory and network threads are simulated in software. The behavior of the memory and network threads are described in memory and network engines which can be modified to simulate systems with different memory architectures and systems with different interconnection networks.

The Proteus simulation kernel is essentially a threads package that manages the threads associated with each node of the system. The threads corresponding to the processor, memory and network on each node in the system are created at the start of simulation. The run-time simulation engine first schedules the simulation of the processor of node 0. To ‘‘simulate’’ the processor, the application is executed on the simulation host. During execution, the application generates memory-access and interprocessor-communication events which must be serviced by the memory, network and remote processor threads. The run-time simulation

engine schedules the different threads on the simulation host machine based on the time-ordered sequence of events (we will discuss the timestamping mechanism shortly). Each thread that is scheduled executes according to the behavior specified in its model and can generate events for other threads in the system. The executing thread returns control to the simulation engine when its task is completed or when it requires the service of another thread. The kernel includes a context-switching routine to allow control to be switched between the simulation engine and a thread.

The Proteus simulation environment includes a programming environment for application development as well as an operating system environment for run-time support. To support a shared memory programming environment, Proteus includes a preprocessor that extends the C language with new storage classes and operators that allow easy definition and use of shared memory variables. Proteus provides the application with primitives for reading from and writing to shared memory, test-test-and-set synchronization, interprocessor interrupts, and message passing. Proteus also supports barrier synchronization, which is implemented through interprocessor messages [Kopp95].

The Proteus kernel provides basic operating system functions required to run applications in a parallel environment. Proteus provides thread management routines that allow the application to dynamically create and destroy threads. Proteus also provides system calls to allocate and deallocate memory. Application threads executing on Proteus can allocate and deallocate regions of shared memory (local to the node or non-local) using these system calls.

In order to be executable in the Proteus simulation environment, an

application must be written to conform to Proteus' programming paradigm and must be compiled using the Proteus compilation preprocessor. The application must use the programming constructs provided in Proteus's extended C language to identify all shared variables and all accesses to the shared address space. All other variables and accesses in the application are viewed as private to each process. The application must allocate memory and perform application thread management using the primitives provided in Proteus. Processes of the application must synchronize and communicate with each other using the synchronization and communication primitives provided by Proteus.

The Proteus compilation phase preprocesses the application code to modify it in two ways. First, all shared address space accesses and all interprocess communication and synchronization events in the application source code are translated into calls to corresponding routines in the memory and network engines. Second, the assembly code generated from the preprocessed application source is instrumented with additional assembly instructions. The assembly code augmentation adds code to count the number of instructions executed, the number of private load accesses and the number of private store accesses during application execution. Each basic block in the application assembly is augmented with instructions that accumulate the number of instructions, number of private load accesses and number of private store accesses in that basic block to global variables that maintain their overall counts. These global variables are maintained independently for each node in the model. At any stage of the simulation, the value of these global variables for a specific node is the total count of instructions executed, private load accesses and private store accesses on that node until that

stage of application simulation.

Applications execute in Proteus under the control of the Proteus run-time simulation engine. The simulation engine initiates application execution by scheduling the processor thread of node 0 with a timestamp of 0. When a thread that is scheduled to run requires service from another thread (e.g., processor must access memory, memory must send a message on the network etc.), it generates an output event requesting the service and transfers control to the simulation engine. The thread generating the output event must timestamp the event appropriately. Processor threads timestamp their output events based on the number of instructions of application code that was executed since it was scheduled (Proteus assumes one processor cycle per instruction). Memory and network threads timestamp their output events based on the latency in generating the output events as modeled in their respective engines. The memory engine models the latency of various tables that must be accessed to service input memory events. The network engine models the latency in the network switches as well as the latency of network links. The simulation engine handles the task of scheduling the appropriate thread (processor, memory or network threads of any node) that can service the simulation event with the earliest timestamp. Application execution is complete when no thread is scheduled to run and no simulation events are pending.

Proteus advances the execution time of the application on each node through the latency of executing instructions which access only private data and through the latency of accessing shared data. The latency of executing instructions which access only private date is measured by counting the number of such instructions (each instruction is assumed to execute in one processor cycle). The latency of

accessing shared data is measured by the latencies computed in the memory and network threads that participate in satisfying the access. The process of servicing the shared access request is initiated by the memory thread on the node that issued the request. Subsequently, a sequence of network and memory events may be generated to send requests to remote nodes and to access memory tables in the memory threads of remote nodes. As described earlier, each memory and network thread timestamps their output event based on the latency of servicing their input event. Hence, the timestamp of the event that eventually grants the requested access incorporates the cumulative latency in the network and remote memories. The difference between the timestamps of the event requesting access and the event granting access measures the latency of accessing shared data. The overall execution time of the application is the time of the node with the highest time.

4.4.2.2. Implementation of Hierarchically Managed DSM on Proteus

When an application executing under Proteus accesses shared memory, the processor thread executing the application code issues a memory event which must be serviced by memory threads. The behavior of these memory threads must be described in a software memory engine module. We have developed a memory engine for Proteus that accurately models our hierarchically managed DSM.

The memory engine implements all the hierarchical management mapping tables — Present Pages Table (PPT), Shared Pages Table (SPT), Migrated Pages Table (MPT) and the copy directory — for each node to support address translation. The implementation faithfully models the inverted table implementation of the PPT. PPT entries have the same format presented earlier in

Section 4.1.1. However, the entries also maintain a pointer to the corresponding page frame of memory. The SPT is accurately implemented and SPT entries for shared pages are dynamically allocated. The SPT entries support the full set of access_permits and access_waitlists associated with block states (Section 4.3.2). They also maintain a pointer to a copy directory as required by the hierarchical management scheme. Additional fields are maintained in the SPT entry to accumulate statistics about the page during simulation. The MPT is also implemented for each node to track the current owner of all pages default owned by that node. Our copy directory implementation in the memory engine supports both Block Map per Processor (BMpP) and Processor List per Block (PLpB) organizations. In both cases, the copy directory is dynamically allocated. A copy directory is allocated for a shared page on a node only when information about the sharing of blocks need to be maintained on that node.

Since the PPT, SPT and the copy directory are bounded, as they would be in a real system, entries may need to be replaced from these tables. All replacements from the PPT, SPT and copy directory are random in our memory engine. While they are performed exactly as described in Section 4.3.4.3, the selection of a replacement victim does not consume any latency in our model. When a PPT entry is not available to service a local processor's request, the network request is not issued until a PPT entry is available. Request and Reply buffers (as shown in Figure 4.1 in Section 4.1) are used to buffer remote requests and replies when SPT entries or copy directory entries are not available to service them. However, in our implementation, the size of these buffers is unbounded. Also, we do not model the latency of accessing these buffers. Our implementation does not model the

presence of the Pending Requests Cache. The information contained in the Pending Requests Cache is maintained in the PPT, SPT and copy directory.

The pipelined implementation of the MMU control is not accurately modeled in the memory engine. Specifically, the model does not implement the pipeline stages in the MMU design shown earlier in Figure 4.1 of Section 4.1. However, the implementation models some degree of pipelining by allowing an access to be issued to the MMU tables when another access issued in a previous cycle may be in progress. The model does not implement a cache or a Translation Lookaside Buffer (TLB). However, as we will discuss in Chapter 5, the presence of a cache and a TLB can be approximately modeled by assigning appropriate latencies to accessing local data, the PPT and the SPT.

The memory engine also models the presence and access of a disk on each node. At the start of simulation, all pages reside on the disk of their respective default page owners. We use a first-touch policy to assign page ownership in our implementation. When a node receives a page from disk (read from disk and forwarded by the default owner as dictated by our coherency protocol) in reply to a read or write request, it is assigned ownership of the page.

The memory engine services all application accesses to the shared address space. The memory engine implements our distributed directory coherence protocol (described in Section 4.3) completely to service these accesses. All coherence protocol states and message types discussed in Section 4.3 are supported. If the block requested by an access on a node is available locally with sufficient access permission, the access is performed by the memory thread of that node without accessing the network. If the processor access cannot be satisfied

locally, messages are sent to remote nodes to acquire access. The processor is blocked until the access is satisfied (i.e., the processor thread of this node will not be scheduled until the access can be satisfied).

To service shared accesses that cannot be satisfied locally, memory threads exchange coherence messages with memory threads on remote nodes. These coherence messages are sent from a memory thread on one node to a memory thread on another node through several network threads which model the latency of sending the message through the network. A memory thread sends coherence messages to the network by generating a event that must be serviced by the network thread of the local node. A memory thread receives a coherence message from the network when the network thread local to that node generates an event for the memory thread delivering the coherence message.

The simulation events generated by the memory and network threads provide all the information associated with the coherence message. This information includes the identity of the node sending the message, the type of coherence message (Section 4.3.3), the address being accessed, the node that is the destination of the message, and the identity of the client node that is performing the memory access. When a request is forwarded in the coherence protocol (Section 4.3.4), the client field is used to identify the node that originally issued the shared memory access. When a coherence message includes data, a data field is also present.

The network engine implements the network as an interconnection of switching elements which communicate by exchanging simulation events. Messages are not partitioned into multiple packets (each packet is a complete message). The network thread corresponding to each switching element (node)

implements the routing logic and maintains, for each output port at that switching element, the earliest simulation time at which the output port is available. Network threads use the source and destination node identities in any incoming packet to determine the node next in the packet route and the output port to be used. The switching elements handle packets using the store-and-forward discipline. If the required output port is marked as available at a time earlier than the simulation time at which the incoming packet is received, the packet does not wait in the switching element buffer and is immediately transmitted through the output port. Otherwise, the incoming packet must be buffered and can be transmitted only at the time the output port becomes available. In either case, the time of transmission is determined and a simulation event forwarding the packet is generated for the network thread at the successor node. The timestamp of this simulation event is calculated as the sum of the time of transmission and the switching and link latencies of the packet (switching latency is a parameter and link latency is computed from the link bandwidth parameter and the size of the packet). The time at which the output port becomes available after transmitting this packet is updated likewise. The packet is routed to the destination node through a sequence of switching elements in the network. When the packet arrives at its destination node, the network thread at that node generates a simulation event that delivers the packet to the local memory thread.

Network packets support data structures that allow statistics about the link latency and queueing delay incurred by each packet to be accumulated. Network threads at each switching element in the path of a packet add the link latency and queueing delay incurred by the packet at that switching element to these data

structures. The value of these data structures at the destination node is the overall link latency and overall queueing delay incurred by the packet.

Chapter Five
Evaluation of Hierarchically Managed DSM

The hierarchically managed DSM system optimizes the management of shared data based on the assumption that only a small percentage of the shared address space of a process is actually shared at any time. This optimization can potentially enable the hierarchical scheme to achieve the performance advantage of uniform block-based schemes without incurring their large storage overhead (see Chapter 3). We have evaluated the performance of the hierarchical management scheme using trace-driven and execution-driven simulations. We discuss our performance evaluations and present results from these evaluations in this chapter.

Our initial studies evaluated the performance of the hierarchical scheme using trace-driven simulations. The trace-driven simulator does not implement the complete coherence protocol and executes accesses atomically (Section 4.4 in Chapter 4). By using address traces from real workloads with our trace-driven simulations, we obtained initial validation of our premises about sharing behavior in real applications.

Subsequently, we have evaluated the performance of the hierarchical scheme using execution-driven simulation. The execution-driven simulation includes a complete implementation of the coherence protocol and accurately models the behavior of the processors and the network. Since the execution-driven simulation also executes real applications (rather than using address traces from a prior execution on a different architecture), the results from this simulation more accurately reflect the performance of a real DSM system.

For both evaluation studies, the simulations also implemented the one-level scheme to allow the performance of the hierarchical scheme to be compared with the performance of the one-level scheme. Our simulations also provide the flexibility to study the effectiveness of the hierarchical scheme over a range of block sizes and mapping table sizes. We first describe the evaluation based on execution-driven simulation and its results in Section 5.1. In Section 5.2, we describe the results from our trace-driven simulations.

5.1. Evaluation based on Execution-driven Simulation

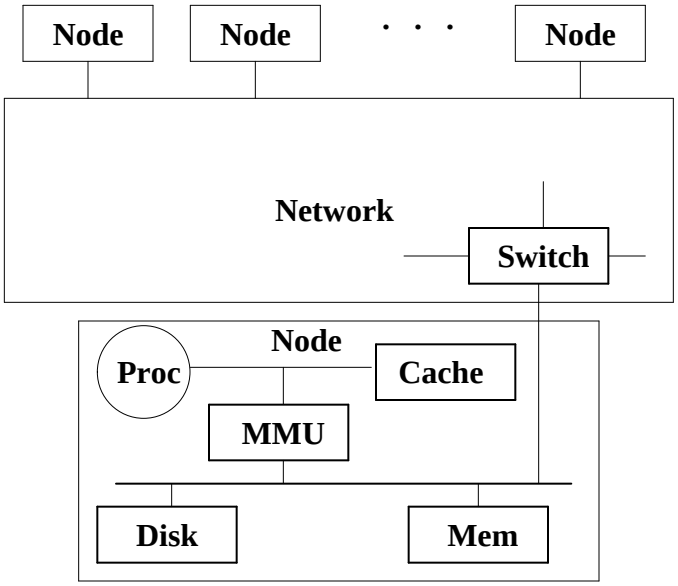


Figure 5.1: Model of the execution-driven simulation. A system with 64 nodes and a 2-D mesh network was simulated.

As described in Section 4.4.2 of Chapter 4, our execution-driven simulation is based on Proteus [Brew92b], which models the system as a number of nodes connected by an interconnection network (Figure 5.1). Each node model in the

simulated system is comprised of a processor, cache, memory, disk, a memory management unit (MMU), and an interface to the network. The processor is ‘‘simulated’’ by directly executing the application code on the simulation host. The cache, memory, disk and MMU on each node are simulated by a *memory engine*. The memory engine implements the mechanisms needed to access the shared address space (such as address translation, disk access, coherence protocol) and models their respective latencies. The network interface and the network are simulated by the *network engine*. This engine implements message sending, transmission across the network and reception.

We performed our evaluations for a system with 64 nodes. The behavior of the processor is emulated by executing all application instructions that would be executed by the processor directly on the host machine (Sun Sparc) on which we ran our simulations.

The network is modeled as a packet-switched bidirectional 2-D mesh (8×8). The network switches use infinite output port buffers. In Proteus, the bandwidth of the network links is parameterized as a function of the processor cycle frequency in the processor model. We used the network engine over a range of link bandwidths. For a processor operating at *P* MHz, we have varied the link bandwidth from *P* MB/s to *8*P* MB/s (link bandwidths from 200 MB/s to 1.6 GB/s for a 200 MHz processor, for example). Control messages sent through the network have a size of 5 bytes. The size of data messages is the size of the data plus 5 bytes.

The memory engine models the memory and disk on each node. The local disk controller component of the memory engine controls access to the disk. The disk controller models a latency of 1000 processor cycles to access data from the

disk, for all size of data accessed (with current technology, latencies of 1,000,000 cycles can be expected). With the hierarchical scheme, disk accesses are at the granularity of pages. However, with the one-level scheme, disk accesses are at the granularity of blocks. While a real system would access disk only at the granularity of pages, the one-level scheme implemented in our simulations does not include the page-level tables necessary for supporting page-level disk accesses. The implementation of page-level disk accesses with block-level tables is tricky since the eviction of a block to disk from one node necessitates the coordinated eviction of all other blocks of the page from all other nodes. In addition, if page-level disk accesses are supported without page-level tables, a node requesting a block of data may need to allocate an entry for each block of the page being accessed to map all the data received from disk. A practical implementation of the one-level scheme should extend it with page-level tables to support page-level disk accesses (making the scheme pseudo-hierarchical).

The memory engine also implements the memory management unit (MMU) which supports distributed shared memory. The MMU model implements all the mapping tables and our distributed directory coherence protocol for hierarchical management (the MMU also implements the one-level scheme which is described in the next sub-section). The MMU models the presence of a cache on each node in a simple way by assuming that a fixed proportion of shared accesses that hit in the local node's memory hit in the local cache. Our simulations used a cache hit rate of 97% and a cache hit time of one processor cycle. The MMU does not model the Translation Lookaside Buffer (TLB) in detail. However, the average latency of accessing the Present Pages Table (PPT) and the Shared Pages Table

(SPT) are estimated assuming that entries from the PPT and SPT are cached in the TLB. The average PPT access latency is estimated based on a raw PPT access time of 20 processor cycles, a TLB hit access time of one processor cycle and a TLB hit rate of 90% for PPT entries. The average SPT access latency is estimated based on a raw SPT access time of 20 processor cycles, a TLB hit access time of one processor cycle and a TLB hit rate of 85% for SPT entries. Table 5.1 tabulates the latency of accessing various tables in our simulation.

Access Latencies of Coherency Tables			
Table	Acronym	Latency	Comments
Present Pages Table	PPT	3 cycles	cached in TLB from memory
Shared Pages Table	SPT	4 cycles	cached in TLB from memory
Copy Directory	CoD	7 cycles	look-up is pipelined; updated in parallel with message sending
Migrated Pages Table	MPT	40 cycles	cached in memory from disk

Table 5.1: Table Access Latencies.

Our simulations of the hierarchical scheme simulated a page size of 1024 bytes (with varying block sizes) while our one-level scheme simulations simulated several page sizes from 16 bytes to 1024 bytes. Table 5.2 lists the sizes of the PPT, SPT and the copy directory (on each node) in our simulations of the hierarchical and one-level schemes. For the one-level simulations, the number of entries in the PPT, SPT and copy directory are tabulated for each page size that was simulated. At small page sizes, the number of entries in the PPT and SPT are higher since each entry maps a smaller fraction of the local memory. The total amount of memory mapped by the PPT and SPT on each node (product of the number of entries in the table and the page size) are also shown in Table 5.2 (columns labeled

‘‘Size of Memory Mapped’’) for each page size. As the table shows, the amount of memory available in each node (the size of memory mapped) is not the same for all page sizes simulated. At small page sizes, the amount of memory available is smaller in our simulations. This is because the large number of PPT and SPT entries required to map larger memory at small page sizes increases the memory requirements of the simulator posing a burden for our simulation host machine. In all our simulations, we used a full copy directory with 64 entries (last column of the table). The Migrated Page Table (MPT) was allowed to grow as large as necessary to maintain an entry for each page of shared data on disk.

Scheme	Page Size (bytes)	Present Pages Table		Shared Pages Table		Number of entries in Copy Directory
		Number of entries	Size of Memory Mapped	Number of entries	Size of Memory Mapped	
Hierarchical	1024	512	512 KB	128	128 KB	64
Onelevel	16	8192	128 KB	2048	32 KB	64
	32	4096	128 KB	1024	32 KB	64
	64	2048	128 KB	512	32 KB	64
	128	2048	256 KB	512	64 KB	64
	256	1024	256 KB	256	64 KB	64
	512	1024	512 KB	256	128 KB	64
	1024	512	512 KB	128	128 KB	64

Table 5.2: PPT, SPT and Copy directory sizes in simulations. Our hierarchical scheme simulations used a page size of 1024 bytes. The page size was varied from 16 bytes to 1024 bytes in our one-level scheme simulations.

Proteus measures the execution time of the application on the simulated architecture as the sum of the time spent executing local instructions and the time spent executing shared accesses. As described in Section 4.4.2 in Chapter 4, to

measure the time spent executing local instructions, the Proteus compilation process augments the assembly code of the application to count the number of instructions executed by the application. The Proteus compilation process also converts all accesses to shared data in the application to calls to the memory engine which services these accesses. The time spent executing shared accesses is the latency within simulator engine modules such as memory and network. During simulation, the cycles spent in the execution of application basic blocks and the cycles used up in simulator engine modules are both measured. In addition, the simulator engine modules are instrumented to measure several behavior and performance characteristics of the application and the system.

We describe our implementation of the one-level scheme and the characteristics of our benchmark applications in the following two sub-sections before presenting performance evaluation results from our simulations.

5.1.1. One-level Scheme Implementation

Our Proteus memory engine also implements the one-level scheme to allow the performance of the hierarchical and one-level schemes to be compared. The one-level scheme manages memory at a single granularity which can be as large as a page (Section 3.1.1), or as small as a cache block (Section 3.1.2). We evaluate page-level and block-level schemes by varying the size of the block in the model. In the rest of this subsection, we describe the key differences between the one-level scheme implementation and the implementation of the hierarchical scheme.

In the one-level scheme data is mapped and transferred at only one granularity (we refer to it as a *block*, whose size is variable). Hence, the coherence protocol

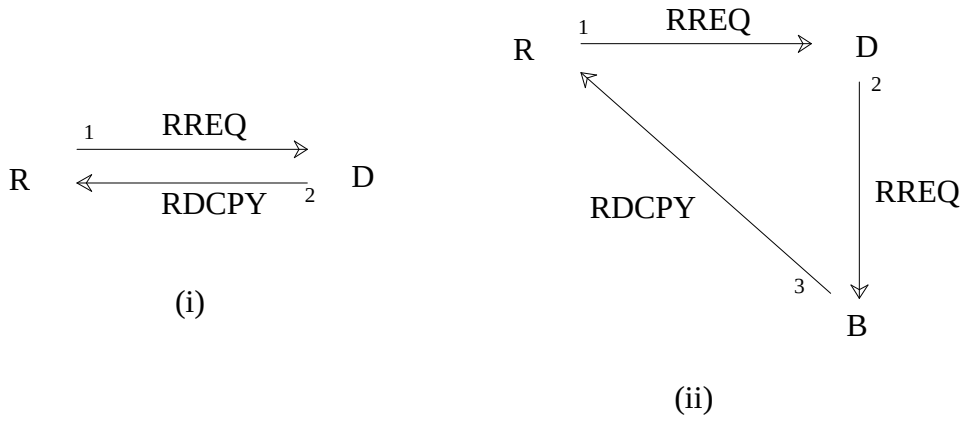


Figure 5.2: One-level scheme protocol flow sequences for read requests. *R* is the read requester, *D* is the default owner, and *B* is the block owner. Protocol can flow as in case (i) when the default owner is also the block owner, and as in case (ii) otherwise.

used for this implementation allows only one level of ownership. Every block of data in physical memory is owned only at the block level by a node referred to as the block owner. A statically assigned default owner for the block maintains the identity of the block owner. The default owner in the one-level scheme serves the function of both the default owner and the page owner in the hierarchical scheme.

The protocol transactions for read and write requests differ from the transactions in the hierarchical scheme primarily in the absence of a second-level page owner. When a node has a read or write miss, the request is sent to the default owner. The default owner identifies the current block owner using its MBT (Migrated Block Table). Read requests are forwarded by the default owner to the current block owner, who directly sends a reply to the requester. In case of a write request, the MBT entry is locked at the default owner. The default owner sends a message to the current block owner requesting it to write the block back. Upon receiving the request, the block owner, as in the hierarchical scheme, invalidates all

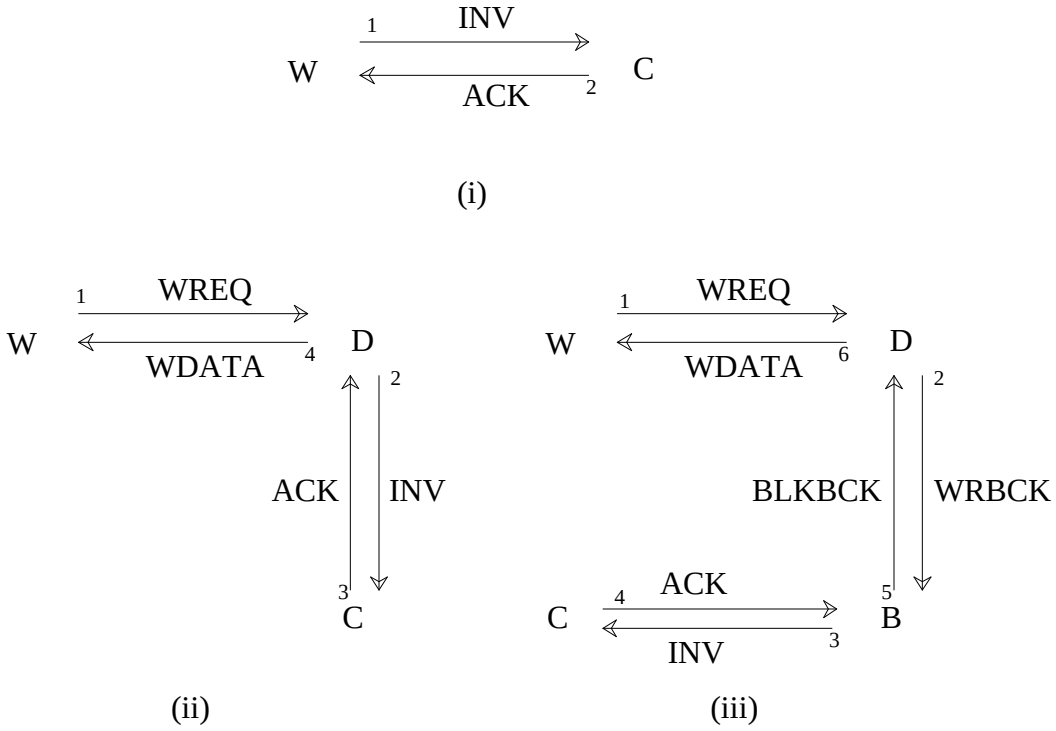


Figure 5.3: One-level scheme protocol flow sequences for write requests. *W* is the write requester, *D* is the default owner, *B* is the block owner, and *C* is some node that has a read copy of the block. Protocol can flow as in case (i) when the *W* is the block owner. Case (ii) will occur when the default owner is also the block owner. Case (iii) also occurs when the block is owned by a node other than the default owner.

read copies of the block. Once all copies are invalidated, the block is written back to the default owner. Upon receiving the write-back, the default owner updates its MBT and the block data and ownership are sent to the requester. We illustrate the typical protocol flows for reads and writes in Figure 5.2 and Figure 5.3 respectively.

The protocol message types and protocol states in the implementation are modified slightly to take advantage of some simplifications allowed by the one-level scheme. Since only one size of data is transferred, messages are not needed

to support both blocks and pages. Page-related messages such as *PGOWN*, *PGINV*, *PAGE* and *PGBCK* are not used in the one-level scheme implementation. A block that is absent in a node has no entry associated with it on that node in the one-level scheme (in the hierarchical scheme, an entry is present if some other block of the page is present). Hence, the block access_permit *ABS* is not used. The absence of an entry in the Present Block Table (PBT) provides the same information.

5.1.2. Application Benchmarks

Our execution-driven simulation results are based on the execution of three scientific applications from the SPLASH-2 application suite — Cholesky, LU and Radix [Sing92, Woo95]. These SPLASH-2 applications were ported to use the Proteus API for shared memory programming (Section 4.4.2 in Chapter 4) by the author of [Kopp95]. The port modified the application source code to explicitly identify accesses to the shared address space. Also, synchronization points in the applications were modified to use the test-test-and-set semaphore locks (which are implemented through reads and writes to the shared address space) and barriers (which are implemented through messages) provided by Proteus.

The Cholesky application performs parallel Cholesky factorization of a sparse positive definite matrix. A sparse matrix is factored into the product of a lower triangular matrix and its transpose. The program primarily performs the numerical factorization step of Cholesky factorization [Sing92], which determines the actual numerical values of the non-zero entries in the lower triangular matrix. The numerical factorization is efficient due to the use of supernodal elimination

techniques. In our simulations, the application processed the tk14.O input data set provided in the SPLASH-2 suite.

The LU kernel [Woo95] factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The dense 256×256 matrix is divided into a 32×32 array of 8×8 blocks to exploit temporal locality on sub-matrix elements. Block ownership is assigned using a 2-D scatter decomposition, with blocks being updated by the processors that own them. The kernel version is a less-optimized version that does not allocate elements of a block contiguously.

The Radix kernel [Woo95] implements an integer radix sort based on the method described in [Blel91]. The algorithm is iterative, performing one iteration for each radix r digit of the keys. In each iteration, a processor passes over its assigned keys and generates a local histogram. The local histograms are then accumulated into a global histogram. Finally, each processor uses the global histogram to permute its keys into a new array for the next iteration. This permutation step requires all-to-all communication. The permutation is inherently a sender-determined one, so keys are communicated through writes rather than reads. The application sorted 256 K keys with maximum key value 512 K using a radix of 1024 in our evaluations.

Table 5.3 summarizes the problem size solved by the applications and the amount of disk storage available to the application. The second column of the table specifies the size of the problem solved by the application. All shared data used by the application in solving this problem is mapped to the disks in the system (one per node). Pages of shared data are mapped across the disks in the system with a round-robin policy. During execution, the application must access the data

Application	Problem Size	Disk storage per node in simulation	Shared Data used by application
Cholesky	tk14.O	1 Mbyte	18.02 Mbytes
LU	256 × 256 matrix, 8 × 8 blocks	256 Kbytes	719 Kbytes
Radix	256 K keys, 512 K maximum key value, radix 1024	256 Kbytes	3.08 Mbytes

Table 5.3: Memory usage in application benchmarks. The third column presents the amount of storage available in the disk of each node in the simulation. The simulated system had 64 nodes. The last column lists the actual amount of memory used by the application for storing its shared data.

from the disk maintaining that page. The third column of the table lists the amount of disk storage provided in each node in the simulation. The last column of the table shows the total amount of shared data that was accessed by the application during the simulation.

Table 5.4 characterizes the applications by the number of instructions executed, the total number of read accesses to shared data and the total number of write accesses to shared data. The table lists these measures for simulations with 64-byte and 256-byte blocks, both for the one-level scheme and for the hierarchical scheme. LU and Radix are roughly equal in size in the number of instructions executed (about 400 million). Cholesky is a larger application executing four times as many instructions. As the table shows, the number of instructions executed can be different in simulation runs with different block sizes. Since the latency of shared accesses is influenced by the block size, processes can arrive at synchronization barriers and locks in different sequence in simulations with

Application	Scheme	Block Size (bytes)	Number of Instructions (x 10^6)	Number of Shared Reads (x 10^6)	Number of Shared Writes (x 10^6)
Cholesky	Onelevel	64	1433.47	69.23	7.74
		256	1401.09	67.83	7.74
	Hierarchical	64	1390.37	67.43	7.74
		256	1362.38	65.75	7.74
LU	Onelevel	64	460.12	12.09	5.72
		256	469.58	12.09	5.72
	Hierarchical	64	458.01	12.09	5.72
		256	466.58	12.09	5.72
Radix	Onelevel	64	417.29	11.86	6.51
		256	421.06	11.96	6.51
	Hierarchical	64	416.75	11.86	6.51
		256	420.64	11.95	6.51

Table 5.4: Number of instructions and shared accesses in the applications.

different block sizes. As a result, the number of instructions executed in checking whether all processes have reached the barrier or in suspending and waking up processes may be different in simulations with different block sizes. With Cholesky, such differences in the number of instructions executed can also occur due to the additional synchronization structures (e.g., to signal that a task is ready for processing) that are implemented in the application. The number of instructions executed spinning on these structures (waiting for a task to become ready, for example) can be different in simulations with different block sizes. In all three applications, shared accesses constitute a small fraction of the application's instructions. The percentage of instructions which are shared

accesses is only 5% in Cholesky, 3.7% in LU and 4.2% in Radix.

Application	Scheme	Block Size (bytes)	Number of Instructions in Parallel Phase (x 10^6)	Number of Shared Reads in Parallel Phase (x 10^6)	Number of Shared Writes in Parallel Phase (x 10^6)
Cholesky	Onelevel	64	1227.63	65.23	2.72
		256	1203.06	63.83	2.72
	Hierarchical	64	1193.49	63.43	2.72
		256	1165.50	61.75	2.72
LU	Onelevel	64	451.91	11.96	5.59
		256	461.66	11.96	5.59
	Hierarchical	64	450.18	11.96	5.59
		256	458.75	11.96	5.59
Radix	Onelevel	64	417.03	11.33	6.51
		256	420.86	11.43	6.51
	Hierarchical	64	416.53	11.34	6.51
		256	420.44	11.42	6.51

Table 5.5: Number of instructions and shared accesses in the parallel phase of the applications. Applications have an initial serial phase when a single process executes initialization code and spawns other processes. Most of the instructions and shared accesses are in the parallel phase of the application.

The execution of all three applications have an initial serial phase before the parallel phase of execution. In the serial phase, a single process executes initialization code and then spawns all other processes for the parallel phase. Table 5.5 lists the number of instructions executed in the parallel phase of the execution and the number of read and write accesses to shared address space in the parallel phase. The instruction count and access count measures for the parallel phase are the sum of the instruction counts and the access counts, respectively, of

all the processes that executed during the parallel phase. The table presents measurements from simulation runs with 64-byte blocks and 256-byte blocks for the one-level and the hierarchical schemes. Comparing with the numbers tabulated for the overall execution in Table 5.4, it can be seen that all three applications execute most of their instructions in their parallel phase. Cholesky executes 86% of its instructions in the parallel phase, while LU and Radix execute nearly all their instructions in the parallel phase. While roughly 94% of Cholesky's read accesses to shared data are in its parallel phase, only 35% of its write accesses to shared data are in the parallel phase. The majority of Cholesky's shared write accesses occur in the initialization. 99% of LU's shared reads and 98% of its shared writes occur in its parallel phase. In Radix, the array to be sorted is initialized only in the parallel phase. Hence, all its shared writes are in the parallel phase.

To study the distribution of the workload amongst the processors, we have measured the number of instructions, the number of shared reads and the number of shared writes executed by each processor in the parallel phase. The histograms in Figure 5.4, Figure 5.5 and Figure 5.6 show the breakdown of instructions, shared reads and shared writes, respectively, amongst the processors (for the parallel phase). In each figure, the breakdown is shown for all three applications on the one-level managed and hierarchically managed DSM systems with 64 byte blocks.

Figure 5.4 shows that the total number of instructions executed is very well distributed among the individual processors in the Cholesky application (mean of 19.17 $\times 10^6$ cycles and a standard deviation of 0.37 $\times 10^6$ cycles). The breakdown of instructions is less balanced in the LU kernel (mean of 6.79 $\times 10^6$ cycles and a standard deviation of 0.91 $\times 10^6$ cycles). The breakdown of instructions is least

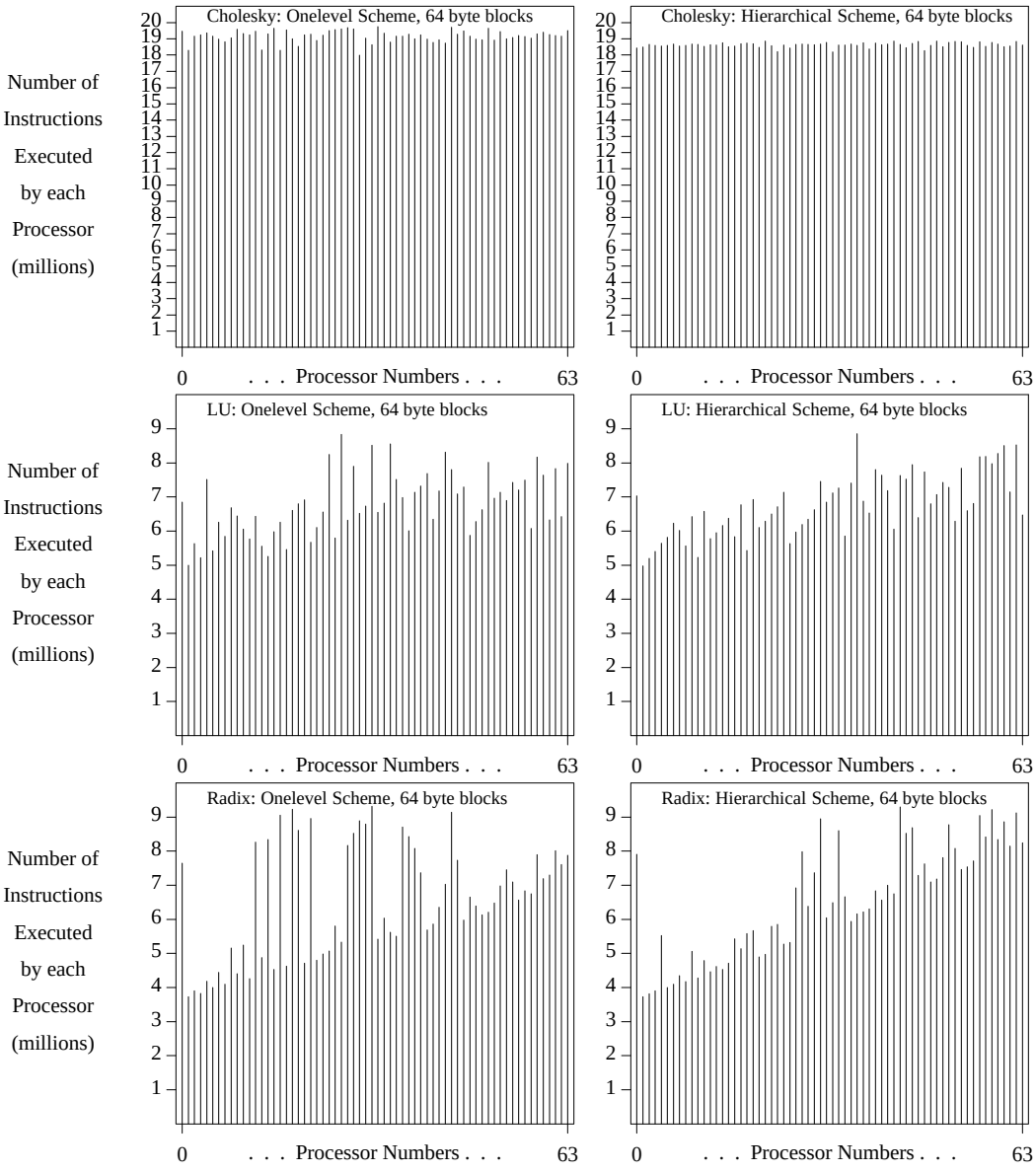


Figure 5.4: Distribution of application instructions amongst processors.

uniform in the Radix kernel (mean of 6.54×10^6 cycles and a standard deviation of 1.65×10^6 cycles). The figure also shows a clear pattern in the distribution of instructions among the processors in Radix. In the Radix application kernel, all

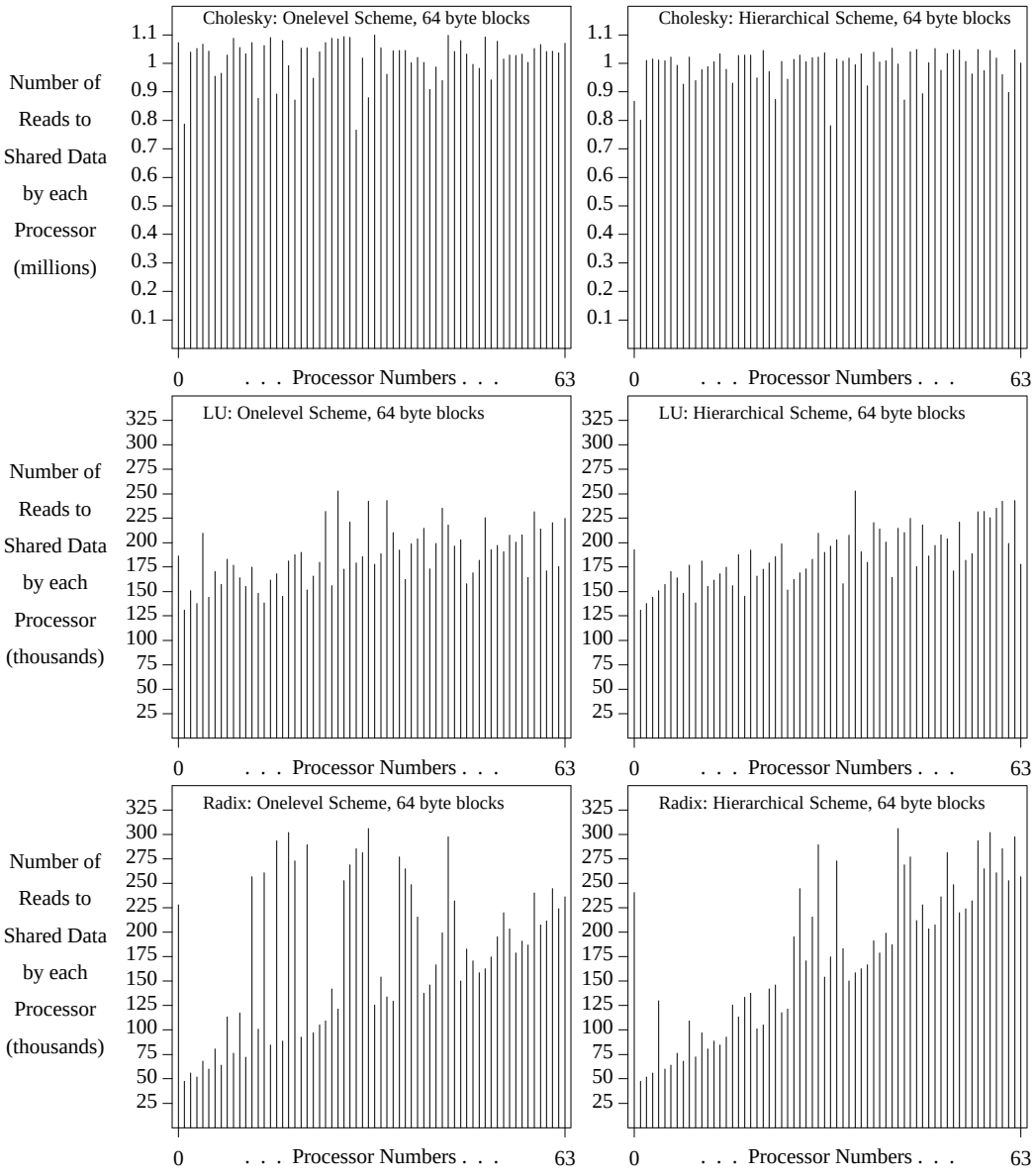


Figure 5.5: Distribution of application shared reads amongst processors.

processes are ranked in a fixed order. In every iteration of the sort, each process determines the location of its keys by reading the local histograms generated by each process with a lower rank. This phase results in a non-uniform distribution of

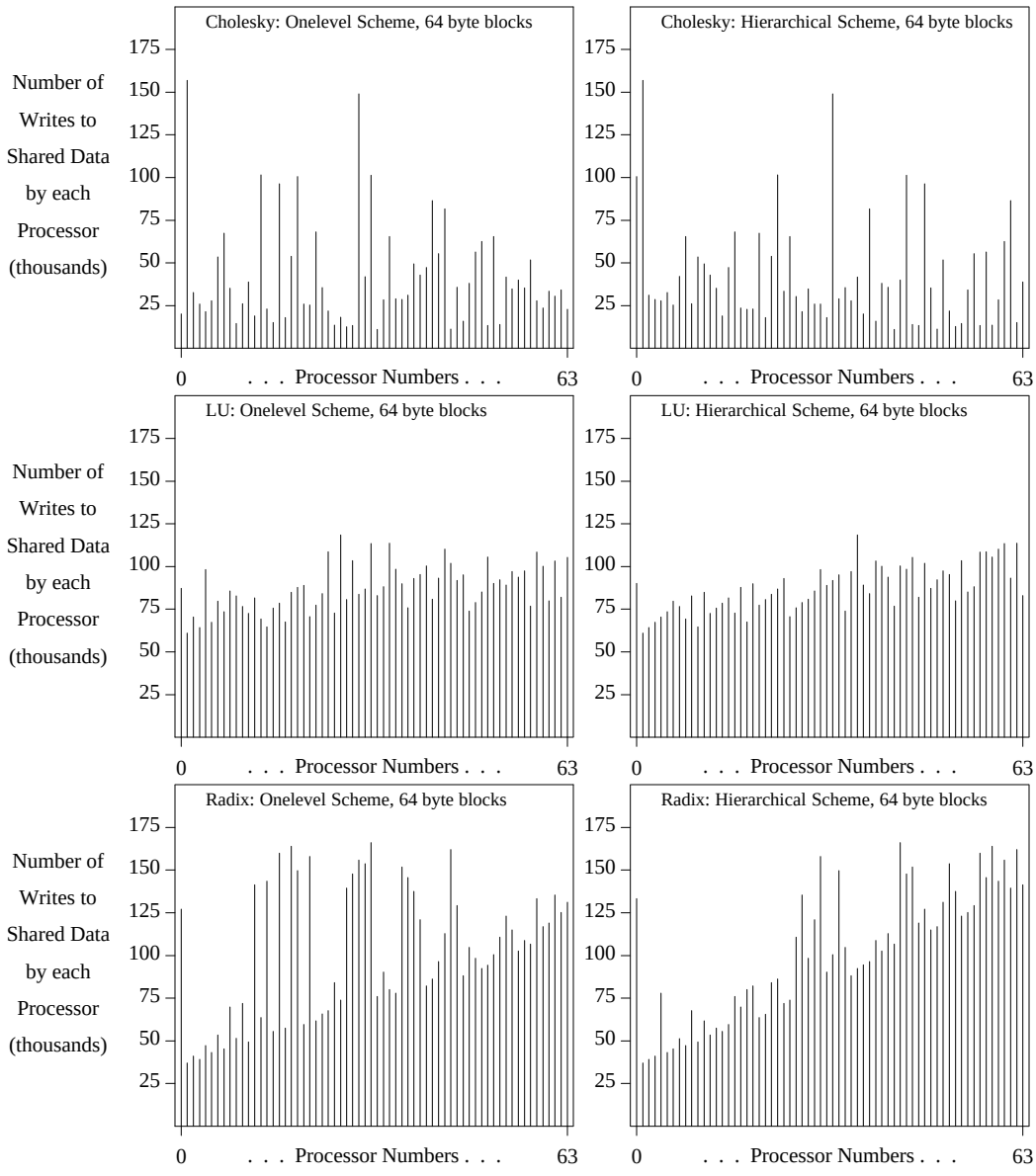


Figure 5.6: Distribution of application shared writes amongst processors.

workload, where the workload of a process is proportional to its rank in the order. The figure shows the workload changing nearly linearly among the processors (it is not exactly linear because the rank number of a process and the number of the

processor where it executes are different).

Figure 5.5 and Figure 5.6 show the breakdown of read and write accesses to shared address space amongst the processors. The distribution of shared read and write accesses among the processors in LU and Radix is very similar to the distribution of instructions among the processors in these applications. In the Cholesky application, accesses to shared data, especially write accesses, are less evenly distributed among the processors than instructions.

Most of our evaluation has studied the behavior of the system and application in the parallel phase of execution. In the discussion of our simulation results in the following sections, we present our results for the parallel phase of application execution.

5.1.3. Measurement of the Shared Working Set Size

In many applications with a large shared data set, only a small fraction of the references may be to shared data [Dare87]. In addition, even when shared data is accessed by a process, the shared pages in the working set of the process may be dynamically private to the process [Pink92]. As a result, only a small fraction of the shared data set may actually be shared by any process at any time. Based on this observation, the hierarchical scheme allows only a small fraction of data in the physical memory of any node to be shared at any time. The Shared Pages Table (SPT) which manages the dynamically shared component of the application's working set has fewer entries than the Present Pages Table (PPT). Hence, only a small fraction of the data mapped to a node's physical memory can be dynamically shared with other nodes. If the number of SPT entries is insufficient to maintain

the shared working set of the application, frequent SPT entry thrashing can lead to performance degradation.

To evaluate the validity of our premise about parallel application behavior, we have studied application performance at different SPT sizes. We studied the applications on a system simulating the one-level scheme with small 16-byte pages. The small size of the page minimizes false sharing, allowing us to study the effect of true-sharing in the application. We simulated the one-level system for this study since the number of SPT entries is a better measure of the memory available for the application's shared working set with the one-level scheme than with the hierarchical scheme. While SPT entries in the hierarchical scheme allow data to be shared in units smaller than a page (block-level sharing), each SPT entry corresponds to a page of data. Hence, in the hierarchical scheme, when an SPT entry is used by a block of a given page, any unused space in the page frame is not available for use by blocks of other pages.

We ran several simulations of the execution of each application on a one-level system with 16-byte pages. The number of entries in the Shared Pages Table (SPT) was varied in these simulation runs. In all runs, 128 Kbytes of physical memory and 8 K Present Pages Table entries were available on each node. Figure 5.7 presents the results from these simulations.

Figure 5.7 plots the execution time of each application as a function of the number of entries in the SPT. Application performance is lower (longer execution time) at smaller SPT sizes due to higher contention for SPT entries. However, for all three applications, small SPTs with only 256 entries can support application performance within 20% (within 3% in the case of LU and Radix) of the

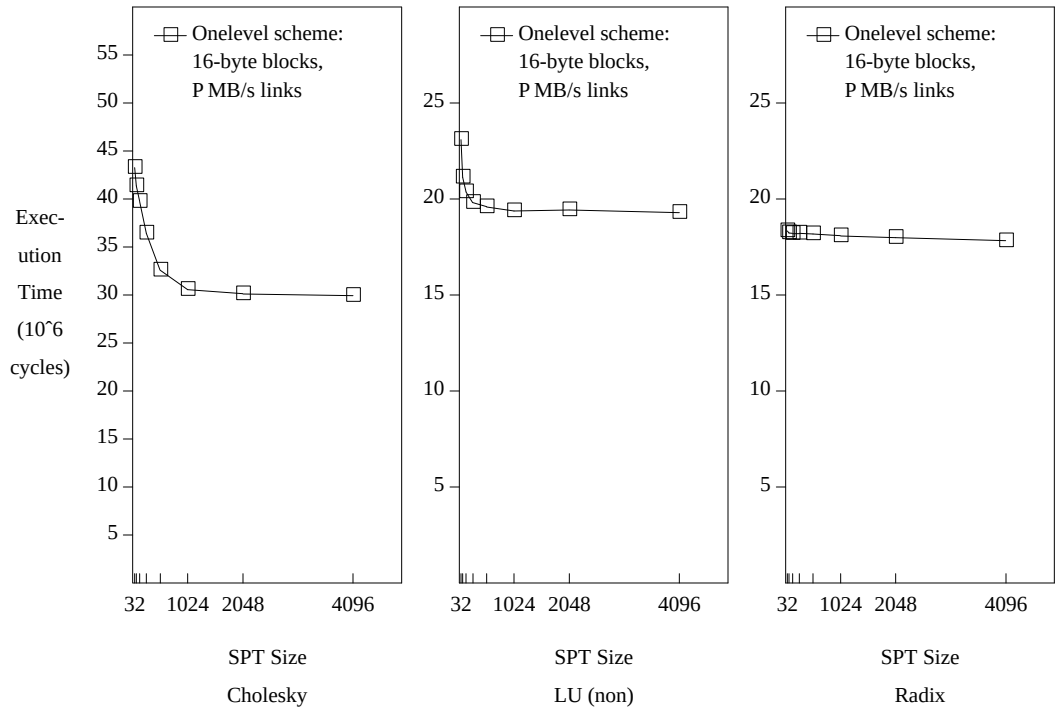


Figure 5.7: Small Shared Working Set. The application's execution time is plotted as a function of the number of entries in the Shared Pages Table (SPT), for each of the applications executed in our simulation. The scheme simulated is the one-level scheme with 16 byte blocks. The number of entries in the Present Pages Table was 8 K. The applications perform well even when the amount of shared data mapped by the SPT is a small fraction of of the total amount of shared data used by the application (listed in Table 5.3).

performance with 4 K SPT entries. This shows that most of the data that is actually shared at any time can be managed with 256 SPT entries on each node. The amount of shared data mapped by 256 SPT entries is 4 Kbytes (16-byte blocks), which is a small fraction of the total shared data used by the applications (see Table5.3). Thus, Figure 5.7 verifies that, due to application sharing behavior, for many applications there is a relatively small performance penalty for using an SPT which is much smaller than the PPT.

5.1.4. Measurement of the Effectiveness of the Copy Directory Organization

To support block-level sharing, the hierarchical scheme maintains copy directory information for each block of every shared page. In Section 3.2.2 of Chapter 3, we discussed two possible ways to organize the copy directory. The straightforward organization, called the Processor List per Block or PLpB organization, maintains an independent list of processors for each block of a shared page. We proposed an alternative organization, called the Block Map per Processor or BMpP organization (Section 3.2.2) for the hierarchical scheme. In the BMpP organization, the copy directory information associated with all blocks of a page at a node are unified in a compact copy directory common to all blocks of the page on that node. The BMpP copy directory maintains the list of processors that share any block of the page and maintains a bit map for each processor identifying the blocks held by that processor. In this sub-section, we discuss our results comparing the effectiveness of these two copy directory organizations. We also discuss the feasibility of bounding the number of entries in the BMpP copy directory.

In the hierarchical scheme, a node that owns any block of the page (block owner) maintains in its copy directory for the page the identities of nodes having read-only copies of any block of the page that it owns locally. The page owner maintains the same information if it owns any blocks of the page, and also maintains the identities of all nodes that own any blocks of the page in its copy directory. The simulation measures the number of nodes in the copy directory when the copy directory is maintained independently for each block (PLpB organization) and when a common copy directory is maintained for all blocks of

the page (BMpP organization). These numbers are measured independently for the copy directory in block owners and for the copy directory in the page owner.

During execution, a copy directory is initialized for a page on a node when a block of the page is first shared from that node by another node. When other nodes stop sharing these blocks, the copy directory is deallocated. Within the lifetime (time between initialization and deallocation) of each copy directory, the number of nodes in the copy directory varies with time. During the lifetime of each copy directory, the simulator tracks the changes in the number of nodes in the copy directory and the time interval between these changes. For the lifetime of each copy directory that is used, the simulator computes the time-weighted average number of nodes in the copy directory if the copy directory is organized as a PLpB copy directory and if it is organized as a BMpP copy directory. The simulator also measures the lifetime of each copy directory. At the end of execution, the average number of nodes in a copy directory is computed for each copy directory organization. This average is computed by averaging the numbers measured for each copy directory that was used during the simulation over all copy directories that were used weighted by the lifetime of each copy directory. The average is computed independently for copy directories in block owners and for copy directories in page owners.

The effectiveness of the PLpB and BMpP copy directory organizations can be compared by comparing the storage required with each organization to maintain the average number of copy directory entries measured during simulation. The size of the copy directory needed to accommodate the entries can be easily calculated for both copy directory organizations. The size of an entry in the PLpB

organization is simply the size of a node identifier (6 bits for a 64-node system). The size of the copy directory of a block in the PLpB organization is the product of the size of the entry and the average number of entries measured by the simulation. Since a copy directory must be maintained for each block of a page in the PLpB organization, the total size of PLpB copy directories for a page is the product of the size of the block copy directory and the number of blocks per page. The size of an entry in the BMpP organization is the size of the node identifier plus the number of blocks in a page (to store the bit map). The total size of the BMpP copy directory for any page is the product of the size of an entry and the measured number of entries per copy directory.

We performed this simulation and analysis of copy directory organization effectiveness with different block sizes. We simulated a 64-node hierarchical DSM system using 1 KB pages. The results of this analysis are shown in Figure 5.8. The figure shows the computed average size of the copy directory for a page for both copy directory organizations (independently for the block owner and the page owner) as a function of the block size. The BMpP organization of the copy directory requires less storage than with the PLpB organization, especially with small blocks. With both organizations, the storage required for the copy directory is higher at smaller block sizes. At small block sizes, false-sharing is lower, which can lead to fewer entries in the copy directory. However, smaller blocks result in a larger number of blocks per page, which can lead to higher storage overhead for the copy directories. Specifically, more blocks require more copy directory entries with the the PLpB organization (one per block) and require larger block maps with the BMpP organization (one bit per block).

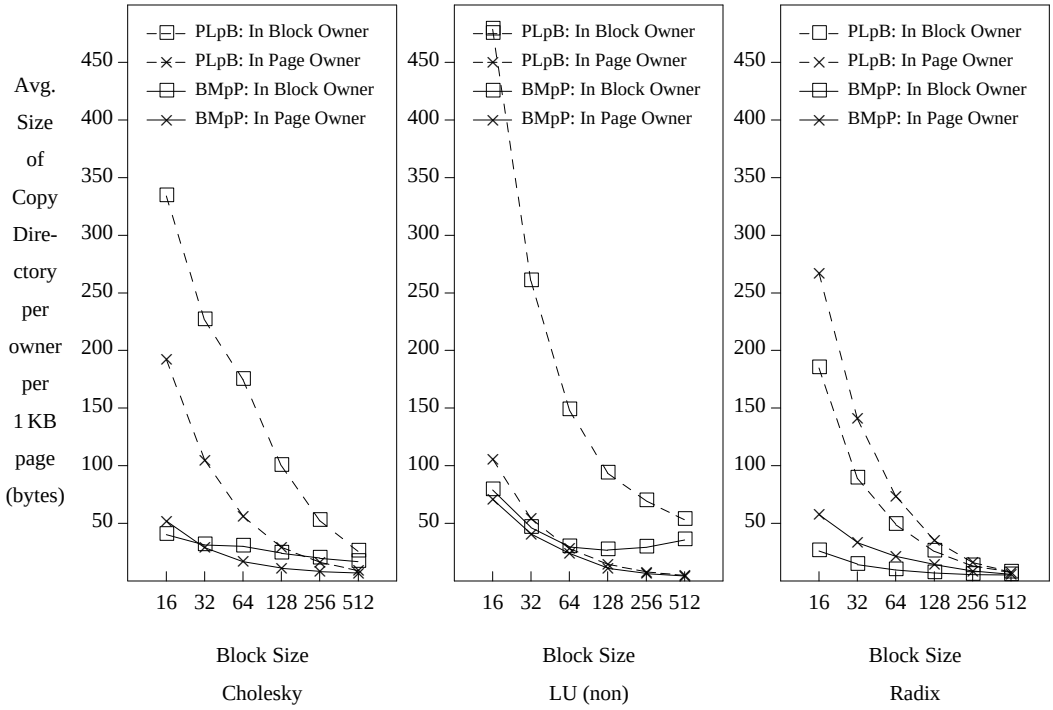


Figure 5.8: BMpP organization of the copy directory is more efficient. The simulation measured the average number of nodes in the copy directory, when it is maintained as an independent processor list per block (PLpB), and when it is maintained for the entire page as a block map for each processor(BMpP). The size of the copy directory needed to accommodate these number of nodes can be computed, and is plotted here for both organizations. The BMpP copy directory can be smaller in maintaining the same information.

Figure 5.9 shows the average number of nodes in the BMpP copy directory as a function of block size. The average is shown independently for block owners and page owners. At larger block sizes, the number of nodes having read copies of a block is higher due to false sharing, requiring more entries in the copy directory at the block owner (marginal in the Radix application). However, the number of entries in the page owner's copy directory decreases with increasing block size. As discussed earlier, this is because for a fixed page size (1 KB in the simulations) the

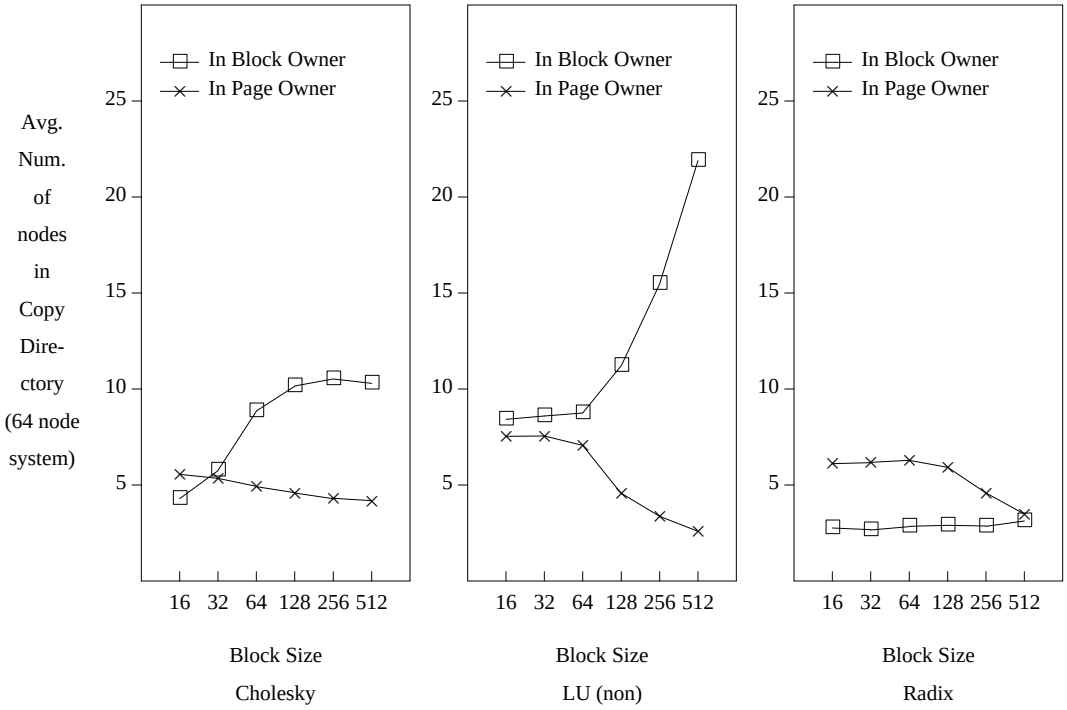


Figure 5.9: Copy Directory can be bounded. The graph plots the number of nodes in the block owner and page owner copy directories in a 64-node system implementing the hierarchical scheme with 1 Kbyte pages. The number of nodes in the copy directory is small at small block sizes, making it reasonable to bound the number of entries in the copy directory.

number of blocks per page (and, hence, the number of potential block owners) decreases with increasing block size. At small block sizes, the average required size of the copy directory is small (≈ 5 nodes in a 64-node system) in both the page owner and the block owners. Since the small average indicates that only a few nodes occupied the directory during a large proportion of the lifetime of the copy directory, it may be reasonable to bound the size of the BMpP copy directory.

5.1.5. Performance Comparison Between the Hierarchical and One-Level Schemes

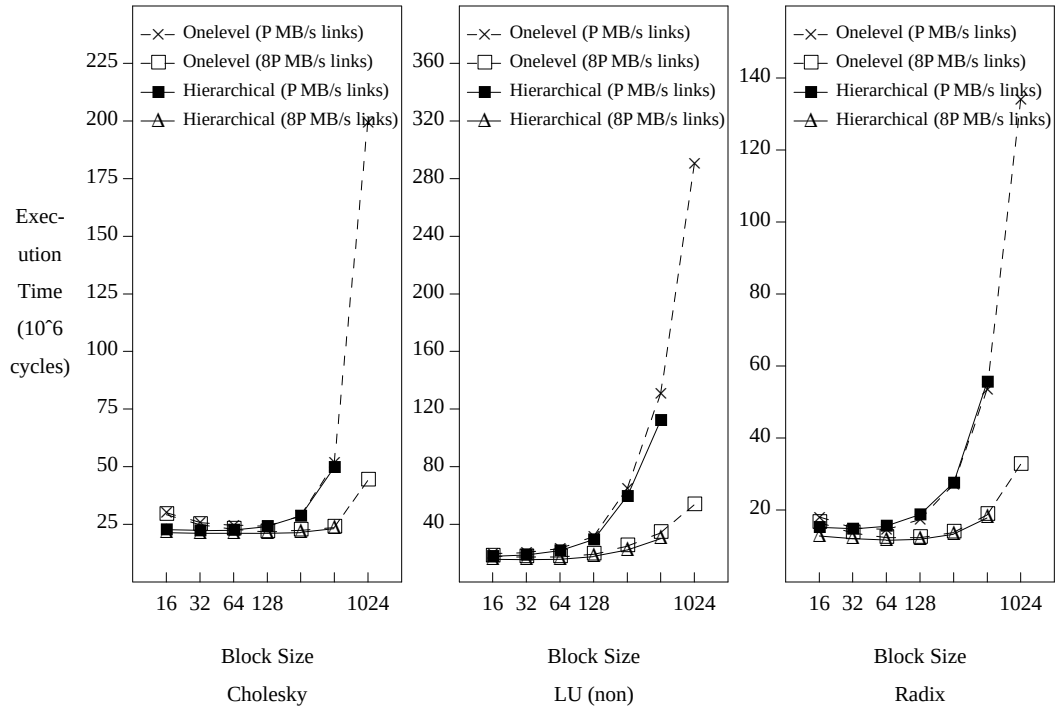


Figure 5.10: Performance of the hierarchical scheme. The graph compares the application execution time with the hierarchical scheme using 1 KB pages, and with the one-level scheme, at various block sizes. Performance is poor with the page-sized one-level scheme (one-level scheme with block size 1024 bytes). The one-level scheme performs well at small block sizes, and the hierarchical scheme matches its performance. When the network is faster, all schemes improve and the qualitative comparison remains the same.

We compare the overall performance of the hierarchical scheme and the one-level scheme in Figure 5.10. For each scheme, the figure shows the execution time of each application as a function of the block size. As expected, when blocks are the size of a memory page (1 KB), the one-level scheme has very poor performance. This is primarily due to the high false sharing and high network

latency at large block sizes. Also, the one-level scheme performs very well at small block sizes. The graph also shows that the hierarchical scheme (page size is 1 KB) achieves nearly identical performance to the one-level scheme even for small blocks. As we have seen earlier, the hierarchical scheme incurs a small fraction of the storage overhead incurred by the one-level scheme in supporting this high performance.

We evaluated the hierarchical and one-level schemes at various network link bandwidths. Figure 5.10 shows the comparison for two different networks, one of which has a higher link bandwidth (8P MB/s versus P MB/s link bandwidth) than the other. When the network link bandwidth is higher, the average latency through the network is lower. This leads to overall improvement in application performance with both schemes. However, the qualitative comparison of the schemes remains the same.

Both the hierarchical scheme and the page-sized one-level scheme map data at large page sizes. However, it can be seen from Figure 5.10 that the hierarchical scheme supports higher application performance than the page-sized one-level scheme. This can be attributed to the support for fine-grain block-level sharing in the hierarchical scheme. The smaller block size leads to lower false-sharing, lower network latency and lower network contention. We address the influence of small block sizes in the rest of this sub-section.

When the size of a block is the smallest addressable unit, the fraction of a page shared from a node reflects the true sharing in that page. When the block size is larger than this minimal word size, a larger portion of the page may be shared from the node than required by true sharing due to the larger sharing granularity.

The degree of false sharing is indicated by the increase in the fraction of the page that is shared. To evaluate this false-sharing, we have studied the impact of block size on the average fraction of a fixed sized page that is shared. Since the Block Map per Processor (BMpP) copy directory organization maintains information about the sharing of each page, we make use of its information in this study.

A specific block of a page is actively shared only if the BMpP copy directory of the page identifies some node for that block. The BMpP copy directory at the page owner identifies nodes that have a read or an ownership copy of any block of the page. Hence, the fraction of blocks that have any nodes indicated in the page owner's copy directory identifies the fraction of the page that is shared by any node. The copy directory at the block owner identifies nodes that hold a read copy of blocks owned by the block owner. Hence, the fraction of blocks that have any nodes indicated in the block owner's copy directory identifies the fraction of the page that is read-shared from that block owner.

When a block of a page is first shared from a node during execution, an SPT entry must be allocated for the page on that node. When the page is no longer shared from that node, the SPT entry is deallocated. During the lifetime of the shared page on that node (or lifetime of the SPT entry), different number of blocks of the page may be shared by other nodes. Our simulator measures, for each page shared in the execution, the average fraction of blocks of the page that is indicated as shared in its BMpP copy directory as well as the lifetime of the page. These measures are collected independently for the copy directory at the block owners and at the page owner. The overall average of the fraction of blocks of a shared page that are shared is computed (independently for page owners and for block

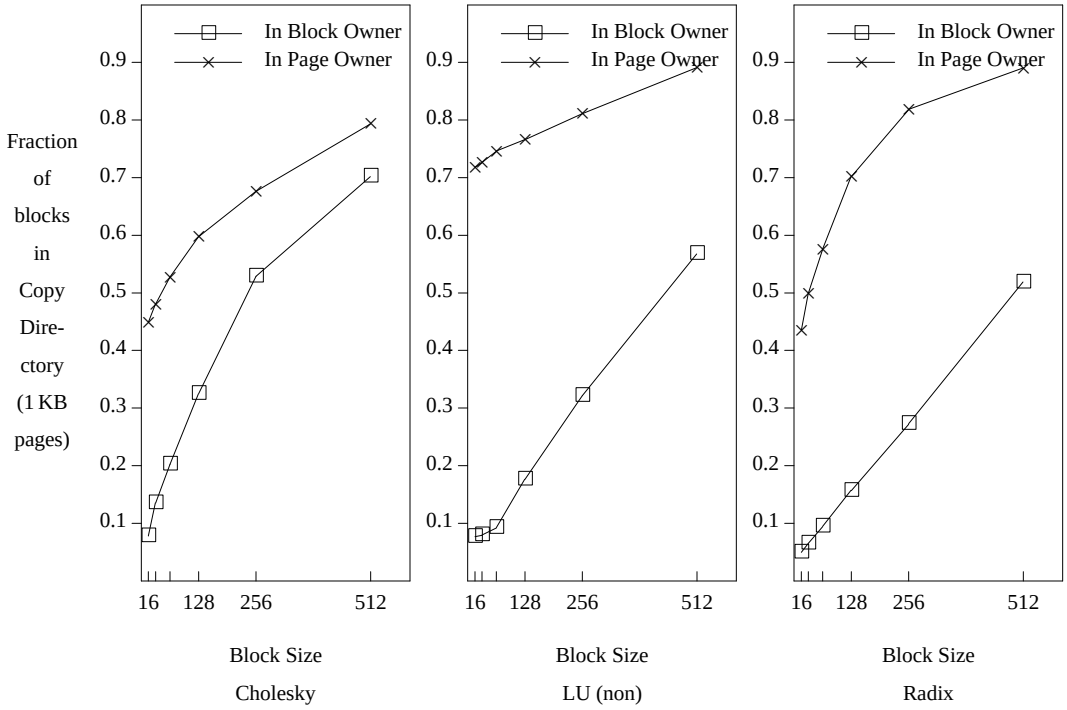


Figure 5.11: False sharing is higher at larger block sizes. A larger fraction of the blocks of the page are shared when the block size is larger (for a fixed page size). The page size used for these simulations is 1 KB.

owners) by averaging all the individual page measures, weighted by the lifetime of each page. For the hierarchical scheme with 1 KB pages, Figure 5.11 shows this computed fraction of blocks of a page (for page owner and the block owner) as a function of the block size.

The fraction of blocks indicated as shared in the block owner copy directory is smaller than the fraction of blocks indicated as shared in the page owner copy directory in Figure 5.11. This is expected since the page owner maintains entries for all blocks that are not locally owned even if they are locally invalid, while the block owner only maintains entries for locally owned blocks that are read shared.

The graphs also show that at small block sizes, under 10% of the blocks are actively read-shared from any block owner. At large block sizes, each block owner manages (owns) a larger fraction of the page and a node that read-shares a block shares a larger fraction of the page. Hence, each block owner shares a higher fraction of blocks at larger block sizes.

Figure 5.11 also shows that the fraction of blocks that are shared in the page owner copy directory is smaller at small block sizes, especially in Cholesky and Radix. This indicates that, in Cholesky and Radix, at larger block sizes, additional data is unnecessarily shared from the page owner by other nodes due to false sharing. In the case of LU, the fraction of blocks that are shared from the page owner is quite high even at small block sizes. The LU application does not allocate the application data accessed by a single process contiguously (see Section 5.1.2). As a result, data belonging to multiple processors can be mapped to the same page. Since ownership of the page is assigned to the node that accesses the page from disk (see Section 4.3.1 in Chapter 4), other nodes which subsequently access the page share the page from the page owner. Hence, the high fraction of blocks shared from the page owner in LU may not be indicative of the true sharing in LU. It is indicative of the high degree of page-level false sharing.

To demonstrate the impact of block size on network latency, we study the average latency of reads that require network communication. Read accesses may need to be retried when they compete against an ongoing write. To ignore the latency contribution due to such access contention, Figure 5.12 shows the average latency of read accesses requiring network communication which do not involve a retry. The read latency is plotted for the one-level and the hierarchical schemes for

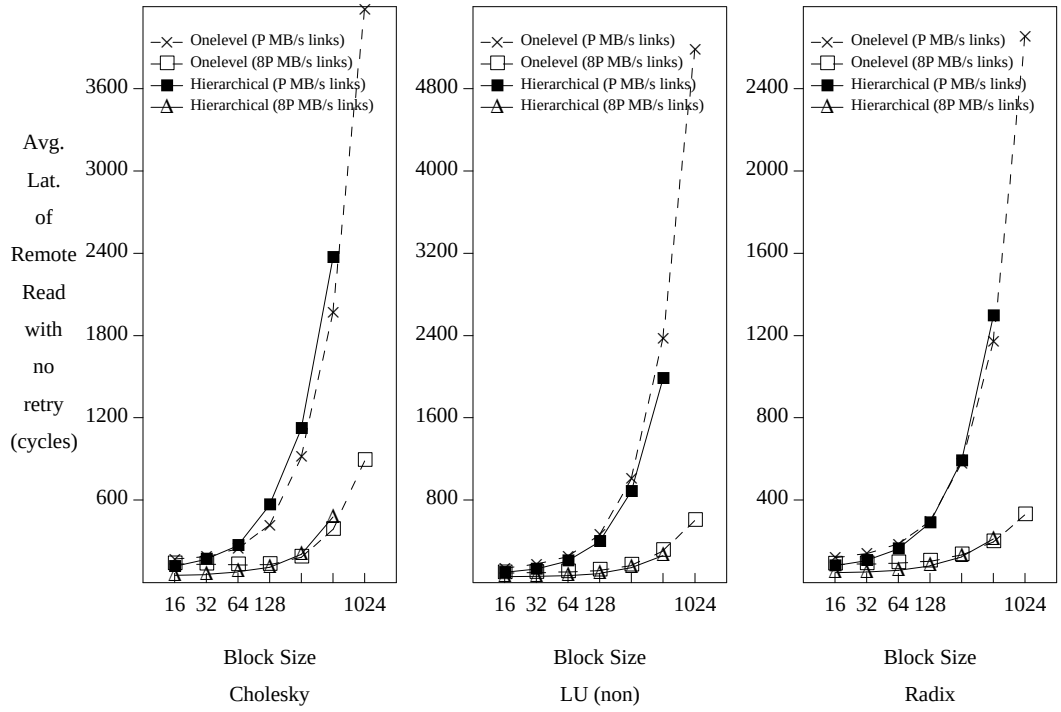


Figure 5.12: Latency of remote read accesses is higher at larger block sizes.

two different network speeds. The response time for read accesses is much higher at large block sizes in all cases. The response time improves significantly in the faster network.

When a processor access cannot be completed locally, the processor is blocked until the access can be satisfied. As we have seen above, non-local accesses take longer to complete at larger block sizes. Hence, at larger block sizes, the processor remains blocked on remote accesses longer. This lowers the processor utilization — the fraction of time a processor is not blocked for remote accesses or synchronization. Figure 5.13 shows the processor utilization as a function of block size for all three applications. In all three applications, the

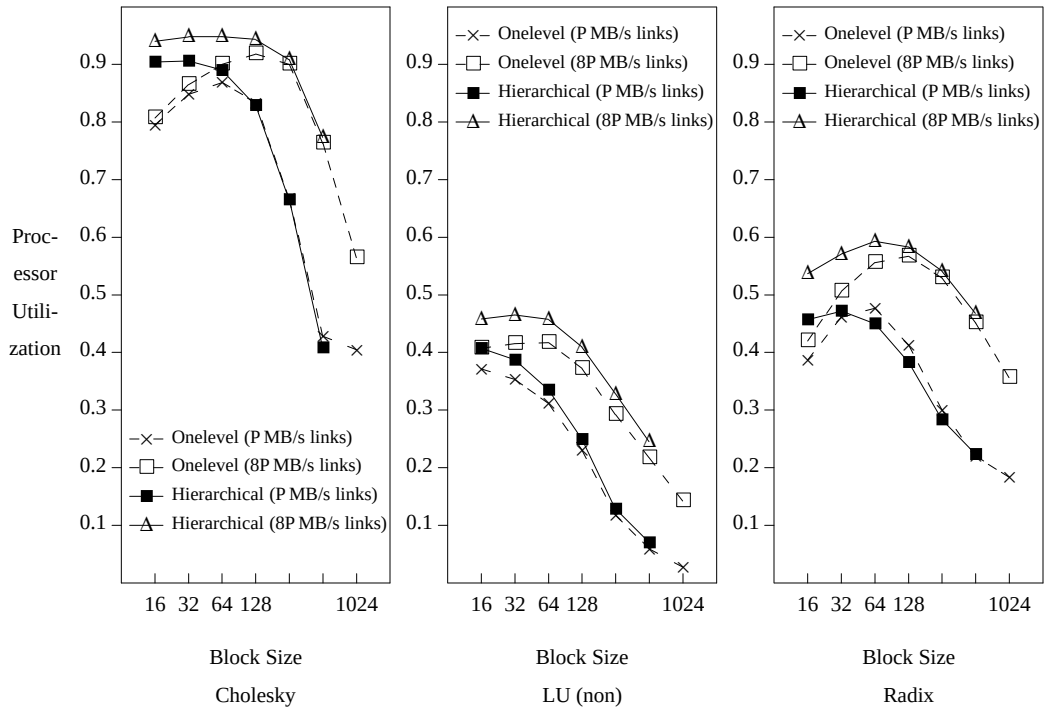


Figure 5.13: Processor utilization is lower at larger block sizes.

processor utilization is very low at large block sizes as expected. At very small block sizes (16 or 32 bytes), more remote accesses may be needed to acquire all data needed for processing than if the block size were 64 or 128 bytes. As a result, the processor utilization also drops at very small block sizes.

The drop in processor utilization at very small block sizes is magnified in the Cholesky and Radix applications with the one-level scheme due to an artifact in the simulation system model. As we discussed in Section 5.1, disk accesses are at the granularity of pages with the hierarchical scheme while disk accesses are at the granularity of blocks with the one-level scheme. At small block sizes, more disk accesses are needed in the one-level scheme to bring in all the data needed in block-sized chunks. The larger number of disk accesses lowers processor

utilization in the one-level scheme at small block sizes. This effect is significant in Cholesky and Radix due to a significant number of disk accesses in these applications. In Cholesky, disk accesses are needed since the large data set does not fit in a node's physical memory. In Radix, the disk accesses are needed to initialize the array to be sorted. It should be noted that no real system accesses small blocks (less than 1 K bytes) from disk. Hence, no conclusions regarding real systems may be drawn from the results in Figure 5.13 with respect to performance of the one-level system versus the hierarchical scheme with small blocks.

5.1.6. Characterization and Measurement of Non-Local Read and Write Accesses

When a read or write request cannot be satisfied locally within the node, remote access is necessary to satisfy the request. Messages must be sent to one or more nodes to acquire the requested data with sufficient access permission. The type of the message(s) exchanged and the nodes that must be contacted are specified by the coherence protocol. They depend on the type of request (read or write). For example, consider the reply message received by the requesting node in the coherence protocol for the hierarchical scheme (see Section 4.3.4 in Chapter 4). A node issuing a read request will receive the read copy reply from the owner of the block (block owner) containing the requested data. But, a node issuing a write request will receive the ownership copy reply from the owner of the page (page owner) containing the requested data. The message types and the set of nodes contacted also depend on the dynamic shared state of the required data. If the requesting node has a copy of some block of the page containing the requested

data, the request is sent to the owner of the page; otherwise, the request is sent to the default owner of the page. Therefore, each remote request will involve a different sequence of messages and will require a different set of tables to be looked up in different nodes.

Since the coherence protocol used for the hierarchically managed and one-level managed systems are different (see Section 4.3.4 in Chapter 4 and Section 5.1.1), remote requests also have different behavior on the two systems. The hierarchical scheme supports two levels of ownership and, as a result, more levels of forwarding than the one-level scheme. Remote requests can, therefore, involve more messages and more table lookups in the hierarchical scheme than in the one-level scheme.

We have used our simulations to profile the behavior of remote requests based on the different sequences of messages and nodes that are involved in satisfying the request and on the network latency and table lookup latency incurred in satisfying the request. Our study has profiled the behavior of remote requests on both the hierarchical and one-level schemes. In our discussion of the coherence protocol in Chapter 4, we identified different flows of the protocol that are possible in executing remote read and write requests (Figure 4.18 and Figure 4.21 in Chapter 4, for example). To assist in characterizing the behavior of remote requests, we have classified the various coherence protocol flows that are used to satisfy remote requests. In the simulation, we have measured the number of occurrences of each class of protocol flow in satisfying remote requests.

We present measurements of the fraction of shared accesses that require non-local access in Section 5.1.6.1. We introduce our protocol flow classification in

Section 5.1.6.2. Based on simulation results, in Section 5.1.6.3, we present the frequencies of occurrence of different classes of protocol flows for a variety of non-local access scenarios with both the one-level and hierarchical schemes. The latency of a remote access depends on the latency of the messages through the network and the latency of accessing tables at the servicing nodes. In Section 5.1.6.4, we present simulation results that quantify the contributions of the network latency and the table lookup latency to the overall latency of remote requests.

5.1.6.1. Measurement of the Miss Rate of Shared Accesses

A shared read or write request by a processor requires remote access only when the processor's request misses in the local memory of that node. We have measured the miss rate for shared accesses and show our results in Figure 5.14. Figure 5.14 shows the fraction of shared reads that require non-local access and the fraction of shared writes that require non-local access for both the one-level and hierarchical schemes. The results are shown for simulations with different block sizes. The overall miss rate for shared accesses can be calculated from the miss rates for shared reads and shared writes shown in Figure 5.14 and the number of shared reads and writes listed listed earlier in Table 5.5 (Section 5.1.2).

The figure shows that the miss rate for shared accesses is under 10% for Cholesky and Radix for nearly all block sizes. In the case of LU, the miss rate is higher since the data accessed by a process are not allocated contiguously. In all three applications, read accesses have better hit rate than write accesses. This is expected since write accesses hit only when the block is owned exclusively by the

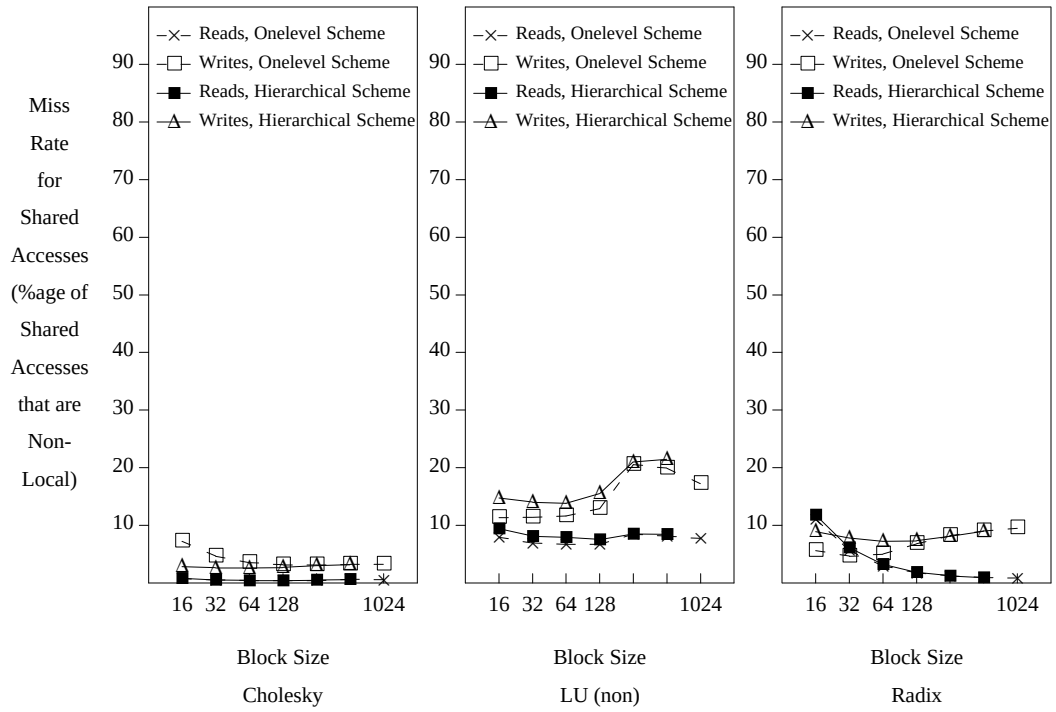


Figure 5.14: Miss Rate for Shared Accesses.

local node. Read accesses do not require exclusive access.

In general, the miss rate for shared reads decreases with increasing block size in both the one-level and the hierarchical schemes. With larger blocks, a larger chunk of data is transfered to satisfy a read miss. Thus, more subsequent accesses are likely to hit on the same block without requiring remote access. However, when the block size is increased beyond 128 bytes, miss rate can increase due to false sharing. This is especially the case in LU, where data modified by different nodes get mapped to the same block when the block size is larger than 128 bytes. When data modified by multiple nodes is mapped to the same block, read copies held by one node can be unnecessarily invalidated when another node writes to a different part of the block. As a result, more shared reads may require non-local

access.

Shared writes also show a small decrease in miss rate with increasing block size at small block sizes. However, shared writes demonstrate higher increase in miss rate at large block sizes than shared reads. This is expected since write accesses require an exclusive copy of the block (unlike reads which require only a shared copy). With larger blocks, the block is more likely to be shared by other nodes. As a result, more write accesses may require remote access to invalidate read copies. With LU, the miss rate for shared writes is especially higher with blocks larger than 128 bytes since data modified by multiple nodes get mapped to the same block.

With LU and Radix, at small block sizes, the miss rates are slightly higher in the hierarchical scheme than in the one-level scheme using the same block size. This is most likely due to the fact that the SPT entry contention can be higher in the hierarchical scheme than the one-level scheme at small block sizes. Due to the page-level mapping of the SPT entry in the hierarchical scheme, there are fewer SPT entries for a given amount of storage in the hierarchical scheme. For example, in our simulations, the SPT has 2048 entries when simulating the one-level scheme with 16-byte blocks while the SPT has 128 entries when simulating the hierarchical scheme with 16-byte blocks (see Table 5.2 in Section 5.1). The contention for SPT entries is hence marginally higher in the hierarchical scheme at small block sizes. When there is contention for SPT entries, it is necessary to free SPT entries by evicting shared pages. Since the eviction discards shared data that is currently mapped to the node's local memory, subsequent accesses to the evicted page require remote access leading to a small increase in miss rate.

As will be demonstrated later in this subsection, the miss rate results are also influenced by the different implementation of disk accesses in the one-level and the hierarchical schemes. As we described earlier in Section 5.1, data is accessed from the disk in block-sized chunks in the one-level scheme while data is accessed in page-sized chunks in the hierarchical scheme. Consequently, a shared access miss that is satisfied from disk brings in a page of data in the hierarchical scheme, while it brings in a smaller block of data in the one-level scheme. The larger chunk of data that is transfered to satisfy the miss in the hierarchical scheme can potentially satisfy more subsequent accesses to the page locally, leading to lower miss rate than in the one-level scheme. However, the larger transfer size in the hierarchical scheme may not benefit subsequent accesses when data modified by multiple nodes are mapped to the same page. In such cases, intervening writes by other nodes writing to the page can invalidate sections of the page from local memory forcing subsequent accesses to the page to issue a remote request.

Application	Number of Shared Read Accesses Satisfied by Disk Access				
	Onelevel Scheme: 16 byte accesses	Onelevel Scheme: 32 byte accesses	Onelevel Scheme: 64 byte accesses	Onelevel Scheme: 256 byte accesses	Hierarchical Scheme: 1 KB accesses
Cholesky	68207	35194	18329	4812	1444
LU	25165	12621	6365	1152	123
Radix	64	64	64	64	64

Table 5.6: Number of Shared Reads Satisfied by Disk Accesses. Disk is accessed at the granularity of blocks with the one-level scheme, and at the granularity of pages in the hierarchical scheme.

In Table 5.6 and Table 5.7, we list the number of shared read misses that are

satisfied from disk and the number of shared write misses that are satisfied from disk respectively, for various granularities of disk access. As expected, the number of accesses satisfied from disk is higher when granularity of disk access is small. The difference between the number of shared reads satisfied from disk in the one-level and hierarchical schemes is a very small fraction of the total number of shared reads (see the total number of shared reads in Table 5.5 in Section 5.1.2) in the application. Consequently, in Figure 5.14, there is only negligible benefit in terms of read miss rate to the larger transfer size from disk in the hierarchical scheme.

Application	Number of Shared Write Accesses Satisfied by Disk Access				
	Onelevel Scheme: 16 byte accesses	Onelevel Scheme: 32 byte accesses	Onelevel Scheme: 64 byte accesses	Onelevel Scheme: 256 byte accesses	Hierarchical Scheme: 1 KB accesses
Cholesky	142810	71966	36474	9707	2880
LU	3	3	3	3	3
Radix	164097	82048	41024	10256	2564

Table 5.7: Number of Shared Writes Satisfied by Disk Accesses. Disk is accessed at the granularity of blocks with the one-level scheme, and at the granularity of pages in the hierarchical scheme.

The difference between the number of shared writes satisfied from disk in the one-level scheme with 16 byte blocks and in the hierarchical scheme is roughly 5% of the total number of shared writes in Cholesky and is roughly 2.5% of the total number of shared writes in Radix (Table 5.7). In Cholesky, it appears that the larger granularity of disk access in the hierarchical scheme allows more subsequent accesses to be satisfied locally leading to lower miss rate in the hierarchical scheme

(see Figure 5.14) than in the one-level scheme at small block sizes. This is not the case in Radix. In the Radix application, after each phase of sorting, the keys being sorted are permuted to their new position after that stage (Section 5.1.2). These shared writes are satisfied by disk accesses. Since the positions updated by different nodes are mapped to the same page, the larger transfer size from disk in the hierarchical scheme does not provide any benefit in terms of local miss rate (there is a benefit of reduced wait for disk accesses). As discussed previously, writes to the same page by other nodes invalidate sections of the page forcing subsequent accesses to issue a remote request. With the LU application, the smaller granularity of disk access in the one-level scheme can actually reduce the number of non-local writes (although more latency is incurred in more disk accesses). In the LU application, processes read their data before writing to them. When the initial read misses are satisfied by transfering small blocks from the disk, the block is likely to be private to the node allowing the subsequent write to hit locally. When data is transfered from disk in the granularity of pages in the hierarchical scheme, the page is installed in the node that receives the page from disk (the node that first reads the page). Read misses in other nodes are satisfied by requesting read copies from the node that received the page from disk (the page owner). Since these copies only allow read access, subsequent writes to the block cannot be satisfied locally and require remote access.

5.1.6.2. Classification of Protocol Flows Used to Satisfy Non-Local Requests

Our classification of the protocol flows used to satisfy remote requests in our coherence protocol is based on the observation that different protocol flows differ primarily in the sequence of messages and node hops needed to reach the owner of the block. Once the owner is reached, the protocol flow follows a fixed pattern for all read requests and another fixed pattern for all write requests. For example, a read request can arrive at the owner of the requested block in one of several ways — directly from the requester, forwarded by the page owner, or forwarded by the default owner. However, when a read request reaches the owner of the required block, in our coherence protocols, a copy of the block is always sent directly to the requester. Likewise, when a write request has reached the current owner of the block, the ownership of the block is always granted through the page owner (in the hierarchical scheme) or through the default owner (in the one-level scheme).

The number of messages exchanged to reach the owner of the block in a protocol flow is indicative of the complexity of the protocol flow. A protocol flow that requires a larger number of these messages incurs higher network latency and potentially incurs higher latency in accessing more coherency tables at intermediate nodes. Hence, we classify protocol flows by the number of messages exchanged in the protocol flow to reach the owner of the block. We refer to the sequence of protocol message and node hops used to reach the owner of the requested block as the *hop-sequence* of the protocol flow. The *hop-sequence type number* of a protocol flow is the number of messages exchanged before the owner of the block is reached (i.e., the number of messages in the *hop-sequence*).

To illustrate this classification, consider a few examples. Suppose a node

requires read access to a block that it does not possess locally. If the node knows the identity of the block owner, the read request can be sent to the owner of the block directly. Since only one message was used to reach the owner of the block, the read request is satisfied by a protocol flow with *hop-sequence* type of 1. If the requesting node does not know the identity of the block owner, the read request may be sent to the page owner who forwards it to the block owner. The *hop-sequence* type of this protocol flow is 2 since it uses two messages to reach the owner of the block. Later in this subsection, we discuss in detail more examples of *hop-sequences* of different type numbers that are used to service read and write requests in the one-level and hierarchical schemes.

It should be noted that our classification is based only on the number of messages and not on the types of messages that are sent or the identity of intermediate nodes (page owner, default owner etc.) that are accessed. Hence, different protocol flows that use different sequences of messages or different sequences of nodes have the same *hop-sequence* type numbers if they use the same number of messages. For example, a protocol flow that forwards a read request through the page owner to the block owner has an *hop-sequence* type of 2. A protocol flow that forwards the read request through the default owner to the page owner which also owns the block (the page owner is also the block owner) also has an *hop-sequence* type of 2. Alternative classifications based on the sequence of messages exchanged, on the sequence of nodes accessed or based on the set of tables looked up are also possible. Our classification based on the number of messages is simple and captures the key differences in the protocol flows.

Table 5.8 presents example *hop-sequences* of protocol flows used to satisfy

Hop-sequences of Protocol for Read Access in One-level Scheme					
Type of hop-sequence of protocol	Nodes (Sequence)	Messages (Sequence)	Tables Accessed at Node	Table Access Latency (cycles)	Comments
Type 0					Not possible
Type 1	Requestor → Default Owner → Requestor	RREQ RDCPY	PPT PPT+SPT Cache	3 7 1	MPT access may be needed
Type 2	Requestor → Default Owner → Page Owner → Requestor	RREQ RREQ RDCPY	PPT PPT+MPT PPT+SPT Cache	3 43 7 1	MPT access may not be needed Copy Directory updated in parallel
Type 3	Requestor → Default Owner → Page Owner → Default Owner → Requestor	RREQ RREQ RREQ RDCPY	PPT PPT+MPT PPT PPT+SPT Cache	3 43 3 7 1	MPT access may not be needed Page owner has changed Default owner is also page owner
Type 4	Requestor → Default Owner → Page Owner → Default Owner → New Page Owner → Requestor	RREQ RREQ RREQ RREQ RDCPY	PPT PPT+MPT PPT PPT+MPT PPT+SPT Cache	3 43 3 43 7 1	MPT access may not be needed Page owner has changed MPT access may not be needed Copy Directory updated in parallel
Type 5	Requestor → Default Owner → Page Owner → Default Owner → Page Owner → Default Owner → New Page Owner → Requestor	RREQ RREQ RREQ RREQ RREQ RREQ RDCPY	PPT PPT+MPT PPT PPT+MPT PPT PPT+MPT PPT+SPT Cache	3 43 3 43 3 43 7 1	MPT access may not be needed Page owner has changed MPT access may not be needed Page owner has changed MPT access may not be needed Copy Directory updated in parallel

Table 5.8: Hop-sequences of protocol for read access in one-level scheme.

remote read requests in the one-level scheme. An example *hop-sequence* is listed for each *hop-sequence* type number between 0 and 5. In each example, the table identifies the sequence of nodes involved in the *hop-sequence* (the Requester initiates the access), the messages sent by these nodes in sequence, the mapping

tables accessed at each node and the latency incurred in accessing tables at each node.

A *hop-sequence* type 0 is not valid since it implies a read-hit in the requesting node. The example for *hop-sequence* type 1 describes the protocol flow where the read request is received and serviced by the default owner itself. The read access initiates a search of the local Present Pages Table (PPT) (or the Present Block Table). When the block is not found in the PPT, the read request *RREQ* is sent to the default owner by the requesting node. When the default owner receives the remote read request, the PPT, and then the SPT are looked up to determine the owner of the block. When the default owner is also the owner of the block, a read copy is sent back to the requester with the message *RDCPY*. When the read copy is received at the requester, the read-copy of the data is directly passed to the cache at the requester for immediate access. The latency of accessing tables at each node is computed based on the access latency of individual tables specified in Table 5.1.

A protocol flow with *hop-sequence* type 2 is required when the requested block is owned by some node other than the default owner. In this case, the default owner must identify the current owner by looking up its Migrated Page Table (MPT) (or Migrated Block Table). The read request is forwarded to the owner by the default owner. The reply is sent directly by the owner of the block. Protocol flows with *hop-sequence* type greater than 2 are possible when the owner of the block changes while the read request is in progress. In these cases, the owner prior to the change forwards the request back to the default owner.

As we mentioned earlier, alternative examples with a different sequence of nodes and messages are possible for many *hop-sequence* types. In addition, even

for the example *hop-sequences* illustrated here, the mapping tables accessed at the nodes can be different. For instance, an MPT access may be needed at the default owner for the read access with *hop-sequence* type 1 if the page is not present at the default owner and the default owner checks the MPT to find that no other node owns the page, i.e., the page is on disk.

Table 5.9 presents the example cases for read accesses in the hierarchical scheme. The example for *hop-sequence* type 1 describes the protocol flow where the read request is received and serviced by the page owner itself. The Present Pages Table (PPT) is first searched at the node issuing the read access. In the example scenario, the page that holds the accessed data is identified to be in the shared state. A lookup of the Shared Page Table (SPT) entry for the page is initiated. When the required block is indicated absent in the SPT, the identity of the page owner is ascertained from the SPT entry. A read request *RREQ* is sent to this page owner. When the page owner receives the remote read request, the PPT, and then the SPT are looked up to determine the owner of the block. When the page owner also possesses ownership of the block, the read copy (*RDCPY*) is sent back to the requester.

The example for *hop-sequence* type 2 deviates from the previous example when the requested block is owned by some node other than the page owner. In this case, the identity of the block owner is determined by the page owner from the copy directory (CoD). The read request (*RREQ*) is forwarded to the block owner. When the request is received by the block owner, the PPT and the SPT are looked up to ascertain ownership of the block, after which the read copy (*RDCPY*) is sent directly to the requester.

Hop-sequences of Protocol for Read Access in Two-level Scheme					
Type of hop-sequence of protocol	Nodes (Sequence)	Messages (Sequence)	Tables Accessed at Node	Table Access Latency (cycles)	Comments
Type 0					Not possible
Type 1	Requestor → Page Owner → Requestor	RREQ RDCPY	PPT+SPT PPT+SPT Cache	7 7 1	Page owner identified by SPT Page owner is also block owner
Type 2	Requestor → Page Owner → Block Owner → Requestor	RREQ RREQ RDCPY	PPT+SPT PPT+SPT+CoD PPT+SPT Cache	7 14 7 1	Page owner identified by SPT Forwarded to block owner Copy Directory updated in parallel
Type 3	Requestor → Default Owner → Requestor → Page Owner → Requestor	RREQ PGOWN RREQ RDCPY	PPT PPT+SPT PPT PPT+SPT Cache	3 7 3 7 1	Page owner not known Bi-lateral response Page owner was identified Page owner is also block owner
Type 4	Requestor → Default Owner → Requestor → Page Owner → Block Owner → Requestor	RREQ PGOWN RREQ RREQ RDCPY	PPT PPT+SPT PPT PPT+SPT+CoD PPT+SPT Cache	3 7 3 14 7 1	Page owner not known Bi-lateral response Page owner was identified Forwarded to block owner Copy Directory updated in parallel
Type 5	Requestor → Page Owner → Block Owner → Default Owner → Requestor → Page Owner → Requestor	RREQ RREQ RREQ PGOWN RREQ RDCPY	PPT+SPT PPT+SPT+CoD PPT+SPT PPT+SPT PPT+SPT PPT+SPT Cache	7 14 7 7 7 7 1	Page owner identified by SPT Forwarded to block owner Block owner has changed Response to requestor Read request re-issued Page owner is also block owner

Table 5.9: Hop-sequences of protocol for read access in hierarchical scheme.

When no blocks from the page that has the required data are present at the node issuing the request (examples for *hop-sequence* types 3 and 4), the node cannot identify the owner of the page (no PPT entry for the page and thus no SPT access). Hence, the read request is sent to the default owner. Since the protocol uses bilateral exchanges, when the default owner is not the owner of the page, a

bilateral response identifying the page owner (*PGOWN*) is sent back to the requester. In the example situations for *hop-sequence* types 3 and 4, the default owner possesses copies of the blocks of the page allowing it to determine the page owner by accessing the SPT, without requiring Migrated Page Table (MPT) access. When the *PGOWN* message is received, the request is sent directly to the page owner by the requester. When the page owner is also the owner of the block, the owner of the data is reached with three messages (*hop-sequence* type 3). The subsequent protocol flow is similar to the example for *hop-sequence* type 1. When the required block is owned by some node other than the page owner, an additional message (*hop-sequence* type 4) is needed to reach the owner of the block. Protocol flows with *hop-sequence* type number greater than 4 can occur when block or page ownership moves during the access.

Table 5.10 presents the example cases for write accesses in the one-level scheme. When the node issuing the write access is also the owner of the block, no messages are required to reach the owner of the block (*hop-sequence* type 0). Present Pages Table (PPT), Shared Pages Table (SPT) and copy directory accesses are needed to determine the locations of read copies. The copies are invalidated (*INV*). When all acknowledgments (*ACK*) are received, the write access is allowed to progress.

When the node issuing the write access is not the owner of the block, the write request *WREQ* must be sent to the default owner. If the default owner is also the owner of the block (*hop-sequence* type 1), the default owner performs the invalidations. When all acknowledgments are received, the ownership copy (*WDATA*) is sent to the requester after the identity of the new owner is updated in

Hop-sequences of Protocol for Write Access in One-level Scheme					
Type of hop-sequence of protocol	Nodes (Sequence)	Messages (Sequence)	Tables Accessed at Node	Table Access Latency (cycles)	Comments
Type 0	Requestor → Sharing Node → Requestor	INV ACK	PPT+SPT+CoD PPT+SPT Cache	14 7 1	Access by page owner Copy invalidated When all ACKs received
Type 1	Requestor → Default Owner → Sharing Node → Default Owner → Requestor	WREQ INV ACK WDATA	PPT PPT+SPT+CoD PPT+SPT PPT+SPT+MPT Cache	3 14 7 47 1	 Default owner is also page owner Copy invalidated New page owner updated in MPT
Type 2	Requestor → Default Owner → Page Owner → Sharing Node → Page Owner → Default Owner → Requestor	WREQ WRBCK INV ACK BLKBCK WDATA	PPT PPT+SPT PPT+SPT+CoD PPT+SPT PPT+SPT PPT+SPT+MPT Cache	3 7 14 7 7 47 1	 MPT access may be needed Invalidate for writeback Copy invalidated Write back (after all ACKs) New page owner updated in MPT
Type 3					Not possible
Type 4					Not possible
Type 5					Not possible

Table 5.10: Hop-sequences of protocol for write access in one-level scheme.

the Migrated Page Table (or Migrated Block Table). The example for *hop-sequence* type 2 describes the case when the default owner is not the owner of the block. In this case, a second message (*WRBCK*) is needed to reach the owner of the block. Upon receiving the write-back request, invalidates are sent by the block owner to all read copies. When all acknowledgments are received, the block is sent back (*BLKBCK*) to the default owner. The ownership is forwarded to the requester (*WDATA*) by the default owner after the MPT is updated.

The MPT entry is locked by the default owner when a write request is in progress. If the MPT entry is locked when a new write request is received, the new

write request must be retried. A retried write request will follow one of the three *hop-sequences* described earlier. Hence, there are no protocol flows with *hop-sequence* type greater than 2.

The example *hop-sequences* for write accesses in the hierarchical scheme are shown in Table 5.11. The example for *hop-sequence* type 0 is identical to the equivalent example discussed for writes in the one-level scheme. The examples for *hop-sequence* types 1 and 2 are also similar to the equivalent cases discussed for writes in the one-level scheme. The only difference is that ownership transfers at writes are coordinated by the default owner in the one-level scheme, while they are coordinated by the page owner in the hierarchical scheme. In the hierarchical scheme, the page owner commits the transfer of block ownership by updating its SPT, without any changes to the MPT. *Hop-sequence* types 1 and 2 are possible in the hierarchical scheme only when the node issuing the write can identify the page owner from the SPT entry for the page (i.e., the node must possess other blocks of the page).

When the node issuing the write cannot identify the page owner (no SPT access), the write request must be sent to the default owner. This is the case in the examples for *hop-sequence* types 3 and 4. As we discussed, for read requests in the hierarchical scheme, the default owner must be queried for the identity of the page owner using a bilateral exchange. The examples for *hop-sequence* type 3 and type 4 differ from the examples for *hop-sequence* type 1 and type 2 only in the need for two additional messages for the bilateral exchange. In the example for *hop-sequence* type 3, the page ownership changes during the write access.

Hop-sequences of Protocol for Write Access in Two-level Scheme					
Type of hop-sequence of protocol	Nodes (Sequence)	Messages (Sequence)	Tables Accessed at Node	Table Access Latency (cycles)	Comments
Type 0	Requestor →	INV	PPT+SPT+CoD	14	Access by block owner
	Sharing Node →	ACK	PPT+SPT	7	Copy invalidated
	Requestor		Cache	1	When all ACKs received
Type 1	Requestor →	WREQ	PPT+SPT	7	Page owner identified by SPT
	Page Owner →	INV	PPT+SPT+CoD	14	Page owner is also block owner
	Sharing Node →	ACK	PPT+SPT	7	Copy invalidated
	Page Owner →	WDATA	PPT+SPT	7	Copy Directory updated in parallel
	Requestor		Cache	1	
Type 2	Requestor →	WREQ	PPT+SPT	7	Page owner identified by SPT
	Page Owner →	WRBCK	PPT+SPT+CoD	14	Request block write-back
	Block Owner →	INV	PPT+SPT+CoD	14	Invalidate for write-back
	Sharing Node →	ACK	PPT+SPT	7	Copy invalidated
	Block Owner →	BLKBCK	PPT+SPT	7	Write back (after all ACKs)
	Page Owner →	WDATA	PPT+SPT	7	Copy Directory updated in parallel
	Requestor		Cache	1	
Type 3	Requestor →	WREQ	PPT	3	Page owner not known
	Default Owner →	PGOWN	PPT+SPT	7	Bi-lateral response
	Requestor →	WREQ	PPT	3	Page owner was identified
	Page Owner →	INV	PPT+SPT+CoD	14	Page owner is also block owner
	Sharing Node →	ACK	PPT+SPT	7	Copy invalidated
	Page Owner →	WDATA	PPT+SPT	7	Copy Directory updated in parallel
	Requestor		Cache	1	
Type 4	Requestor →	WREQ	PPT	3	Page owner not known
	Default Owner →	PGOWN	PPT+SPT	7	Bi-lateral response
	Requestor →	WREQ	PPT	3	Page owner was identified
	Page Owner →	WRBCK	PPT+SPT+CoD	14	Request block write-back
	Block Owner →	INV	PPT+SPT+CoD	14	Invalidate for write-back
	Sharing Node →	ACK	PPT+SPT	7	Copy invalidated
	Block Owner →	BLKBCK	PPT+SPT	7	Write back (after all ACKs)
	Page Owner →	WDATA	PPT+SPT	7	Copy Directory updated in parallel
	Requestor		Cache	1	
Type 5	Requestor →	WREQ	PPT	3	Page owner not known
	Default Owner →	PGOWN	PPT+SPT	7	Bi-lateral response
	Requestor →	WREQ	PPT	3	Page owner was identified
	Page Owner →	WREQ	PPT	3	Page owner has changed
	Default Owner →	WRBCK	PPT+SPT+CoD	14	Default owner is also page owner
	Block Owner →	INV	PPT+SPT+CoD	14	Invalidate for write-back
	Sharing Node →	ACK	PPT+SPT	7	Copy invalidated
	Block Owner →	BLKBCK	PPT+SPT	7	Write back (after all ACKs)
	Default Owner →	WDATA	PPT+SPT	7	Copy Directory updated in parallel
	Requestor		Cache	1	

Table 5.11: Hop-sequences of protocol for write access in hierarchical scheme.

5.1.6.3. Breakdown of Non-Local Requests by Hop-Sequence Type of Protocol Flow

During simulation, the *hop-sequence* type number of the protocol flow used to satisfy each remote request was monitored. To eliminate cold-start accesses from this measurement, non-local accesses which fetch the data from disk are not included in the measures. The number of such accesses which fetch the data from disk is tabulated in Tables 5.6 and 5.7 in Section 5.1.6.1. In this subsection, these measurements are used to present a breakdown of the frequencies of occurrence of different classes (*hop-sequence* types) of protocol flows. Results are presented for a range of block sizes, for reads and writes, with the one-level and hierarchical schemes.

Figure 5.15 shows the breakdown of remote read accesses in the one-level scheme simulation, based on their *hop-sequence* type. For each application, we measured the breakdown in several simulation runs, each at a different block size. The breakdown is plotted at several block sizes from 16 bytes through 1 KB in Figure 5.15. At any block size, the total height of the bar is the total number of remote read accesses during the entire execution of the application. The number of remote read accesses that were satisfied using protocol flows of a specific *hop-sequence* type number is indicated by the height of the segment of the bar that corresponds to that type.

In most cases, the total number of non-local read accesses decreases with increasing block size. This is because larger block sizes capture more spatial locality. When large blocks are transfered to service a read miss, more subsequent accesses are likely to hit on the same block without requiring remote access. In the

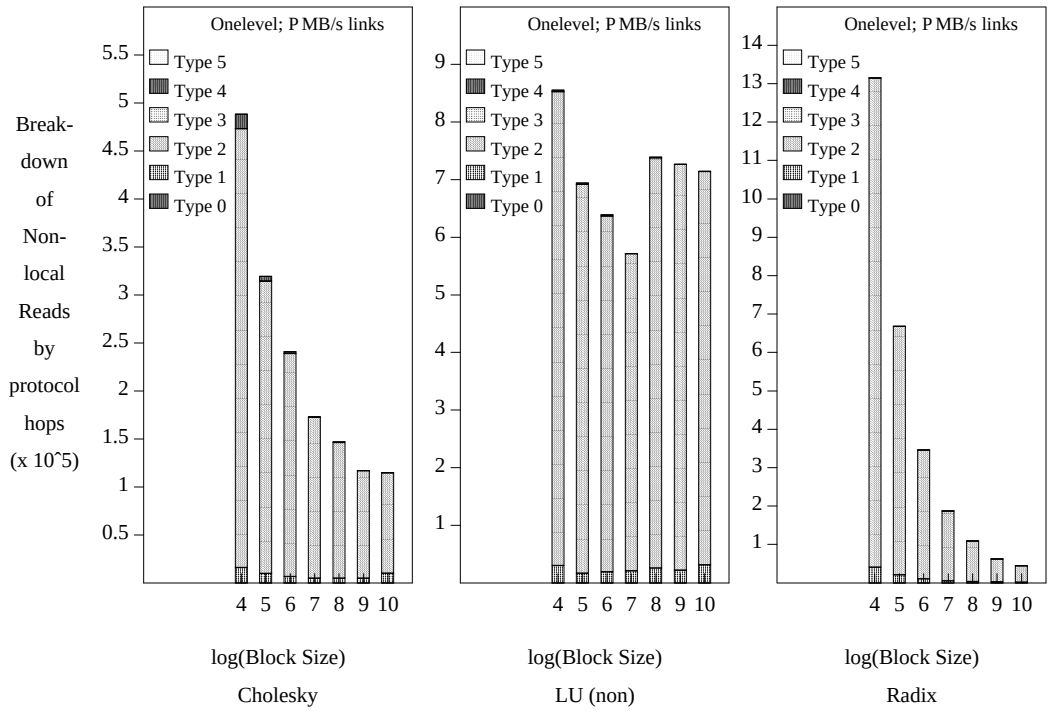


Figure 5.15: Break down of read accesses by hop-sequence type in the one-level scheme.

case of LU, when the block size is increased beyond 128 bytes, data accessed by different nodes get mapped to the same block. As a result of this false sharing, a larger number of read access require remote access.

From Figure 5.15, it is also clear that most read accesses have *hop-sequence* type 2 in the one-level scheme. This *hop-sequence* type corresponds to the scenario where the read request is sent to the default owner and then forwarded by the default owner to the current page (block) owner (see Table 5.8). The hop to the default owner is expected since a node requesting a read copy typically does not know the owner of the page (block) in the one-level scheme. The node knows the identity of the page owner only when it is the default owner. The additional hop to

forward the request from the default owner to the current owner is also common because the default owner is rarely also the owner. This is due to two reasons. First, the ownership of a block keeps changing because different nodes write to the same block. Second, the assignment of blocks to default owner nodes and processes to nodes are done independently without regard to the behavior of the application. When the block is not written by multiple processes in the application, this points to an opportunity for improving performance. Application performance can be improved by assigning processes to nodes which are the default owner of blocks which are written to mostly by these processes. Figure 5.15 also shows that accesses are rarely satisfied by *hop-sequence* type 1 protocol flows. This occurs when the node requesting read access is also the default owner or when the default owner is also the page owner.

Figure 5.16 shows the breakdown for read accesses in the hierarchical scheme. The figure shows that a greater proportion of accesses are satisfied by protocol flows with *hop-sequence* type 1 in the hierarchical scheme than in the one-level scheme. This indicates that the node issuing the request frequently sends the request to the owner of the block directly. Since only the page owner knows the block owner, we can draw two conclusions from this observation. First, the requesting node frequently has other blocks of the page and, hence, knows the identity of the page owner. Second, it indicates that the page owner owns the required block in the hierarchical scheme more frequently than the default owner owns the block in the one-level scheme. This is to be expected since the page owner is determined dynamically based on the application's access pattern to the page, while the default owner is determined statically, independent of the

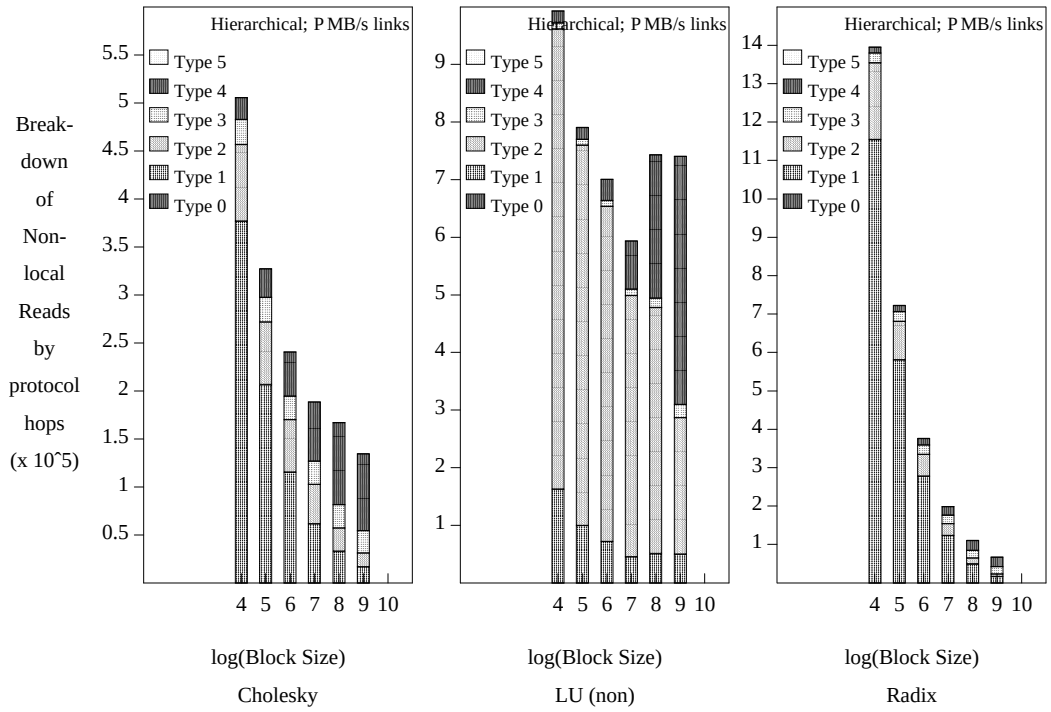


Figure 5.16: Break down of read accesses by hop-sequence type in the hierarchical scheme.

application.

With Cholesky and Radix, for small blocks, the predominance of requests serviced by protocol flow with *hop-sequence* type 1 shows that it is common that the requesting node knows the page owner and the page owner is also the block owner. As block size increases, the first access to a page brings more of the data needed from the page, allowing more subsequent accesses to be satisfied locally. Also, since there are fewer blocks per page at larger block sizes, there are fewer additional blocks to be requested after the first access to the page. For both these reasons, it is less likely to have many accesses which will reach the block owner in one hop at larger block sizes. This leads to the interesting result where, with

increasing block size, the total number of remote reads in Cholesky and Radix decreases mostly by the reduction in the number of reads with *hop-sequence* type 1 protocol flow. In the application kernel LU, forwarding from the page owner to the block owner is more likely since data modified by multiple processors are mapped to the same 1 KB page (recall from Section 5.1.2 that the LU kernel is a less-optimized version where data modified by a process is not allocated contiguously). This results in a larger proportion of accesses with *hop-sequence* type 2.

Protocol flows with *hop-sequence* type higher than 2 are needed in the hierarchical scheme when the default owner must also be contacted (see Table 5.9). This occurs more frequently with large blocks. With larger blocks, there are fewer blocks per page and there is greater false-sharing. Hence, when an access is issued, it is more often the case that other blocks of the page are not resident locally. This increases the proportion of accesses requiring default owner lookup (since page owner is not known locally). The higher false sharing associated with the larger block size also increases the proportion of accesses where the request must be forwarded to the block owner by the page owner. Both these factors contribute to a higher count of of accesses that are satisfied by protocol flows with *hop-sequence* types 3 and 4 at higher block sizes. (Recall that default owner lookup requires two additional messages due to the bilateral exchange.)

Figure 5.16 also shows that with increasing block size, the number of reads satisfied by protocol flows with *hop-sequence* type 4 increases more sharply than the number of reads satisfied by protocol flows with *hop-sequence* type 3. This is most likely due to the fact that larger blocks have one effect when most of a page is modified and a different effect when most of a page is unmodified. When most of

a page is unmodified, most of the page resides with the page owner and an access to the page from a node that has no other blocks of the page is more likely to be satisfied by a protocol flow with *hop-sequence* type 3. When the block size increases, the first access to a page brings more of the data. But, it does not affect the number of accesses that are issued when a node has no other blocks of the page. Therefore, the number of accesses with *hop-sequence* type 3 is largely unchanged when the block size is increased. When most of a page is modified, data of the page is likely to be owned by nodes other than the page owner (nodes which modify the data). In this case, an access to the page from a node that has no other blocks of the page is more likely to be satisfied with a protocol flow of *hop-sequence* type 4. As before, with larger blocks, more of the data is brought by the first access to the page. However, with larger blocks, the number of writes to a block is also higher due to the higher false sharing. Data brought by the first access to the page is, hence, more frequently invalidated. Hence, it is more likely (with pages where most of the page is modified) that the next read access is also issued when the node has no other blocks of the page, requiring default owner lookup. Thus, the larger block size increases the number of reads which must be satisfied using protocol flow with *hop-sequence* type 4. As mentioned earlier, with LU, data modified by multiple processors get mapped to the same 1 KB page. Hence, the block is frequently owned by a node other than the page owner. This results in a larger number of protocol flows with *hop-sequence* type 4 than in other applications.

We can compare the behavior of remote read accesses in the one-level and the hierarchical schemes (Figures 5.15 and 5.16). Most remote read accesses reach the

owner of the block within two protocol hops in both schemes. However, with the hierarchical scheme, it is more likely for a remote read access to require more than two protocol hops. This is because the hierarchical scheme uses bilateral exchanges to query the default owner and supports an additional level of ownership.

Figures 5.15 and 5.16 also show that at small block sizes, the total number of non-local read accesses is marginally higher in the hierarchical scheme than in the one-level scheme using the same block size. As we discussed in Section 5.1.6.1, this is most likely due to the fact that SPT entry contention can be higher in the hierarchical scheme than the one-level scheme at small block sizes. Since each SPT entry corresponds to a page of shared data in the hierarchical scheme, the hierarchical scheme has fewer SPT entries (e.g., 128 entries in the hierarchical scheme and 2048 entries in the one-level scheme with 16-byte blocks in our simulations, as listed in Table 5.2 in Section 5.1.2) than the one-level scheme. Hence, it is more likely that shared data that is mapped to local memory is evicted to free an SPT entry in the hierarchical scheme. Subsequent reads that would have been satisfied locally by the evicted page can only be satisfied with non-local access. This can lead to a small increase in the number of non-local read accesses in the hierarchical scheme.

Figure 5.17 shows the breakdown of remote write accesses based on their *hop-sequence* type in the one-level scheme. Protocol flows with *hop-sequence* type 2 are predominant. This is expected since it corresponds to the most common case when the default owner transfers the write ownership to the requester from another node. Cholesky and LU display a significant number of cases when a

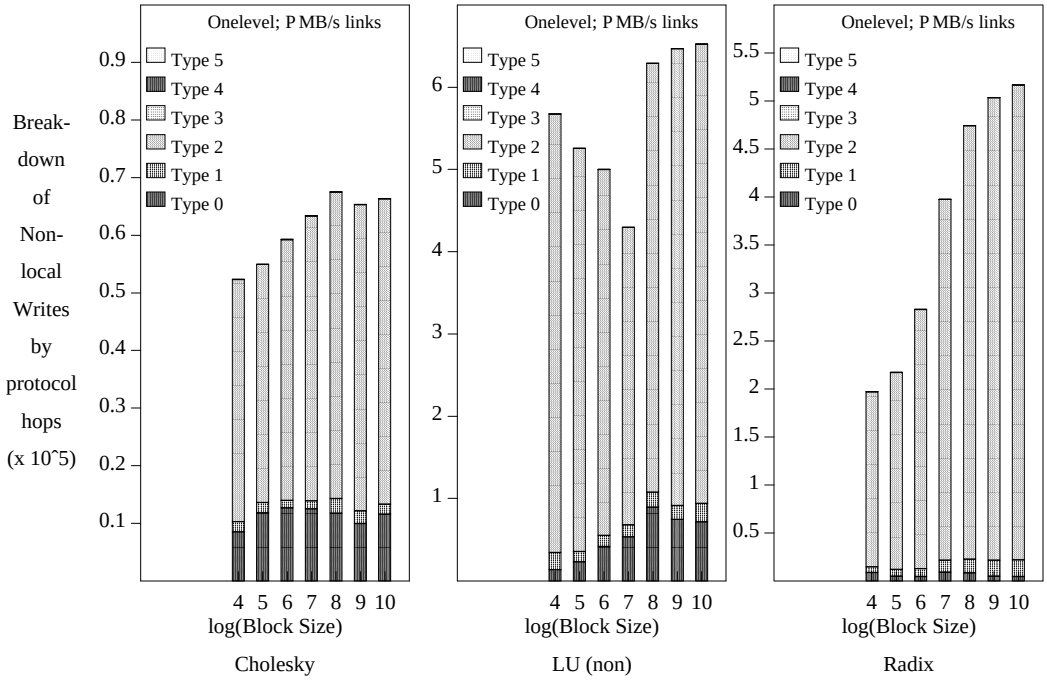


Figure 5.17: Break down of write accesses by hop-sequence type in the one-level scheme.

shared block is written by the owner itself (*hop-sequence* type 0), requiring invalidation only.

The breakdown of remote write accesses based on their *hop-sequence* type in the hierarchical scheme is shown in Figure 5.18. More write accesses are satisfied with *hop-sequence* type 1 protocol flow in the hierarchical scheme than in the one-level scheme. As we described in the discussion of read access behavior, this indicates that frequently the requesting node knows the identity of the page owner, and frequently the page owner is also the block owner. In the application kernel LU, the page owner frequently does not own the block, accounting for a higher proportion of accesses serviced by *hop-sequence* type 2 protocol flows (similar to the read access behavior discussed earlier).

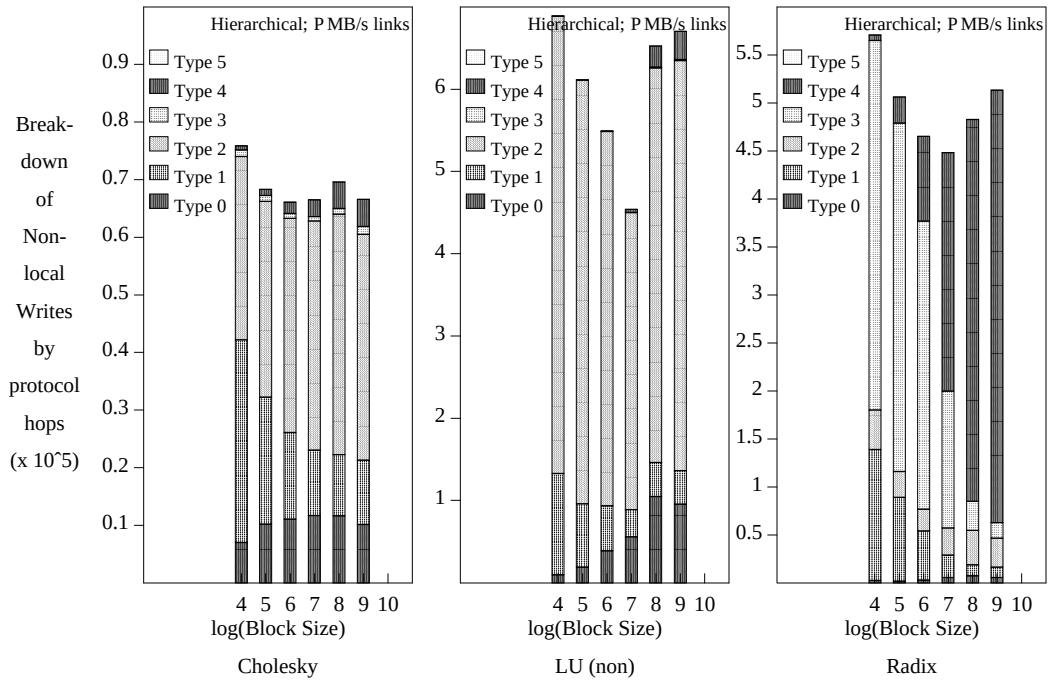


Figure 5.18: Break down of write accesses by hop-sequence type in the hierarchical scheme.

The behavior of write accesses in the Radix kernel is significantly different on the one-level and the hierarchical systems (Figures 5.17 and 5.18). We describe the reasons for the differences below. In the Radix kernel, processes initialize (write) the keys to be sorted at the start. In addition, after each phase of the sort, the keys being sorted are written to their new positions in the array. These array locations may not be mapped to the local memory of the processor writing the location. Therefore, many write accesses are first accesses to a page. Since these pages are not mapped to the physical memory of the requesting node at the start of execution, these write accesses require a default owner lookup. This results in a higher proportion of protocol flows with *hop-sequence* types 3 and 4. At higher block sizes, there are fewer blocks per page and higher false sharing. As

previously described in the discussion of read access behavior in the hierarchical scheme, these lead to a high incidence of protocol flows where the default owner, the page owner and the block owner are all accessed (*hop-sequence* type 4).

Another difference in the behavior of Radix on the two schemes is that with small blocks the overall number of non-local write accesses is higher with the hierarchical scheme. This difference is partly an artifact of the fact that the disk is accessed in block-sized chunks in the one-level scheme, while it is accessed in page-sized chunks in the hierarchical scheme. In the one-level scheme, these cold-start writes fetch the required block from disk and are, hence, not counted in our non-local write access counts (e.g., about 1.6×10^5 accesses with 16-byte blocks, as listed in Table 5.7 in Section 5.1.6.1). In the hierarchical scheme, the first such write access brings the entire page from disk. All subsequent writes to blocks of the page must be satisfied by the page owner. These are not disk accesses and hence, are included in the count of non-local write accesses in the hierarchical scheme. The higher number of non-local accesses in the hierarchical scheme at small blocks can also be attributed to the higher contention for SPT entries in the hierarchical scheme.

It is also apparent from Figures 5.17 and 5.18 that the overall number of non-local write accesses do not decrease with increasing block size, unlike the number of non-local read accesses. This is partly due to the fact that these figures do not include the counts of non-local writes that were satisfied from disk. We have previously discussed the effect of block size on the total number of non-local writes (including those that are satisfied by disk accesses) in Section 5.1.6.1. As we pointed out in Section 5.1.6.1, the total number of non-local write accesses does

decrease slightly at small block sizes. With large blocks, it is more likely that the block is shared by other nodes. Since writes require exclusive access to the block, more writes require remote access to invalidate these copies at larger block sizes.

5.1.6.4. Breakdown of Non-Local Request Latency

The latency of a non-local access depends on the network latency of the messages exchanged for the access and the access latency of coherency tables accessed at nodes involved in the access. We have investigated the contributions of message latency and table access latency to the latency of non-local accesses. We describe our investigation and its results below.

The most direct way to analyze the latency of non-local accesses is to trace the progress of each non-local access. To accomplish this trace, the simulator must be instrumented to follow the protocol flow's progress through different nodes. At each stage of the protocol flow, the latency of the messages exchanged and tables accessed must be measured and accumulated by the simulator. For these capabilities, the messages must be tagged with data structures that identify the source non-local access which caused the generation of the message and track the latencies accumulated until that stage of the protocol flow. The data collection mechanisms must also address the issue that asynchronous replacement of data can prematurely satisfy the non-local access before the completion of the protocol flow. To illustrate this issue, consider the case of an asynchronous writeback of data when a write request is in progress for the same block. Suppose that the protocol flow for the write request is at the stage when the page owner sends a write back request to the current block owner. If the block is written back by the

block owner to the page owner before the write-back request is received (due to asynchronous replacement of the data), the message carrying the block back (*BLKBCK*) will not contain any information regarding the non-local write access. Additional data structures must be maintained in the page owner to handle these situations.

Our simulator does not support the functionality to trace individual non-local accesses. However, the simulator traces the progress of individual messages through the network to measure the transmission latency of messages through the network links and the network queuing delay encountered by messages. We use these measurements and the measurements we have taken to characterize the occurrence of protocol flows with different *hop-sequence* types (Section 5.1.6.3) to study the breakdown of remote access latency. This allows us to indirectly (and less precisely) break down remote access latency into message latency and table access latency components. We describe our methodology and results below.

During application execution, the simulator measures the link latency and queuing delay of each message that is exchanged. The link latency of a message measures the latency of the message from source to destination in the absence of any contention in the network. The link latency of a message is determined by the bandwidth of the links, the size of the message and the number of links traversed to reach the destination from the source. The queueing delay of a message measures the additional delay caused by buffering within the network during network contention. The overall latency incurred by a message is the sum of its link latency and queuing delay components. The simulator computes the average link latency and average queuing delay independently for control messages and block data

messages. These averages are computed as the arithmetic mean of the measures collected for each individual message. Table 5.12 lists the average link latency and average queuing delay of control messages and block data messages measured by the simulator in simulation runs with network link bandwidth of 8P MB/s. The averages are listed for simulation runs with block sizes of 64 bytes and 256 bytes both for the one-level scheme and the hierarchical scheme.

In Tables 5.8 through 5.11 (Section 5.1.6.2), we have previously presented nominal cases of protocol flows for each of *hop-sequence* types 0 through 5. In each case, the tables identify the latency of tables accessed during the protocol flow. The tables also identify the messages that are exchanged during the protocol flow. We calculate the average link latency and average queuing delay of these messages using the average end-to-end link latency per message and the average end-to-end queuing delay per message measured (independently for control and block data messages) during the simulation. This allows us to identify the table access, message link latency and message queueing delay components of each *hop-sequence* type of protocol flow listed in Tables 5.8 through 5.11. As previously presented (Section 5.1.6.3), our simulator measures the number of occurrences of each *hop-sequence* type of protocol flow during the entire execution. The average cost breakdown of non-local accesses is calculated by computing the average of the costs of individual *hop-sequence* types weighted by their frequency of occurrence.

It should be noted that the breakdown of remote access latency into table access and network latencies is based on the tables accessed and messages exchanged in the specific protocol flows listed in Tables 5.8 through 5.11

Application	Scheme	Block Size (bytes)	Network Measures for Control Packets (in processor cycles)		Network Measures for Block Data Packets (in processor cycles)	
			Link Latency	Queuing Delay	Link Latency	Queuing Delay
Cholesky	Onelevel	64	8.561	2.429	16.923	7.539
		256	8.841	19.688	40.835	49.946
	Hierarchical	64	9.139	3.013	17.119	4.096
		256	9.056	17.659	41.102	44.590
LU	Onelevel	64	8.538	2.227	16.482	1.694
		256	8.584	16.344	40.528	9.633
	Hierarchical	64	7.939	1.321	15.837	0.812
		256	8.801	11.789	40.784	6.588
Radix	Onelevel	64	9.074	2.480	17.087	2.819
		256	9.066	17.416	41.105	17.290
	Hierarchical	64	9.063	7.892	17.062	6.627
		256	9.080	15.316	41.068	12.417

Table 5.12: Average Link Latency and Queuing Delay of Network Messages. The measures are listed from simulation runs with 8P MB/s links. The average link latency is the average latency of the message through the links in the absence of network contention. The average queuing delay is the average additional delay due to buffering in the network when there is network contention.

(Section 5.1.6.2). As we have mentioned earlier, alternative protocol flows are possible during application execution. These protocol flows can have a different *hop-sequence* type number than our example protocol flows. Even when protocol flows have the same *hop-sequence* type numbers as one of our example protocol flows, they can access different tables incurring a different total table access latency. Our analysis does not account for these alternative protocol flows. We

report the measured average latency of remote accesses along with the results of our analysis. The inaccuracy in the analysis can be estimated by comparing the measured average remote access latency with the average remote access latency computed by our analysis.

Comparison of Cost of Hop-sequences of Protocol for Read Accesses				
Scheme	Type of hop-sequence of protocol	Total Table Access Latency (cycles)	Number of Control Messages	Number of Block Data Messages
One-level	Type 1	11	1	1
One-level	Type 2	54	2	1
One-level	Type 3	57	3	1
One-level	Type 4	100	4	1
One-level	Type 5	146	6	1
Two-level	Type 1	15	1	1
Two-level	Type 2	29	2	1
Two-level	Type 3	21	3	1
Two-level	Type 4	35	4	1
Two-level	Type 5	50	5	1

Table 5.13: Summary of cost of hop-sequences of protocol for read accesses.

For clarity, the cost components (table access latency, number of control messages and number of data messages) of each *hop-sequence* type of protocol flow are summarized in Table 5.13 and Table 5.14 (from Tables 5.8 through 5.11). Table 5.13 summarizes the costs of protocol flows for read accesses both on the one-level and on the hierarchical scheme. The message link latency for a protocol flow is the sum of the message link latencies for the control messages and the

message link latencies for the block data messages exchanged in the *hop-sequence* type of the protocol flow. The message queuing delay in the protocol flow can be calculated similarly.

Comparison of Cost of Hop-sequences of Protocol for Write Accesses				
Scheme	Type of hop-sequence of protocol	Total Table Access Latency (cycles)	Number of Control Messages (c is number of copies)	Number of Block Data Messages
One-level	Type 0	22	2+2*(c-1)	0
One-level	Type 1	72	3+2*(c-1)	1
One-level	Type 2	86	4+2*(c-1)	2
Two-level	Type 0	22	2+2*(c-1)	0
Two-level	Type 1	36	3+2*(c-1)	1
Two-level	Type 2	57	4+2*(c-1)	2
Two-level	Type 3	42	5+2*(c-1)	1
Two-level	Type 4	63	6+2*(c-1)	2
Two-level	Type 5	66	7+2*(c-1)	2

Table 5.14: Summary of cost of hop-sequences of protocol for write accesses.

Table 5.14 summarizes the costs of protocol flows for write accesses on the one-level and hierarchical schemes. The average number of copies invalidated during a write access — 'c' in Table 5.14 — is measured during simulation. We estimate the cost associated with invalidating multiple copies (c > 1) based on a model in which the node sending the invalidate messages sends them in a pipelined manner and the nodes receiving invalidate requests process their invalidate concurrently. With this model, the total latency of invalidating multiple copies is

the latency of invalidating the last copy. Also, the additional latency incurred by the write access due to the invalidation of multiple copies is the pipeline delay encountered by the last invalidate message at the node issuing the invalidations. We account for this additional delay by including a pipeline delay equal to the latency of one control message through one link for each extra (greater than one) copy that must be invalidated. We report this additional delay as part of the table access latency component of the write access protocol flow. The method of calculating the message link latency and message queuing delay for write protocol flows is the same as the method described for read protocol flows.

The methodology that we described above allows us to identify the contributions of coherency table access, network link latency and network queuing to the overall latency of remote accesses. As we have shown earlier, due to the multiple levels of ownership, protocol flows with higher *hop-sequence* type numbers are more likely in the hierarchically managed system. As described below, in order to account for the overhead associated with complicated protocol flows, we report separately the latency for the simplest remote accesses and the added latency for protocol flows with higher hop counts.

Using simple protocol flow for remote accesses, we define the base *hop-sequence* for read accesses and the base *hop-sequence* for write accesses. The cost overhead (in table access, in message link latency and message queuing) associated with complicated protocol flows can be estimated by comparison with the cost of this base *hop-sequence*. Our base *hop-sequences* are shown in Table 5.15. The base *hop-sequence* for read accesses is the same as the example *hop-sequence* of type 1 for reads in the one-level scheme (example in Table 5.8). The base *hop-*

sequence for write accesses is a *hop-sequence* of type 1 in the hierarchical scheme. In the protocol flow of the base *hop-sequence*, the default owner is also the page owner and the block owner. The default owner services the write request after invalidating all copies of the block. We use the same base *hop-sequences* in our analysis of protocol flows in both the one-level and hierarchical schemes to provide a common basis to compare the protocol flows in the two schemes.

Hop-sequence used as Base type in Access Latency Breakdown Analysis					
Type of Access	Nodes (Sequence)	Messages (Sequence)	Tables Accessed at Node	Table Access Latency (cycles)	Comments
Read	Requestor → Default Owner → Requestor	RREQ RDCPY	PPT PPT+SPT Cache	3 7 1	Same as One-level Type 1 hop-sequence 11 cycles table access; 1 control message, 1 block data message
Write	Requestor → Default Owner → Sharing Node → Default Owner → Requestor	WREQ INV ACK WDATA	PPT PPT+SPT+CoD PPT+SPT PPT+SPT Cache	3 14 7 7 1	Modified Two-level Type 1 hop-sequence 32 cycles table access; 3+2*(num of copies - 1) control messages, 1 block data message

Table 5.15: Hop-sequence used as base type in access latency breakdown analysis.

The base *hop-sequence* for read accesses incurs 11 cycles of table access latency and uses one control message and one data message. The base *hop-sequence* for write accesses incurs 32 cycles of table access latency. It uses one control message for the initial request and two control messages for each shared copy (invalidate and an acknowledgement) that is present during the write. One data message is used to return the data to the requester.

The following access scenarios illustrate the additional cost incurred by

protocol flows with higher *hop-sequence* type numbers than the base *hop-sequences*. Consider the *hop-sequence* type 4 read access protocol on the hierarchical scheme shown in Table 5.9 (Section 5.1.6.2). The cost for this protocol flow is summarized in Table 5.13: 35 cycles of table access, 4 control messages and 1 data message. This protocol flow has an additional cost of 24 cycles of table access and 3 control messages over the cost of the base *hop-sequence* for read accesses. Similarly, consider the *hop-sequence* type 2 write access protocol flow on the one-level scheme illustrated in Table 5.10. This protocol flow has an additional cost of 54 cycles of table access, one control message and one data message over the cost of the base *hop-sequence* for write accesses.

Note that the protocol flow with *hop-sequence* type 0 for write accesses (Table 5.14) incurs lower table access latency and requires fewer messages than the base *hop-sequence* for write accesses. We account for the negative added overhead of this write protocol flow relative to the base write protocol flow by reducing the additional overhead due to protocol flows more complicated than the base protocol flow. The total negative overhead due to write protocol flows with *hop-sequence* type 0 is calculated from the number of occurrences of such flows during execution. The net overhead due to write protocol flows different than the base *hop-sequence* type is calculated by subtracting this total negative overhead from the additional overhead due to write protocol flows more complicated than the base protocol flow.

We have analyzed the breakdown of remote read and write access latency in all three applications for both the one-level and the hierarchical schemes. We

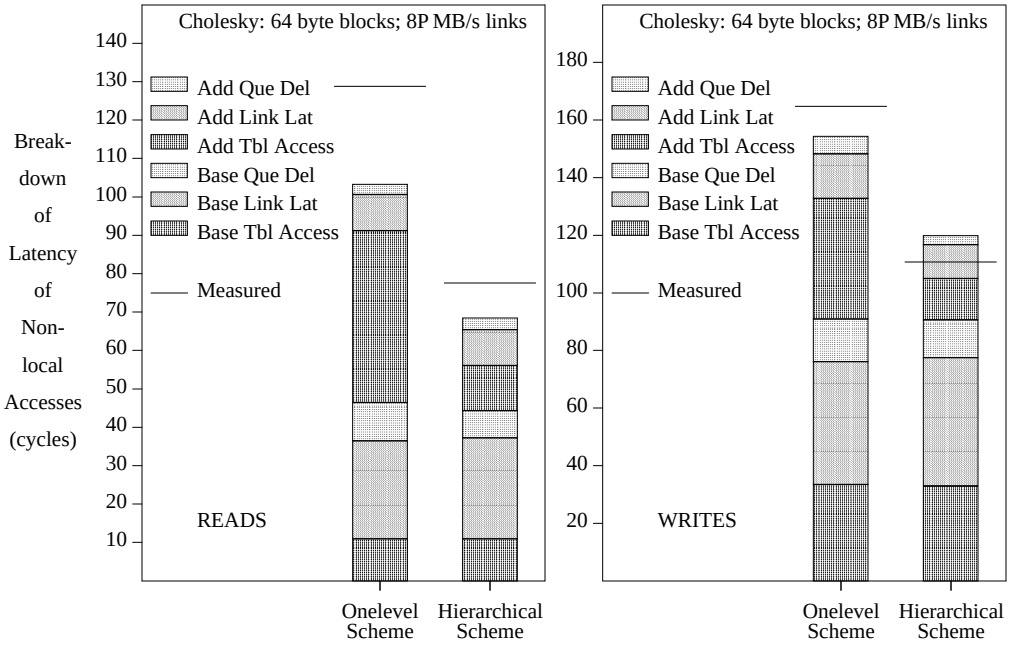


Figure 5.19: Access Latency Breakdown (Cholesky).

present and discuss the breakdown for simulation runs with a block size of 64 bytes. This block size represents a reasonable design point where our method of analysis is most meaningful. Our analysis is based on arithmetic means of the various latency components. As such, its accuracy is likely to be poor when the system behavior is bursty. With blocks significantly larger than 64 bytes, bursty system behavior is expected due to network traffic and hot spot contention for shared resources. Specifically, large blocks require large messages for moving blocks between nodes. Our simulations assume that messages cannot be partitioned into multiple packets. Hence, large blocks lead to large packets being transmitted across the network. Large packets occupy network links for long periods of time, blocking other packets which need the link. For small messages (e.g. control messages), the total latency through the network may be increased

dramatically by the time spent waiting in queues for large packets to clear required links. As discussed earlier, larger blocks lead to higher false sharing. Thus, it is more likely that different processors contend for access to the same block (to access different shared variables) when large blocks are used. At various intervals during execution the owners of such blocks must service a large number of requests and can become hot spots [Pfis85] in the system. During those intervals, the latency of messages to the hot spots can be much higher than the average message latency. Furthermore, many nodes in the system may be blocked due to the limited rate at which requests are handled by the shared block owners.

Figure 5.19 shows the breakdown of remote read and write access latency in the Cholesky application. The breakdown is shown for both the one-level and hierarchical schemes (block size of 64 bytes). The height of the segments of the bar corresponding to the base table access, link latency and queuing delay (labeled ‘‘Base ..’’) show the latencies if all remote accesses in the execution were satisfied by protocol flows of the base *hop-sequence* type. The height of the other segments of the bar (labeled ‘‘Add ..’’) show the additional latencies due to protocol flows with *hop-sequence* different than the base. This is easily computed since the difference in cost of each *hop-sequence* from the cost of the base *hop-sequence* is known. The overall height of the bar shows the overall average remote access latency computed using our analysis. The horizontal line marks the average remote access latency that was directly measured in the simulation.

The discrepancy between the computed and measured average access latency is expected due to our indirect analysis. We have used the average message link latency and queuing delays measured by the simulator. Any burstiness in the

network traffic during the simulation is not captured by these numbers. In addition, we have based our analysis on specific *hop-sequences*. As we have mentioned earlier, other *hop-sequences* as well as different table lookups for the *hop-sequences* are possible. The difference between the computed average access latency and the measured average access latency can be attributed to the differences in the protocol flows and to the dynamic variations in the network behavior.

Figure 5.19 shows that table access latency and the message link latency are the main contributors to the access latency (with 64 byte blocks). Although the hierarchical scheme uses protocol flows with higher *hop-sequence* type numbers (Section 5.1.6.3), the one-level scheme incurs a much higher cost in table access than the hierarchical scheme. This is primarily due to frequent access to the Migrated Block Table (MBT) in the one-level scheme. In the one-level scheme, the MBT must be updated on every write access and must be looked up at the default owner on many read accesses. The effect of MPT access on the latency of remote accesses in the one-level scheme may be largely artificial since the latency of accessing the MBT in our simulations (40 cycles) may be excessive. A realistic implementation of the one-level scheme must minimize access to the MBT as well as speed up accesses to the MBT. To perform a fair comparison, the remote access latencies on the one-level and the hierarchical schemes must be compared with an optimized MBT.

We have run additional simulations to study the one-level and hierarchical schemes when the latency of accessing the Migrated Block (Page) Table is only 4 cycles. Figure 5.20 shows the breakdown of remote access latency in the Cholesky

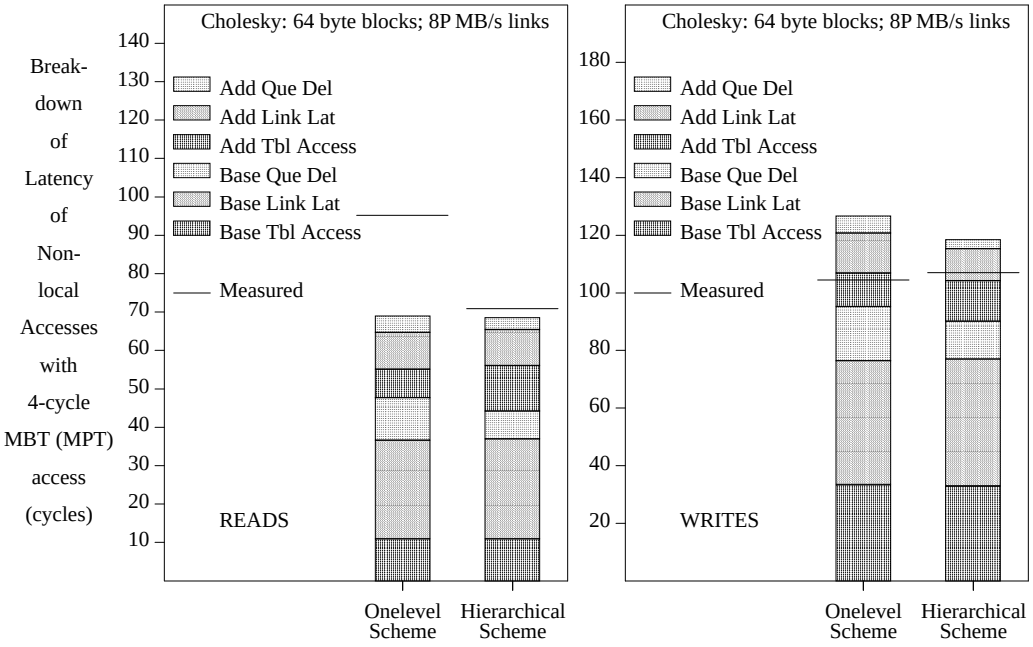


Figure 5.20: Access Latency Breakdown (Cholesky) with MBT access latency of 4 cycles.

application kernel with the optimized MBT (MPT) access for a block size of 64 bytes. As expected, the latency incurred in mapping table lookup in the one-level scheme is smaller when the MBT access latency is only 4 cycles. By comparison with the access latency breakdown shown in Figure 5.19 for an MBT access latency of 40 cycles, it can be seen that the measured and the computed overall latency of remote accesses decrease when the MBT access latency is reduced, both with the one-level scheme and with the hierarchical scheme. The benefit from an optimized MBT is more significant for the one-level scheme because the MBT is more frequently accessed in the one-level scheme. The figures also show that write accesses appear to benefit more from the optimized MBT than read accesses (the measured latency of remote write accesses drops by about 36% while the measured

latency of remote read accesses drops by about 27%). This is expected since write accesses must always update the information in the MBT while read accesses may not always need to access the MBT.

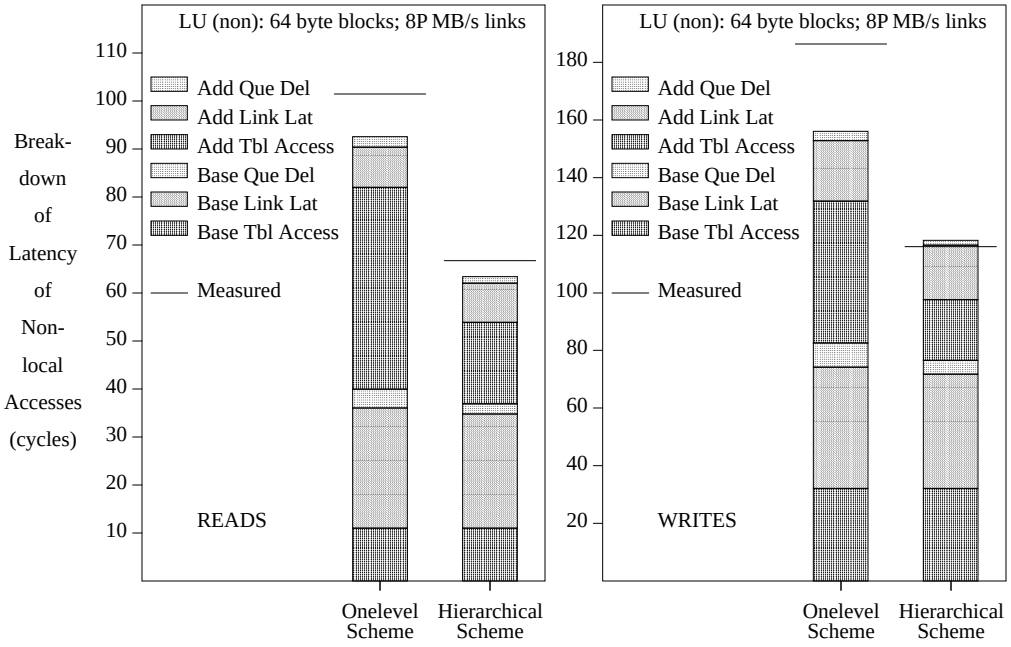


Figure 5.21: Access Latency Breakdown (LU).

Figure 5.21 shows the breakdown of the latency of read and write accesses in LU with the unoptimized MBT (MPT) access (40 cycles). The breakdown of remote access latency in LU has the same character as the breakdown of remote access latency in Cholesky. The latency incurred in accessing mapping tables and the message link latency constitute the major fraction of the remote access latency on both the one-level and hierarchical schemes. Also, the one-level scheme incurs significantly higher mapping table overhead due to the high cost of MBT access.

Figure 5.22 shows the breakdown of remote access latency in LU when the

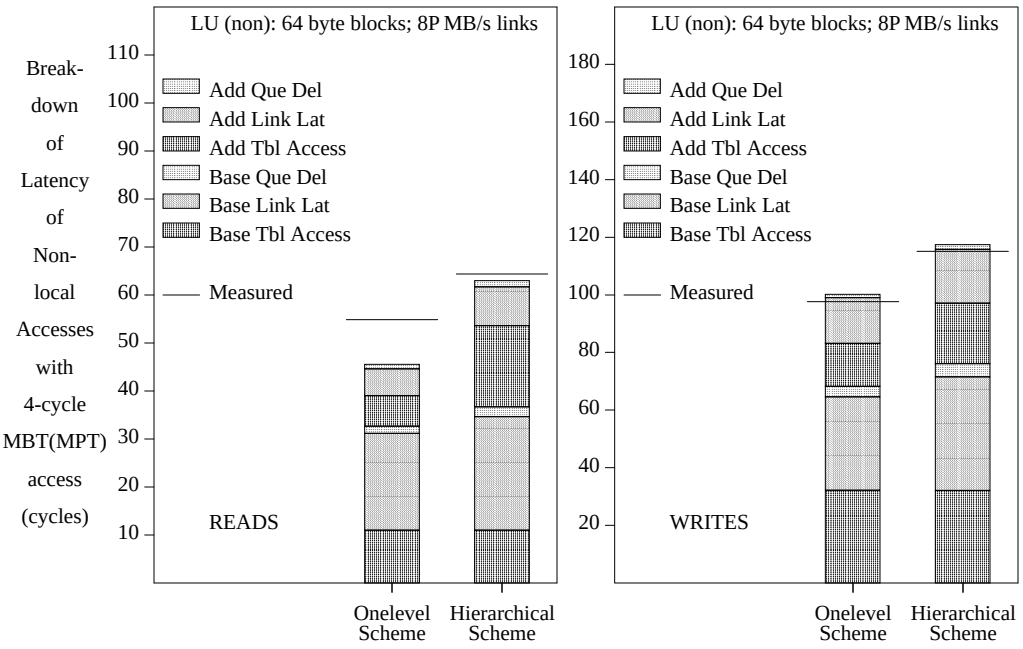


Figure 5.22: Access Latency Breakdown (LU) with MBT access latency of 4 cycles.

MBT (MPT) access has a latency of 4 cycles. Reducing the latency of MBT accesses significantly improves the performance of remote accesses on the one-level managed DSM system. The latency incurred in mapping table lookup in the one-level scheme is much smaller. As we mentioned earlier, the optimized MBT access benefits the one-level scheme much more than it benefits the hierarchical scheme. As a result, the average latency of remote accesses is lower in the one-level scheme than in the hierarchical scheme. The optimized MBT access benefits write and read accesses nearly equally (the measured latencies remote writes and remote reads drop by about 47%).

Figure 5.23 shows the breakdown of the latency of remote accesses in the Radix application when the MBT access latency is 40 cycles. In the Radix

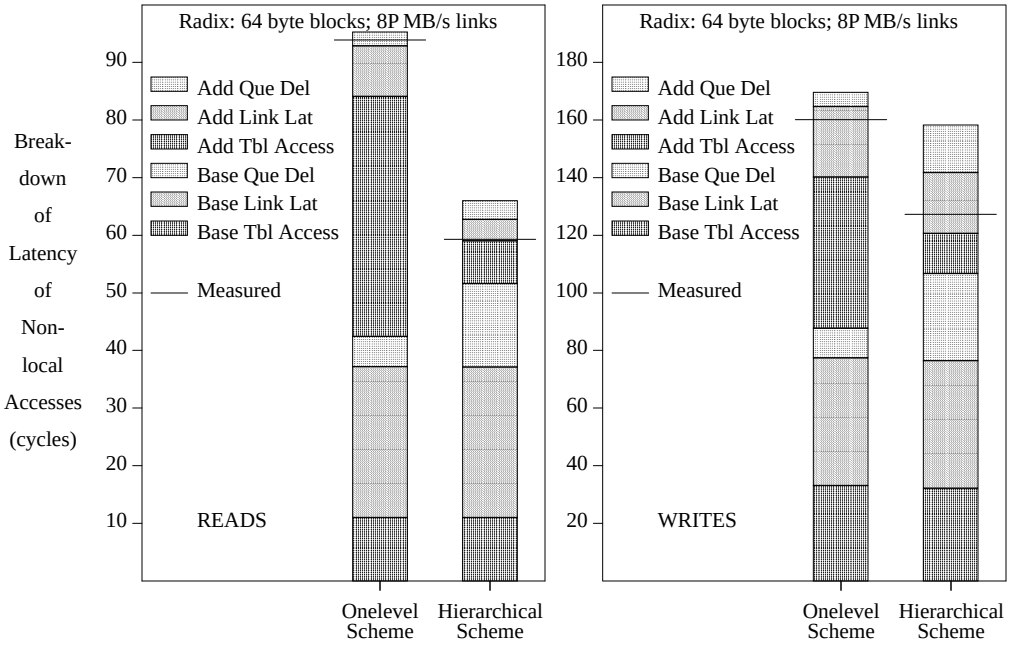


Figure 5.23: Access Latency Breakdown (Radix).

application, mapping table access latency and message link latency are the major components of remote access latency in the one-level scheme. However, the message queueing delay appears to be a significant part of the access latency in the hierarchical scheme. As we have mentioned earlier (in Section 5.1.6.1), in the Radix sorting application, processes may need to access data from disk when the keys being sorted are written to their new locations after each phase of sorting. With the hierarchical scheme, the data is accessed from disk in large pages. Since all processes concurrently attempt to update the array locations and also contend for synchronization locks, the large page-size messages can increase contention and queuing delay in the network. The latency of accessing mapping tables is higher in the one-level scheme than on the hierarchical scheme due to the long latency of MBT accesses, as was the case with the other two applications.

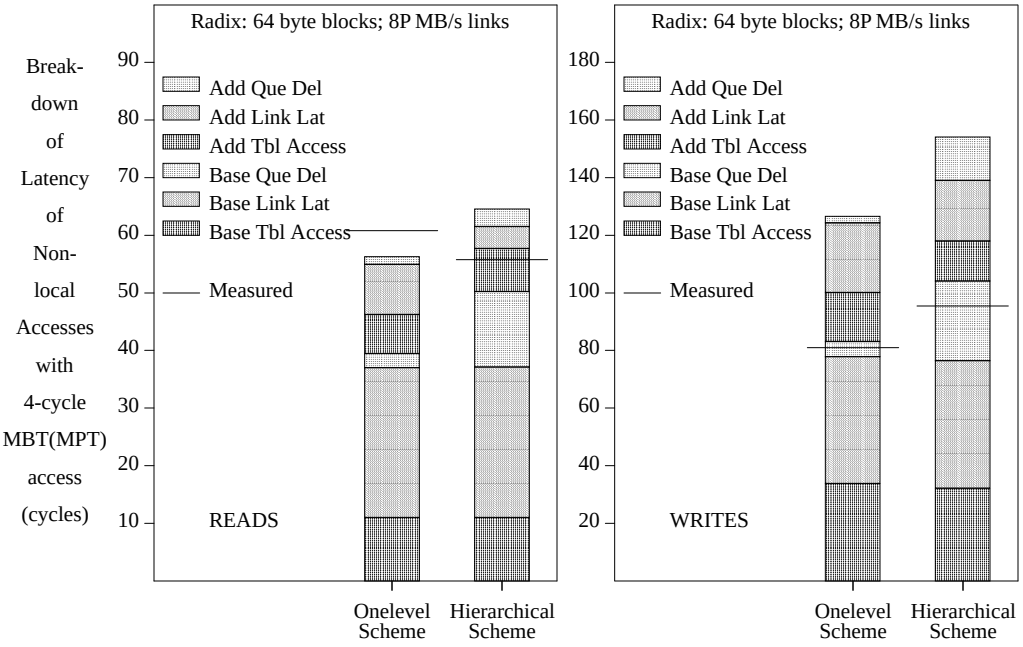


Figure 5.24: Access Latency Breakdown (Radix) with MBT access latency of 4 cycles.

When the MBT (MPT) access latency is reduced to 4 cycles, the latency incurred in mapping table accesses and the overall remote access latency (shown in Figure 5.24) improve significantly. The impact of the optimized MBT access is the same in Radix as in the other two applications: the one-level scheme benefits more than the hierarchical scheme and write accesses benefit more than read accesses. The measured latency of remote writes drops by about 50% while the measured latency of remote writes drops by about 37%.

In Figure 5.24, it can be seen that our analysis overestimated the average latency of a write access (difference between the measured and computed latencies) by over 40 cycles, for an error of over 50%. In Radix, processes concurrently attempt to update the array locations and acquire synchronization

locks after each phase of sorting. Hence, network traffic is likely to be bursty and some shared blocks (blocks with locks and blocks which map array locations modified by different processors) are likely to be hot spots. These problems are likely to be aggravated in the case of the hierarchical scheme due to page-sized transfers from disk. As we discussed earlier, such burstiness and hot spots can affect the accuracy of our breakdown analysis.

5.1.7. Profile of Execution Time

During application execution, a processor can be in one of three states. The processor can be executing instructions, can be performing an access to the shared address space, or can be idle waiting for some synchronization event. A processor's access to the shared address space can be satisfied in several ways: locally (hits), by accessing the data from disk, or by obtaining the data from other nodes in the system. In order to understand the behavior of the system and the applications, it is useful to measure the fraction of application execution time processors spend in each of the three states described above. It is also important to identify the contributions to the average shared access time from hits, disk accesses and remote accesses.

In the following discussion, we analyze application execution by characterizing the behavior of an ''average'' processor, by assuming that all processors behave uniformly. However, it must be noted that, as we discussed previously in Section 5.1.2, the behavior of processors is not uniform in all applications (Figures 5.4 through 5.6). With the Cholesky application, all processors in the system execute nearly the same number of instructions and total

accesses to shared data. The distribution of instructions and shared read accesses amongst the processors is less uniform in LU. In the case of Radix, instructions and shared accesses are unevenly distributed amongst the processors due to the uneven distribution of workload in the application.

The simulator measures the fraction of cycles that each processor spends executing instructions and the fraction of cycles during which each processor is idle. The behavior of the ‘‘average’’ processor is characterized by computing the arithmetic mean of these quantities over all the processors in the system (assuming the behavior of processors is uniform). The total number of cycles that the ‘‘average’’ processor spends executing instructions is calculated as the product of the fraction of cycles the ‘‘average’’ processor spends executing instructions and the measured application execution time (in cycles). The total number of cycles during which the ‘‘average’’ processor is idle is calculated similarly.

The time spent by the processor accessing shared space is characterized using several other measures taken during simulation: the total number of shared address space accesses that were hits, the average latency of such hits, and the total number of accesses to disk. The total number of cycles spent by all processors in shared access hits is the product of the total number of shared access hits and the measured average latency of a shared access hit. The total number of cycles spent by all processors in disk accesses is computed as the product of the total number of disk accesses and the disk access latency (1000 cycles). The total number of cycles spent by the ‘‘average’’ processor in shared access hits is calculated (assuming all processors behave uniformly) as the ratio of the total number of cycles spent by all processors in shared access hits to the total number of processors in the system

(64). Similarly, the number of cycles spent by the ‘‘average’’ processor in disk accesses is calculated (assuming all processors behave uniformly) as the ratio of the total number of cycles spent by all processors in disk accesses to the total number of processors in the system (64).

We use the measurements taken to profile the latency of non-local accesses (Section 5.1.6.4) to estimate the number of cycles spent performing non-local shared address space accesses. The total number of cycles spent in non-local shared accesses is calculated from the occurrence counts of individual *hop-sequence* types and their individual component costs (calculated using the network performance measures as discussed in Section 5.1.6.4). We again assume that the total number of cycles spent in non-local shared address space accesses is uniformly distributed amongst all processors.

In Figure 5.25, we show the breakdown of execution cycles in the application kernel LU. The figure shows the profile of execution time for two different block sizes (64 bytes and 256 bytes) for both the one-level and hierarchical schemes. The time spent in non-local shared accesses is broken down into its table access, message link latency and message queuing components along the lines discussed earlier in Section 5.1.6.4. The horizontal line marks the actual number of application execution cycles measured by the simulation. The measured and computed execution times differ due to the indirect computation of the time spent in non-local shared accesses (see Section 5.1.6.4).

With both the one-level and hierarchical schemes, at the higher block size (256 bytes), more time is spent in sending messages through the network (message link latency and message queuing delay components are higher). With larger

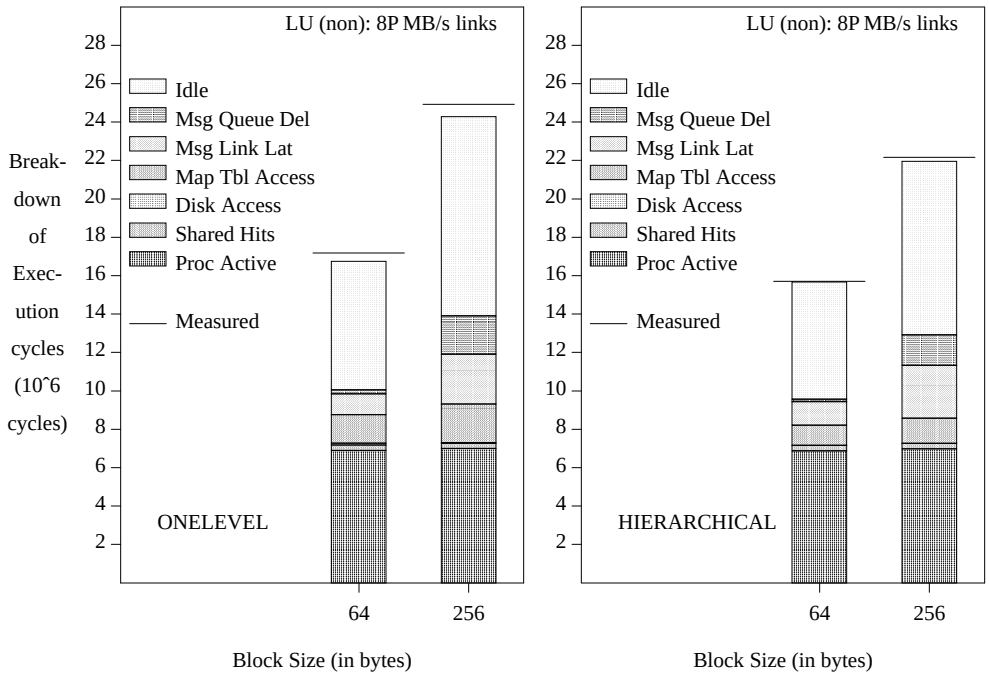


Figure 5.25: Execution cycles breakdown (LU).

blocks, block data messages are larger. For a given link bandwidth, larger block data messages incur higher link latency. Hence, the average link latency of messages is higher at larger block sizes. Larger block data messages also occupy network links for longer duration which increases queuing delays in the network. Due to these effects, the total time spent in remote accesses (sum of the ‘‘Map Tbl Access’’, ‘‘Msg Link Lat’’ and ‘‘Msg Queue Del’’ components) is higher at larger block sizes.

Larger blocks can also increase contention for shared data due to higher false sharing. Since the contention for network bandwidth and shared data is higher with larger blocks, processes that acquire these resources first can complete their task much earlier than processes that acquire these resources later. As a result, with

larger blocks, processes that complete their task early may need to remain idle for longer length of time waiting for the later processes to complete their task (at a barrier synchronization, for example). This can increase the processor idle time when blocks are larger (Figure 5.25). The increase in idle time and the time spent in remote accesses lead to higher overall application execution time and lower processor utilization at larger block sizes.

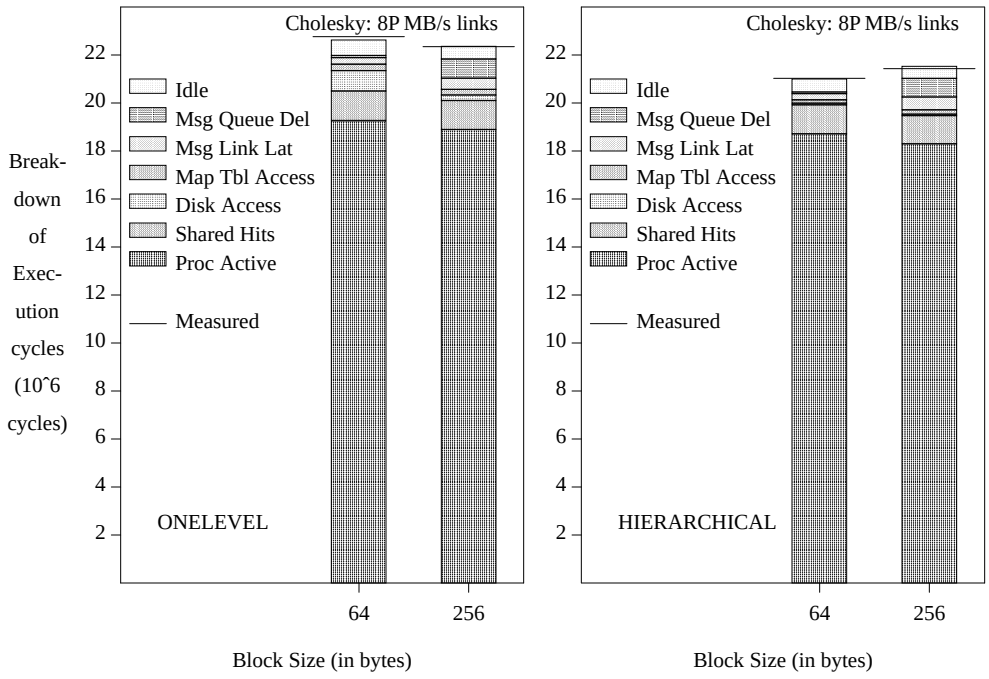


Figure 5.26: Execution cycles breakdown (Cholesky).

Figure 5.26 shows the breakdown of execution cycles in the Cholesky application. As discussed earlier, the message link latency and message queueing delay components are higher at the larger block size (256 bytes). However, the figure shows that the overall execution time of the Cholesky application was higher at the smaller block size (64 bytes) in the one-level scheme. It can be noted from

the figure that this is because more time is spent in disk accesses (the segment of the bar labeled ‘‘Disk Access’’) at the small block size. As we have discussed earlier (Section 5.1), this is an artifact in our implementation where disk accesses in the one-level scheme are at the granularity of blocks. With smaller blocks, more disk accesses are needed to fetch the data from disk increasing the overall time spent in disk accesses.

It is interesting to compare the behavior of Cholesky (Figure 5.26) with the behavior of LU (Figure 5.25). The time spent in shared hits is significantly higher in Cholesky than in LU. This matches our earlier results on the miss rate of shared accesses (see Figure 5.14 in Section 5.1.6.1) where Cholesky demonstrated a higher hit rate than LU. Since the Cholesky application has a higher hit rate for shared accesses than LU, the time spent in remote accesses is lower. As a result, processor utilization is higher in Cholesky.

The breakdown of execution cycles in Radix is shown in Figure 5.27. Radix demonstrates behavior similar to that shown by LU and Cholesky. The message link latency and message queuing delay are higher at the larger block size (256 bytes) increasing the time spent in remote accesses. As we discussed with Cholesky, more time is spent in accessing disks with 64-byte blocks than with 256-byte blocks in the one-level scheme due to the fact that we implement disk accesses at the granularity of blocks in the one-level scheme. With a block size of 64 bytes, the message link latency and message queuing delay components of execution time are higher in the hierarchical scheme than in the one-level scheme. As we explained earlier in Section 5.1.6.4, this is most likely due to the large page-sized transfers from disk in the hierarchical scheme.

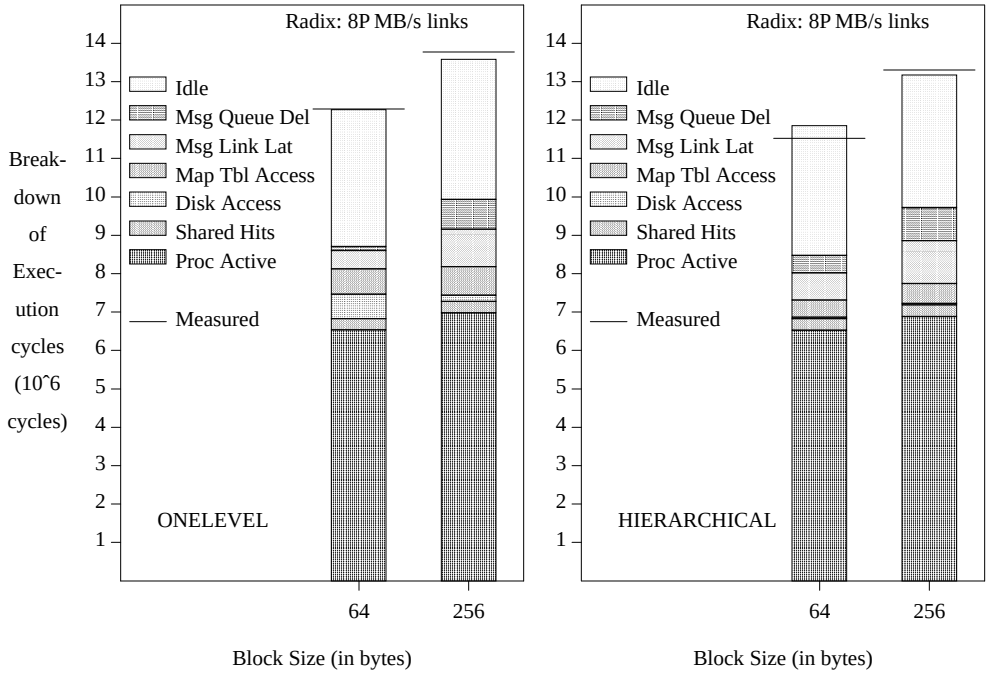


Figure 5.27: Execution cycles breakdown (Radix).

5.2. Evaluation based on Trace-driven Simulation

Our initial evaluations of the hierarchical scheme were based on our trace-driven simulator. As we described in Section 4.4.1, our simple trace-driven simulator does not model the latencies associated with the network or table lookups. Hence, all coherence transactions are completed atomically.

The address traces used to drive our trace-driven simulations were obtained from the simulation of another 64 node multiprocessor architecture on a single processor system. The address traces represent possible executions of three parallel applications: Weather, Speech, and FFT. The Weather application partitions the atmosphere into a three dimensional grid and uses finite difference methods to solve a set of partial differential equations describing the state of the

system. The Speech application comprises the lexical decoding stage of a phonetically-based spoken language understanding system. The application uses a modified Viterbi search algorithm to find the best match between paths through a directed graph representing a dictionary and another directed graph representing the input. FFT is a radix-2 fast Fourier transform.

The basic characteristics of the traces are summarized in Table 5.16. The number of instruction references, data references and shared data references in the trace are listed in the table. We performed a static analysis of the trace (without executing the simulator) to identify the size of the shared address space. The total number of 1 KB pages that were shared in the trace was measured and is reported in Table 5.16. More details about the applications and the traces can be found in[Chai90].

Application	Trace length (10^6 refs)	Number of Instruction refs (10^6 refs)	Number of Data refs (10^6 refs)	Number of Shared data refs (10^6 refs)	Max data foot print (KB) per processor	Shared foot print (KB)	
						Average per processor	of system
Weather	31.76	13.64	18.12	5.02	5346	5192	5330
Speech	11.77	10.78	11.77	11.77	510	434	596
FFT	7.44	3.11	4.33	1.05	164	130	130

Table 5.16: Trace characteristics. The system shared footprint is the number of 1 KB pages accessed by more than one processor during the execution of the application. The per-processor shared footprint is the average number of pages, out of the system shared footprint, that were accessed by individual processors.

In our simulations the size of the Present Pages Table (PPT) is chosen to accommodate most of the process footprint to ensure that the number of misses due

to PPT overflows will be negligible. Specifically, the PPT size for Weather, Speech, and FFT, was 2048, 1024, and 256 pages, respectively. When there are no free entries in the PPT, LRU replacement is used to evict pages. The number of entries in the Shared Pages Table (SPT) and the number of entries in the copy directory were varied in the simulations. The replacement mechanism for the SPT is biased to evict pages that are not owned locally (to minimize traffic). Pseudo-random replacement is used to pick a victim copy directory entry when eviction is necessary.

The simulator determines the state of the PPT, SPT, and the copy directory after each memory reference and builds statistics on miss ratios, network traffic, and the number of SPT and PPT replacements. Simulations were performed for both 1 KB and 4 KB pages for a range of block sizes, SPT sizes and copy directory sizes.

The miss ratio on accesses to the data address space and the total network traffic are good indicators of the potential system performance and are used to present the results of the simulation. The network traffic is calculated assuming that each control message is 16 bytes long, while the size of data messages (Block or Page) is 16 bytes plus the data being transmitted.

Figures 5.28 and 5.29 show the data access miss ratios, for all three applications with various SPT and block sizes. The size of a page is 4 KB in Figure 5.28 and the page size is 1 KB in Figure 5.29. The corresponding network traffic is shown in Figures 5.30 and 5.31. The advantage of a small block size is demonstrated by the decrease in network traffic with decreasing block size. For Speech and FFT, decreasing block size also results in a decrease in false sharing,

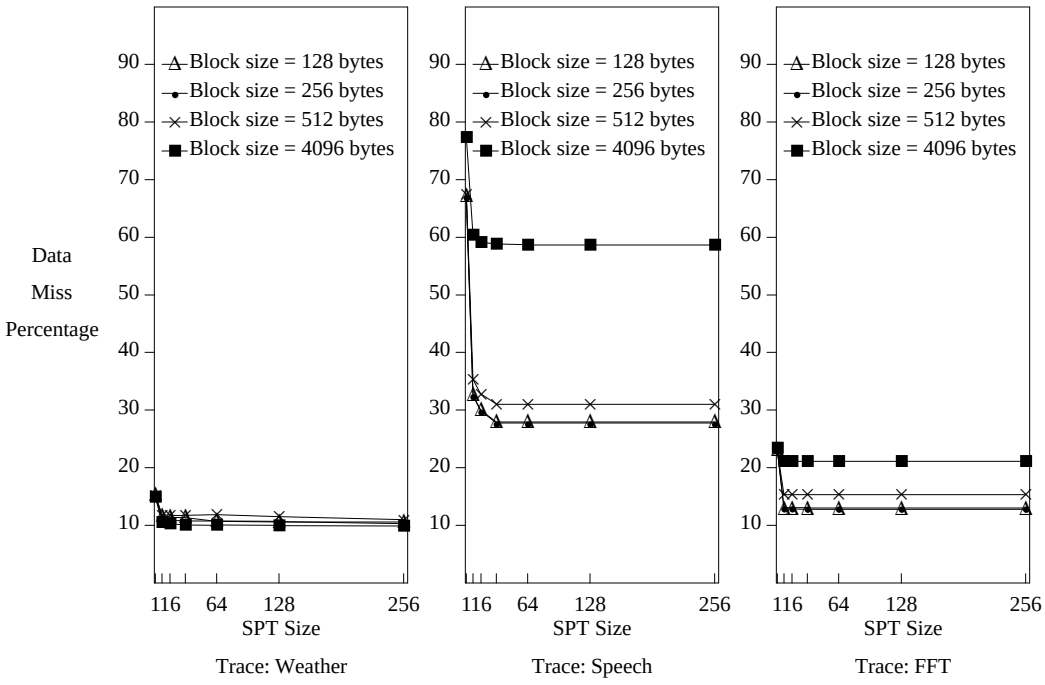


Figure 5.28: Data miss percentage versus SPT size. 4KB pages. 32 BMpP copy directory entries.

leading to a decrease in miss ratio.

An important result of the simulations is that, with the hierarchical scheme, performance is dependent on block size rather than page size. Specifically, for a page-level scheme (block size equal page size), 4 KB pages lead to significantly higher miss ratios and network traffic than 1 KB pages (Figures 5.30 and 5.31). However, for equal block size (128 bytes), performance of the system with 4 KB pages is almost identical to the system with 1 KB pages. Since larger pages result in smaller tables, this is a clear demonstration of the advantage of our multi-level coherency scheme.

For very small SPTs, significant reductions in miss ratio and network traffic

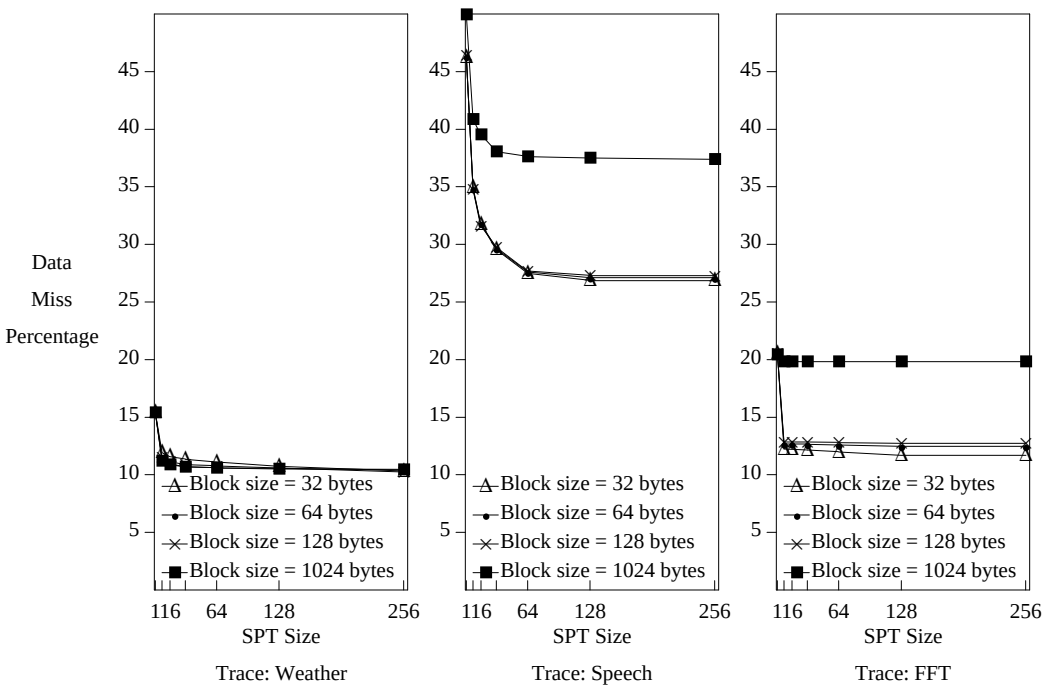


Figure 5.29: Data miss percentage versus SPT size. 1KB pages. 32 BMpP copy directory entries.

are obtained by increasing the SPT size. However, once the knee of the curves is reached (in this case, at around 32 entries), further increases in the SPT size do not provide significant performance improvement. These results confirm that a small SPT (about 10% of the total working set) would be capable of accommodating the ‘‘shared working set’’ and provide good performance. The knee occurs at a higher SPT size for 1 KB pages than for 4 KB pages. This indicates that, for these applications, in our two-level scheme, for a fixed block size, a smaller page size will actually *reduce* performance unless a larger SPT is used.

Figure 5.32 shows the miss ratio and network traffic for various sizes of the Block Map per Processor (BMpP) copy directory. As expected, performance is

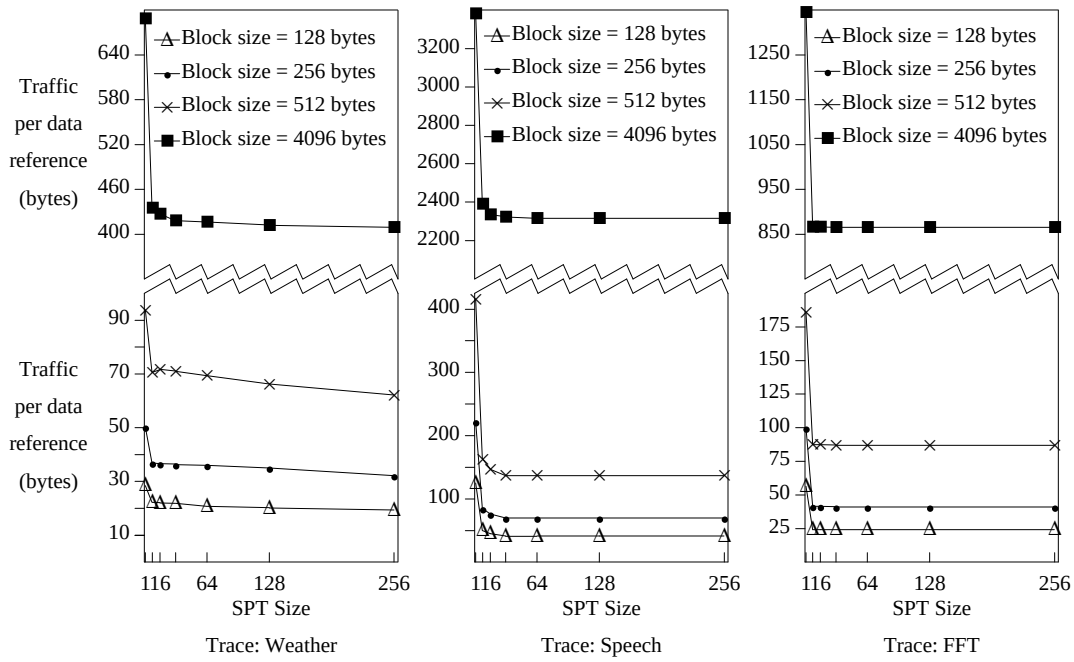


Figure 5.30: Network traffic versus SPT size. 4KB pages. 32 BMpP copy directory entries.

reduced if the directory is too small. For Weather and FFT, the reduction in miss ratio and network traffic with increasing directory size levels off for a BMpP copy directory with 32 entries or more. This agrees with the expectation, based on the access burst model (Section 3.1.2 in Chapter 3), that there is little performance advantage in maintaining full copy directories for shared blocks. The different behavior for Speech is due to the high read-only sharing of a dictionary. A policy of eviction, such as we have employed, will fair poorly in the presence of significant read sharing by a large number of processors. Better support for read-shared pages can be provided with a broadcast policy [Agar88], with a policy that allows a small, dynamically determined, set of pages to have larger copy tables [O'Kr90], or with support for maintaining full copy directories in

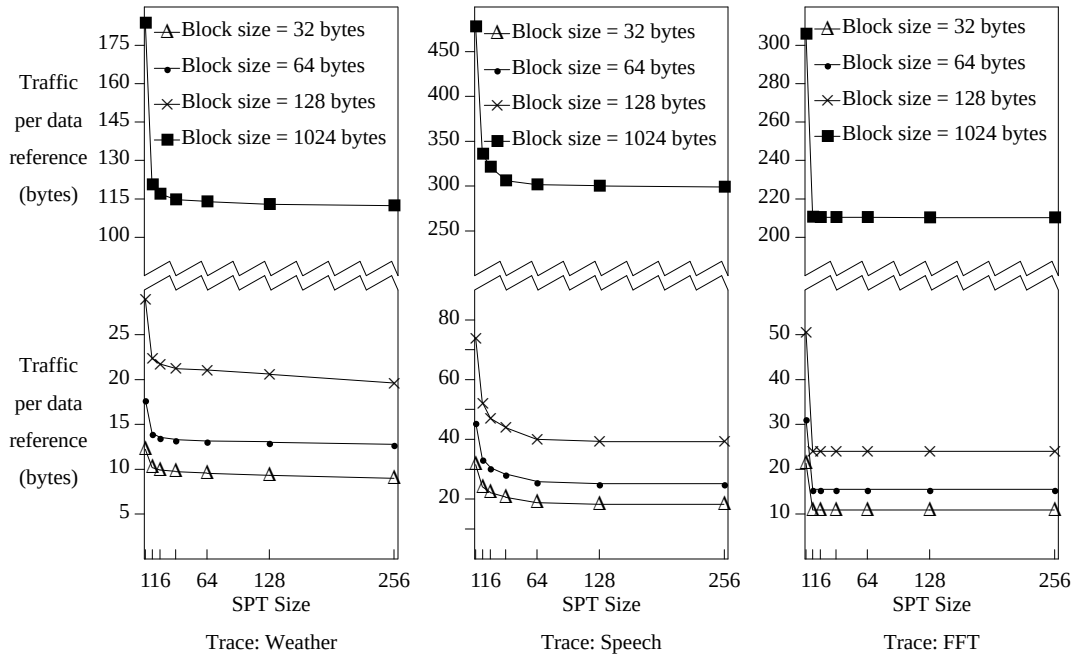


Figure 5.31: Network traffic versus SPT size. 1KB pages. 32 BMpP copy directory entries.

software [Chai91].

Figure 5.33 compares the performance of the Block Map per Processor (BMpP) and the Processor List per Block (PLpB) copy directory organizations. The graphs show the data miss percentage and the network traffic as a function of the size of the copy directory per shared page, for both organizations. The data points for the BMpP organization correspond to 8, 16, 32 and 64 entries per page. The data points for the PLpB organization correspond to 4, 8, 16 and 32 entries per block. As mentioned earlier, the storage overhead of the BMpP organization is smaller than the overhead of the PLpB organization. Assuming that table entries are byte aligned in memory, a BMpP copy directory with 32 entries requires 192 bytes per page. With the PLpB directory organization, 256 bytes per page support

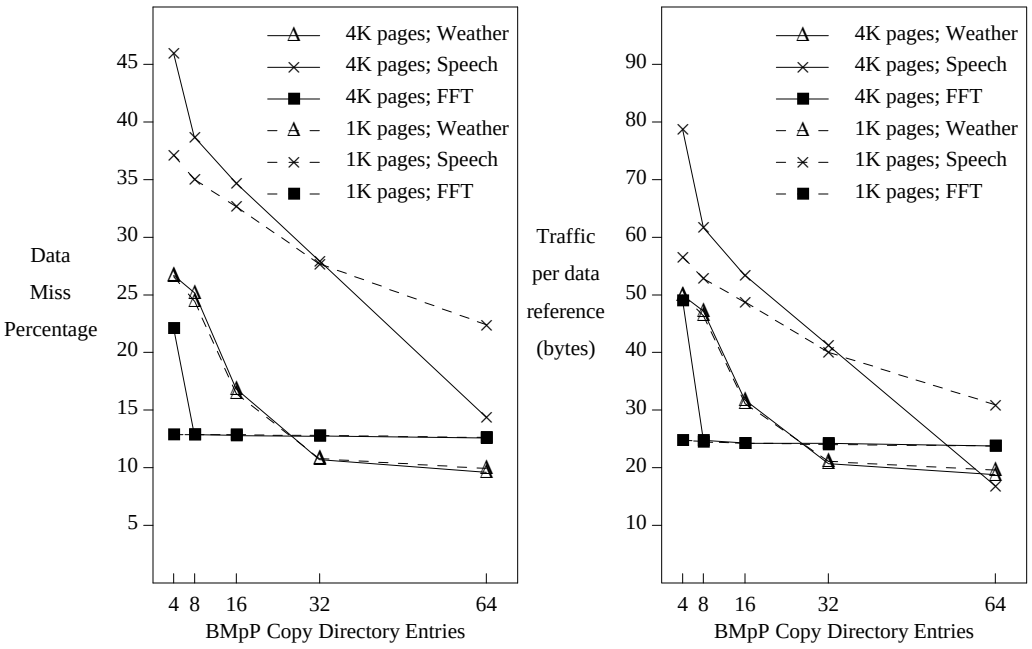


Figure 5.32: Data miss percentage and network traffic versus number of entries in BMpP copy directory for page sizes 1KB and 4KB. SPT size is 64. Block size is 128 bytes.

directories with four entries per block. In our simulations, for comparable performance, the copy directory in the PLpB organization had to be several times larger than in the BMpP organization.

Since our trace-driven simulator models accesses to the shared address space atomically and is based on address traces that provide no timing information, the results from our trace-driven simulations do not provide any estimate of application execution time. The improvement in data miss percentage and network traffic per data reference with the hierarchical scheme may not directly translate into improvement in overall application performance. However, they provide sufficient validation of the basic ideas of the hierarchical scheme to justify detailed evaluation of the hierarchical scheme. As we have described earlier (in

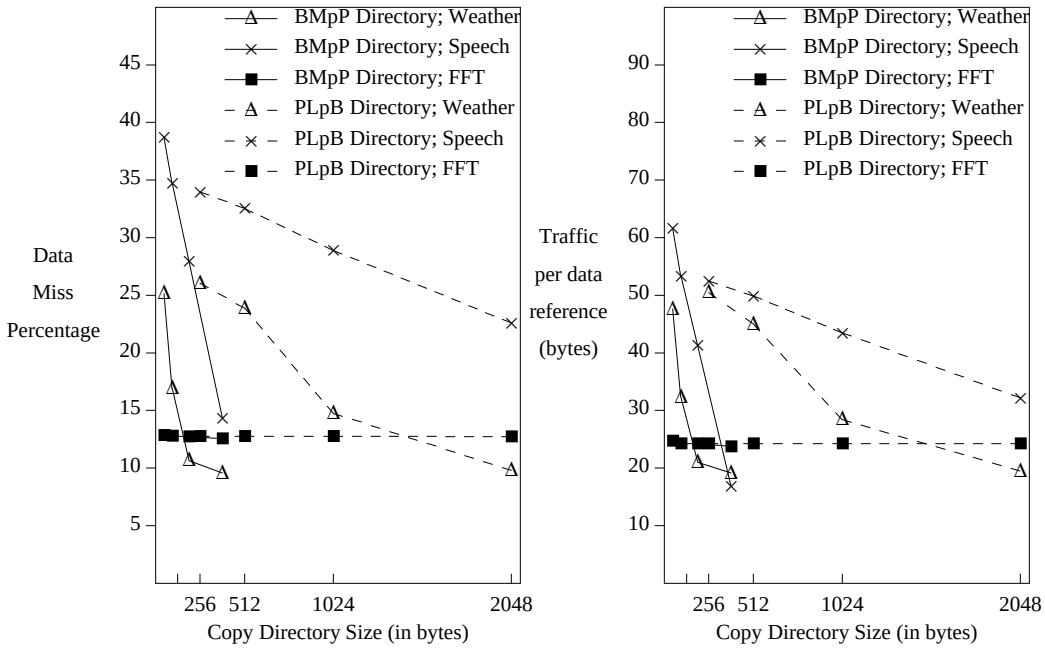


Figure 5.33: Data miss percentage and network traffic versus copy directory size per page for BMpP and PLpB organizations. Page size is 4KB. SPT size is 64. Block size is 128 bytes.

Section 5.1), we have evaluated the hierarchical scheme in more detail using execution-driven simulations.

Our measurement of the shared working set size of applications with the execution-driven simulation (Section 5.1.3) confirm the results about shared working set size from our trace-driven simulations. The results from both our evaluations (Figure 5.7 from execution-driven simulations and Figure 5.31 from trace-driven simulations) indicate that a Shared Pages Table (SPT) which has much fewer entries than the PPT can be used with negligible impact on application performance.

The evaluation based on trace-driven simulation has demonstrated that

network traffic is very high with page granularity one-level schemes and decreases with decreasing block size (Figure 5.31). Our execution-driven simulations verify that remote access latency (Figure 5.12) and application execution time (Figure 5.10) are both very high with page granularity one-level schemes. The figures also show that the decrease in network traffic with decreasing block size leads to lower latency for remote accesses and lower application execution time at smaller block sizes. At small block sizes, the overall time spent by the application in performing remote accesses is smaller (Section 5.1.7) leading to higher processor utilization.

We have studied the size of the copy directory and the effectiveness of our Block Map per Processor (BMpP) copy directory organization using both our trace-driven and execution-driven simulations. Our study based on trace-driven simulations was conducted by varying the size of the copy directory. In our study using execution-driven simulations, we have measured the average size of the copy directory used by the applications. Both studies indicate (Figure 5.9 and Figure 5.32) that copy directories with few entries are typically sufficient. However, in our study using trace-driven simulations, we have observed (Figure 5.32) that small copy directories may be inadequate when there is high read-only sharing of data. The applications we have used in our execution-driven simulations do not demonstrate significant read-only sharing. Figure 5.8 (from execution-driven simulation) and Figure 5.33 (from trace-driven simulation) both agree that the BMpP copy directory organization maintains the copy directory information more efficiently than the Processor List per Block (PLpB) organization).

Chapter Six

Error Recovery for Distributed Shared Memory Systems

The reliability requirements of large DSM multicomputers, consisting of thousands of VLSI chips, can only be met by using fault tolerance techniques [Tami84]. Application-transparent error recovery in message-passing systems can be achieved using *message logging* or based on *coordinated checkpointing*. These schemes do not require checkpointing at every interprocess communication and yet are not susceptible to the domino effect [Rand78]. *Message logging* [Stro85, John87] involves the logging of interprocess messages as well as periodic process checkpoints. The message logs are used to restore a recovering process to a state *consistent* [Rand78, Chan85] with the rest of the system. *Coordinated checkpointing* [Bari83, Tami84, Koo87, Tami89] is based on checkpointing ‘‘together’’ consistent states of all the processes that have interacted since their last checkpoint. Recovery requires rolling back the entire interacting set of processes containing the failed process. In multicomputers, message logging requires significantly higher overhead than coordinated checkpointing during normal operation due to the need to log all communication as well as to generate and process for each message information for tracking dependencies [Fraz89, Fraz95].

Recovery schemes for DSM multicomputers must address the same issues as messages-passing multicomputers. Some proposed schemes use the simple but costly approach of checkpointing prior to every ‘‘communication’’ with another

process [Wu90a, Tam90, Stum90] (see Section 6.5). Based on message logging, it has also been proposed to use independent process checkpoints together with logs of remote page accesses [Rich93]. As mentioned earlier, these schemes suffer from unacceptably high overhead during normal operation.

This chapter presents an error recovery scheme for DSM multicomputers based on coordinated checkpointing [Jana94]. Coordinated checkpointing schemes minimize the overhead during normal operation since they require little or no special recovery-related actions for interprocess communication. There is no need to log messages or process complex dependency-tracking information. Recovery may be more expensive than with message logging. However, the overall cost (overhead) of checkpointing will usually be lower. Far fewer checkpoints may be taken since the frequency of checkpointing can be tuned to the *needs* of the application instead of being determined by the frequency of interprocess communication. Thus, for communication-intensive applications, recovery schemes based on coordinated checkpointing are preferable over other schemes [Tami84, Fraz89, Fraz95].

Our error recovery scheme is an adaptation of coordinated checkpointing and rollback for message-passing systems. This adaptation efficiently solves several potential difficulties with DSM systems which are due to the lack of explicit messages, the presence of a shared replicated address space, and the distributed coherency directory. We have developed mechanisms to identify and track read-write communication in the DSM system, to terminate read and write transactions that are in progress at checkpoint or recovery, and to checkpoint and recover the shared replicated address space. Our scheme does not checkpoint and rollback the

coherency directory. Instead, at recovery, the distributed coherency directory is ‘‘repaired’’ to restore it to a correct state.

The following section is a brief summary of coordinated checkpointing for message-passing systems. Section 6.2 describes our system model and assumptions. Our checkpointing and rollback scheme is presented for the one-level managed DSM system in Section 6.3. We extend this scheme to the hierarchical DSM system in Section 6.4. Section 6.5 is an overview of prior research on DSM system recovery. Performance evaluation based on trace-driven simulations is presented in Section 5, where we compare our schemes (for the block-level DSM system and for the hierarchical DSM system) to previously proposed schemes for DSM recovery.

6.1. Coordinated Checkpointing and Rollback

In a system where processes interact with each other, the states of two processes are consistent if they agree on what messages have been exchanged between them. Coordinated checkpointing and rollback schemes require that the most recent checkpointed state of every process is consistent with the most recent checkpointed state of every other process [Bari83, Tami84, Koo87, Tami89]. Hence, the set of all the committed checkpoints in stable storage is *always* a consistent *snapshot* [Chan85] of the entire system.

One approach to coordinated checkpointing and rollback is to save and restore a consistent snapshot of the entire system state [Tami84]. This approach may be undesirable since for large systems it may take an unacceptably long time to checkpoint or roll back. Furthermore, the overhead on the system for

checkpointing will be highly ‘‘bursty’’ instead of being evenly spread over time. Finally, global checkpointing and rollback does not allow tuning the overhead and the recovery latencies to the different needs of different applications which may be executing on the system simultaneously.

In order to overcome the disadvantages of global checkpointing, it is possible to keep track of which processes have communicated with each other since their last checkpoint and coordinate the checkpointing of only the *interacting set* of processes as opposed to all processes [Bari83, Koo87, Tami89]. For each process, the system maintains the set of processes with which it has interacted directly since its last checkpoint. When a process initiates a checkpointing session, this communication information is used to build a *communication tree* consisting of all the processes with which the first process has communicated directly or indirectly since the last checkpoint [Tami89]. The initiator of the session coordinates the consistent checkpoint of all members of the communication tree [Bari83, Koo87, Tami89] using a two-phase commit protocol [Gray78]. All participants in the checkpointing session suspend their execution and all messages in transit between them are flushed to their destinations and become part of the destinations’ state. Recovery is similar to the checkpoint session, except that the flushed messages are discarded and the process states are restored from stable storage.

6.2. System Model and Assumptions

We develop the error recovery scheme for a DSM multicomputer whose design was introduced in Chapter 3. We summarize the salient features of this system below. The DSM multicomputer system consists of several nodes interconnected by point-to-point links (Figure 1.1 in Chapter 1). Each node in the system includes (Figure 3.1 in Chapter 3) an application processor, local memory, cache memory, and coprocessors to assist in the coherency management and interprocessor communication [Kusk94]. Each node may be time-shared between multiple processes. A set of processes that share their address space form a *task*. The system can execute multiple tasks simultaneously. Unique task identifiers partition the total virtual system space into disjoint task virtual spaces. Processes belonging to different tasks execute in disjoint address spaces and cannot share data.

DSM is implemented on the system using a distributed directory-based coherence protocol [Chai90, Leno90]. Ownership [Katz85] is managed at a granularity of *blocks* (tens of bytes). Multiple read-only copies can be distributed through the system. A node must obtain exclusive ownership of a block prior to modifying it, by invalidating all other copies. To identify the current owner of any block, the scheme uses the fixed distributed manager algorithm [Li89b]. Each node is statically assigned to be the default owner of a set of blocks using a hash function. The default owner keeps track of who is the current owner of the block.

Each node maintains part of the *distributed directory* which indicates, for each block present locally, the access permission and the identity of the block owner. The block owner maintains, in addition, the location of read-only copies.

Each node also maintains a Migrated Block Table (MBT), in which it identifies the current owner of all *its* default owned blocks that have been migrated away from the node. A node that does not know the identity of the block owner can request access through the default owner of the block. (The meaning of ‘‘ownership’’ and the structure of the directory tables will be extended for the hierarchical DSM system in Section 6.4.)

We assume that the underlying network guarantees reliable FIFO delivery of messages between any pair of nodes. Each node on the system is assumed to be self-checking and fail-stop [Tami83, Schn84]. A subset of the nodes in the system are connected to disks, and checkpoints are saved onto these disks. These disks are assumed to be stable storage; a disk failure will lead to a crash of the system. Upon node failure, all the contents of the node memory and the node directory are considered lost. The neighbors of a failed node detect the failure and broadcast the identity of the failed node to all the other nodes in the system.

6.3. Coordinated Checkpointing and Rollback for DSM Systems

In DSM multicomputers, memory accesses that result in a miss on the local node can require multiple message exchanges between the nodes to transmit the data and update the directories [Chai90, Leno90, Li89a]. It is possible to implement coordinated checkpointing and rollback on DSM multicomputers by treating each one of these messages as an ‘‘interaction’’ between processes. With this approach, since the directories modified by these ‘‘messages’’ are accessed by all tasks, processes of different tasks (which do not communicate) become part of the same *interacting set* and must be checkpointed and rolled back together.

Hence, this naive approach to implementing coordinated checkpointing would result in all the processes in the system and all the cache directories quickly becoming part of one interacting set. Thus, virtually all checkpoints and rollbacks would have to be of the entire system. In our coordinated checkpointing and rollback scheme for DSM multicomputers, we take into account the fact that not all messages exchanged on behalf of a process need to be considered in determining the interacting set of processes that are checkpointed and rolled back together.

In order to adapt coordinated checkpointing and rollback to DSM multicomputers, we must examine the unique characteristics of DSM systems: 1) processes communicate by reading from and writing to a shared address space, 2) a single communication *event* at the application level (load or store) may trigger a complex *transaction*, involving multiple internode messages, 3) the address space of each process is distributed among multiple nodes, 4) multiple copies of memory blocks are allowed, and 5) there is a distributed directory which is common to all tasks but is not part of the state of any process.

6.3.1. Tracking Interprocess Communication

A key requirement for coordinated checkpointing is the ability to identify *interacting sets* of processes. For this purpose, an ‘‘interaction’’ is any operation that may cause the state of one process to depend on the state of another process. In message passing systems any transmission of information from one process to another, i.e., any message exchange between processes *may* result in such dependency. Hence, in message passing systems a message exchange causes the participating processes to be included in the same interacting set.

In a DSM system, processes communicate by reading from and writing to a shared address space. However, not all accesses to the shared address space imply dependency. Even accesses that result in transmission of information from one processor to another do not necessarily imply dependency between the processes running on the different processors. Assuming that all accesses to the shared address space imply dependencies would unnecessarily enlarge the interacting set. For example, consider a *load* that is satisfied by accessing a block on a remote node. If the block has not been modified since the last checkpoints of all the processes in the system, the load does not cause a dependency between the state of the process executing the load and the state of any other process in the system. Hence, this remote access should be ignored when determining the interacting set of processes that must be checkpointed together. An efficient checkpointing scheme must identify which accesses to the shared address space cause dependencies that must be taken into account in determining the interacting set.

Processes of a task that run on the same node communicate by direct access to locations in local memory. These intra-node interactions can be identified with some architectural support in the local memory access mechanism [Wu90a], but the cost of providing this support is likely to be too high for the benefits gained. Hence, we view all processes of a task that run on the same node as having interdependent states. We label the set of processes that belong to the same task and run on the same node a *process-cluster*. Processes of a process-cluster are always checkpointed and rolled back together.

Processes of a task that run on different nodes communicate by read-write sharing of memory blocks across the network under the control of the coherency

protocol. To identify these interactions, we extend the functionality of the coherency protocol. With each block in local memory there is an associated *dirty-since-checkpoint* (*dsc*) bit in the local directory. The dsc bit indicates whether the block has been modified since the last time it was checkpointed. Every block transfer is tagged with the block's dsc bit.

The transfer of a block whose dsc bit is set implies that the state of the process requesting the block may become dependent on the non-checkpointed state of the process that modified the block. Hence, the transfer of a block with the dsc bit set implies a possible dependency and is used for determining the interacting sets of processes.

Using the dsc bit mechanism, each node maintains for each process-cluster on the node a list of all process-clusters with which there has been *direct* communication since the last checkpoint (similar information is maintained for each process in message-passing systems[Bari83, Koo87, Tami89]). The dependency between process clusters on two nodes, *P* and *Q*, is established when a block of a the task to which both process clusters belong is transmitted from node *Q* to node *P*, with the block's dsc bit set. This dependency is recorded independently in both nodes.

6.3.2. Transactions in Progress when Checkpointing or Rollback are Initiated

In order to save a consistent snapshot of an interacting set of processes in a message-passing system, messages in transit between the processes when checkpointing is initiated must also be saved. This can be done by ‘‘flushing’’ messages to their destination and saving them with the destination process

checkpoint [Tami84, Tami89]. Messages in transit between the processes when rollback is initiated are discarded [Tami84, Tami89]. In a DSM multicomputer, process interaction takes place through read and write ‘‘transactions’’ which are not necessarily atomic. Each transaction involves the exchange of several network messages. When checkpointing or rollback are initiated, transactions may be *pending*, with the processor waiting for a response from one of the other processors participating in the transaction.

The handling of transactions which are pending when checkpointing or rollback are initiated is a critical issue. There are four alternatives to be considered: 1) abort the transaction, 2) suspend the transaction, 3) continue the transactions while the checkpointing or recovery session proceeds, or 4) attempt to complete the transaction before proceeding with the checkpointing or recovery session. This is more complicated than flushing messages since it involves multiple nodes and interactions with the coherency management protocols.

When rollback is initiated, pending transactions associated with processes being rolled back are orphans [Lisk87] and should be aborted. However, always aborting pending transactions when checkpointing is initiated results in unnecessary performance loss. In many cases it may be possible to complete transactions which are pending when checkpointing is initiated, thus avoiding the performance cost of redoing them later.

It may be possible to always suspend pending transactions when checkpointing is initiated. However, this would add significantly to the complexity of checkpointing since it would require saving the state of partially executed transactions. Depending on how this is done, some of this ‘‘transaction state’’ may

involve nodes that would otherwise not participate in the checkpointing session.

For performance reasons, it might be desirable to allow pending transactions to proceed in parallel with a checkpointing session (so that the transactions and checkpointing are not delayed). This results in a situation similar to the handling of dynamic interacting sets in a message-passing system with asynchronous coordinated checkpointing [Tami91]. Specifically, the size of the interacting set may increase while the checkpointing session is already in progress. Whether or not a block transmitted as part of a pending transaction should be part of the checkpoint depends on the status of all the participants in the transaction. The complexity of special handling needed to manage this situation presents sufficient reason to look at alternative approaches.

The approach we adopt is to wait for the completion (successfully or unsuccessfully) of pending transactions before proceeding with checkpointing or recovery. It is necessary to ensure that checkpointing or recovery sessions do not wait endlessly for the transactions to complete. Hence, the coherency protocols must be extended to provide a negative acknowledgment terminating (unsuccessfully) the transaction when service cannot be provided.

When a failure occurs, the system must ensure that nodes do not end up waiting indefinitely for the completion of transactions that cannot be completed due to the failure. Every participant in a transaction must be capable of identifying the situation when the transaction cannot progress due to the failure of another participant. In this case, the node identifying the situation must terminate the transaction.

In order to facilitate the detection of transactions that must be aborted, a node

failure is broadcast to all other nodes (Section 6.2). However, with many coherency protocols [Li89b, Leno90] the node initiating the transaction is not always aware of the identity of the node servicing the transaction. With these protocols, there is no simple way for the initiator to determine that the transaction cannot progress due to the failure of the node servicing the transaction. For example, a node requesting a read copy of a block may send the request to the block's default owner which, in turn, forwards the request to the current block owner. In this situation, the initiator of the transaction does not know the identity of the block owner, and, hence, cannot associate the failure of the block owner with the failure of the transaction. At the same time, the default owner is not expecting a reply from the current owner (the reply is sent directly to the requester) and thus will not react to the failure of the block owner. The coherency protocols can be modified to solve this problem. For example, the response from the block owner could be always propagated through the default owner. The default owner will thus be able to alert the requester if it determines that the owner failed before responding to the request.

6.3.3. Checkpointing

A checkpointing session is initiated by a checkpoint ‘‘timer’’ associated with each process-cluster, or when a process-cluster needs to write a dirty block to stable storage. All process-clusters that have communicated (directly or indirectly) with the initiating process-cluster checkpoint together using a two-phase protocol [Gray78], exactly as in message-passing systems [Bari83, Koo87, Tami89]. Since processes of different tasks do not communicate, each

checkpointing session is localized to a task.

In the first phase, the system constructs a tree of all process-clusters that have communicated (directly or indirectly) with the process-cluster that initiates checkpointing. This communication tree is built recursively, with each node sending checkpoint messages to all nodes containing process clusters with which it has communicated directly. Upon receiving a checkpoint message for a task, the node suspends execution of all local processes of that task. No new coherency transactions are initiated or serviced for the task until the checkpointing session is terminated. When all the coherency transactions pending for the task have terminated, the node begins sending the checkpoints of the local members of the task to stable storage. In parallel, the node propagates the checkpoint message to its children — all nodes containing process clusters with which it has communicated directly. When a node completes saving the state of its process cluster and receives acknowledgements from all its children, it sends an acknowledgement to its parent. Once the checkpointing coordinator at the root of the tree is informed that all the checkpoints have been saved to stable storage, it commits its checkpoint and initiates the second phase by sending ‘‘commit’’ messages down the tree. Upon receipt of the commit message, each node commits its checkpoint.

With coordinated checkpointing in a message passing system, at any point in time there is only a *single, committed* checkpoint of every process in stable storage (Section 6.1). In a DSM system it is desirable to maintain this property and store in stable storage only a *single* copy of each block in a location that is independent of the current block owner or the existence of multiple copies of the block in system

```
checkpoint →    {
    suspend processes in process-cluster;
    await termination of pending transactions;
    reject all transaction requests;
    begin saving local process states and dirty private blocks;
    begin saving shared locally-owned blocks with dsc = 1;
    propagate checkpoint messages
    await acknowledgments for checkpoint messages;
    await completion of locally-initiated transmission and storage of checkpoint data
    acknowledge parent;
    await checkpoint commit message;
    propagate checkpoint commit message;
    commit checkpoint;
    clear all dsc bits;    }
```

Figure 6.1: A checkpointing session

nodes. This simplifies the management of checkpoints in stable storage and reduces the size of checkpoints.

Shared blocks may be replicated on multiple nodes, and may thus be saved by any one of these nodes. In order to avoid checkpointing multiple copies of the block, the responsibility for checkpointing the block, if needed, is assigned to the block owner. Each node participating in the checkpointing session checkpoints the private data of the local process cluster (registers and any other private memory) and all the shared blocks which are owned locally and for which the dsc bit is set. Since interprocess dependency used to construct the interacting set is determined by the transfer of a block with the dsc bit set (Section 6.3.1), it is guaranteed that if a block of the task has its dsc bit set in any of the participants in the checkpointing session, the current owner of the block will also be one of the participants. Hence, all the blocks of this task that have been modified on the checkpointing nodes since

their last checkpoint (for this task) will be part of the checkpointed state. The dsc bit of all these blocks are cleared at the end of the checkpointing session.

Since the checkpoint mechanism does not change the distribution of data in the local memories of the nodes, the coherency directory continues to be valid, and is, hence, not altered. The key steps in a checkpointing session are summarized in Figure 6.1.

6.3.4. Recovery

The failure of a node causes the loss of all state in that node — register state of processes that were active on that node, sections of the address space of several tasks that were resident in the node's local memory, and the fragment of the distributed coherency directory resident in the node. The most recent committed checkpoint contains the private process states and corresponding shared memory state from which correct execution may be restarted. However, the state of the coherency directory is not a part of the state of any task, and is thus not implicitly checkpointed or recovered with memory and process states. As part of the recovery session, the system must ensure that the entire distributed directory accurately reflects the distribution of copies in the system. This requires modifying (repairing) the distributed directory, possibly including directory fragments on nodes that do not participate in any rollback.

6.3.4.1. State Restoration

With coordinated recovery, the rollback of processes that were active on the failed node requires the rollback of processes on other nodes that communicated (directly or indirectly) with the failed processes [Bari83, Koo87, Tami89]. All these processes are rolled back to their previous checkpoints in coordination, employing the same two-phase protocol used for checkpointing. Recovery is initiated for each process-cluster that was active on the failed node. Since processes belonging to different tasks do not communicate with each other, different tasks can recover independently. Hence, the two-phase protocol is executed independently for each task that was active on the failed node. A task can resume execution once its state is restored and the associated coherency directory repair is completed.

The first step in recovery is to roll back to the most recent checkpoint the private states (e.g., registers) of all the processes belonging to interacting sets that include processes on the failed node. Next, all modifications to the task's shared address space performed by all these processes since their most recent checkpoints are undone. On nodes that have *not* failed, all memory blocks modified since their last checkpoint have their dsc bit set. Since interprocess dependency used to construct the interacting set is determined by the transfer of a block with the dsc bit set (Section 6.3.1), it is guaranteed that if a node that has modified a block since the node's last checkpoint is a member of the interacting set, any other node holding a copy of the block is also a member. Hence, the ''unwanted'' modifications of the shared address space can be undone by each member of the interacting set by restoring all the local blocks with the dsc bit set from their last

```
recover →      { /* in a recovering failed node */
    send rollback messages down the ‘‘rollback tree’’
    reject all transaction requests;
    invalidate MBT entries;
    send repair_directory (selfid) messages to all nodes;
    await ownership messages;
    await acknowledgments to rollback messages;
    restore processor state from checkpoint;
    send rollback commit messages to children in the rollback tree;
}

repair_directory (nodeid) → {
      /* repair coherency directory */
    invalidate local copies of all blocks owned by nodeid;
    eliminate nodeid from the copy directories of all locally-owned blocks;
    send to nodeid ownership messages for owned blocks that are default owned by nodeid;
}
```

Figure 6.2 (continued): A recovery session

checkpoint.

The failed node recovers with all the blocks in its local memory marked invalid. Hence, any modifications to the shared address space performed locally prior to rollback are automatically undone. Since the entire directory of this recovering node is invalidated, when post-recovery execution begins, every access by this node results in a miss. The required blocks will be accessed either from the last checkpoint or from the restored memory on other nodes. Since the directory is empty, references to a block that is not found locally are referred to the block’s default owner. The default owner is either the current owner or can direct the request to the current owner. The current owner either provides the block from its memory or from the last checkpoint. As described later in this section, our

```
rollback → {
    suspend processes in process-cluster;
    await for termination of pending transactions;
    reject all transaction requests;
    propagate rollback messages down the rollback tree;
    await acknowledgments to rollback messages;
    acknowledge parent;
    await rollback commit message;
      /* roll back task state */
    restore state of process-cluster from checkpoint;
    invalidate all dirty private blocks;
    forall (shared blocks with block.dsc = 1)
        if (readonly) invalidate block;
        else {
            restore block from last checkpoint;
            invalidate copy directory;
            block.dsc = 0;
        }
    propagate rollback commit messages down the recovery tree;
}
```

Figure 6.2: A recovery session

directory repair procedures ensure proper handling of recovery sessions in which a recovering failed node is the current and/or default block owner.

Interprocessor traffic associated with restoring memory blocks can be minimized by restoring only dirty blocks that are owned locally. As discussed above, all the blocks with their dsc bit set are orphans. If they are un-owned (read-only) copies, they are invalidated and, if needed, obtained incrementally during post-recovery execution. Consistency of the directory is maintained by restoring the owner's copy of the block from the last checkpoint in the exclusively

owned state (to indicate the absence of read-only copies).

6.3.4.2. Directory Repair

During the recovery session, the distributed coherency directory must be restored to a valid state. Node failure causes three inconsistencies in the directory state. First, the Migrated Block Table (MBT) entries that were in the local memory of the failed node are lost. The absence of these entries in the default owner implies that the default owner retains ownership of the blocks, leading to a conflict of ownership with the actual current owner of these blocks. Second, read-only copies that the coherency directory (on other nodes) indicates as present in the failed node, are no longer present there. Third, blocks that are owned by the failed node are no longer available in the failed node. Furthermore, the copy directories associated with these blocks are also no longer available. The repair of the coherency directory must resolve these inconsistencies.

Since the directory state possesses no application level semantics, it is not necessary to restore it to a prior valid state. It is sufficient to transform it (roll forward) to a new state that correctly represents the state of the local memories. Our scheme restores the failed node's MBT by reconstruction. All the nodes in the system update the restarting node with a list of their owned blocks that are default-owned by the failed node. The recovering node uses this information to reconstruct its MBT.

Directory inconsistencies associated with the loss of read-only blocks and of copy directories in the failed node are repaired by transforming the directory and memory state in other nodes. While several transformations are possible, we

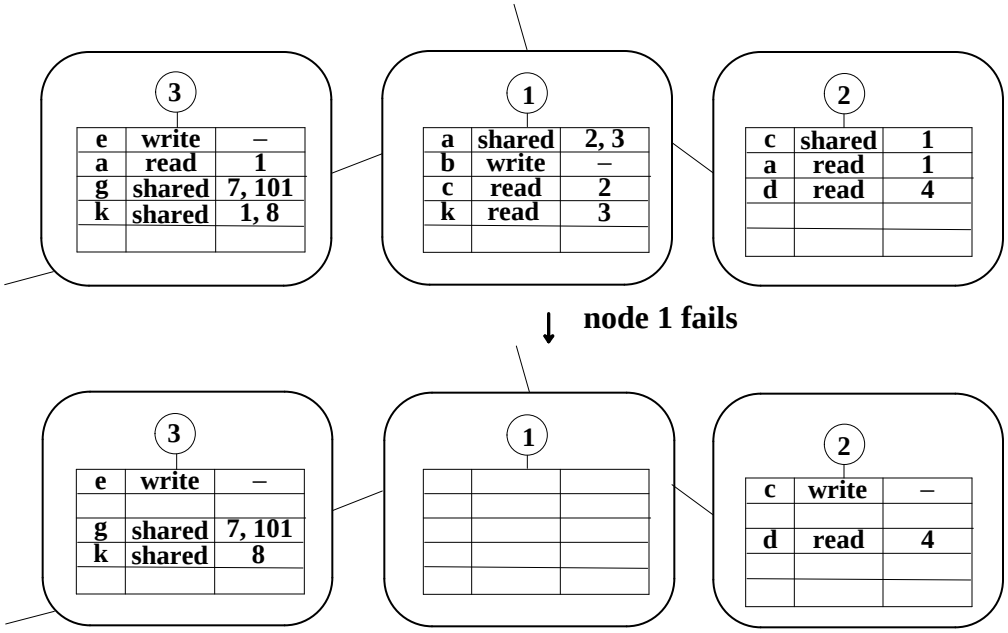


Figure 6.3: Repair of the directory state during recovery. The directory maintains the block id, the access permission, and the copy directory. A node which possesses a block in the ‘‘write’’ or ‘‘shared’’ states is the owner of the block, and maintains the location of copies. A node that possesses a ‘‘read’’ copy of a block maintains the identity of the block owner. Node 1 owns blocks *a* and *b*, and has copies of blocks *c* and *k*. Node 1 fails and later recovers with an empty memory state. Directory fragments on other nodes are repaired by marking the local copy of blocks owned by node 1 (*a* and *b*) invalid, and by eliminating references to read-only copies in node 1 (blocks *c* and *k*).

employ one in which each node requires only local information, thus minimizing network traffic. To repair the inconsistency associated with the loss of read-only copies in the failed node, each node eliminates all references to such copies from its copy directories (recall that all the nodes know the identity of the failed node). When a node loses its copy directories, it loses information about the location of read-only copies of its owned blocks. As discussed previously, this problem is

solved by each node marking *invalid* read-only copies of all local blocks owned by the failed node. This allows the recovering failed node to be restored with null copy directories for all its owned blocks.

Blocks that were owned by the failed node prior to failure are absent from its local memory following recovery. However, these blocks continue to be owned by the recovering node. Accesses to any of these owned blocks are satisfied, on demand, by restoring them with exclusive ownership from their last checkpoint.

It is important to note that the directory repair requires the participation of all nodes that hold any of the blocks that were managed or were resident in the failed node. This includes nodes that were not members of any of the recovery trees used to coordinate rollback. The repair of directory inconsistencies is illustrated in Figure 6.3. The key steps in a recovery session are summarized in Figure 6.2.

6.4. Coordinated Checkpoint and Rollback for Hierarchical DSM Systems

The coordinated checkpoint and rollback scheme presented in the previous section was designed for DSM systems in which the node memories were maintained coherent by managing the address space at a single granularity of blocks. Ownerships and read-only copies are granted and maintained at the level of blocks. Such coherency management at one level of granularity suffers from high storage overhead when the granularity is fine (small block sizes), and from poor performance when the granularity is coarse (large block or page sizes) [Tami92]. As we discussed in Chapter 3, an hierarchical DSM system [Tami92], where coherency is managed at multiple levels of granularity, can achieve the performance advantage of fine grain block-level schemes without

their high storage overhead. The organization of the coherency directory and the protocol associated with hierarchical management differ from those used with single-level management.

Adapting the coordinated checkpoint-rollback recovery scheme presented in Section 6.3 to a DSM system with hierarchical management requires key modifications in: (i) tracking dependencies between processes, (ii) the proper termination of coherency transactions, (iii) dealing with two levels of mapping and with private pages in the saving and restoring of process states, and (iv) the repair of the coherency directory. We discuss and describe these modifications below.

6.4.1. The Hierarchical DSM System

With hierarchical management (Chapter 3), ownership is maintained at *both* the page (thousands of bytes) and block (tens of bytes) levels. To limit storage overhead, pages that are dynamically private are mapped and owned only at the page level. For shared pages, ownership is maintained at both the page and block levels allowing data to be shared at the block level. Private and shared pages are distinguished *dynamically*, based on the sharing behavior of the application [Tami92].

The owner of a shared page identifies the current owner of each block of that page. A node that requires access to a block that is not available locally sends the block request to the page owner for forwarding to the block owner. The owner of the block provides read-only copies of the block and maintains the location of these read-only copies in a copy directory. To reduce storage overhead, read-only copies of blocks are *not* tagged with the identity of the block owner (unlike the one level

scheme). Instead, only the identity of the page owner is maintained. When a node (other than the current block owner) needs to write to the block, the block owner invalidates all read-only copies and transfers ownership of the block to the writing node. Since the page owner identifies the owner of each block, this transfer of block ownership is committed with the page owner.

A block owner knows which blocks it owns. The page owner is the only other node that is aware of the current owner of any of the blocks of the page. The current owner of the page is tracked by the default owner of the page [Li89b]. Default ownership is assigned at the granularity of pages. A Migrated Page Table (MPT) of each node, analogous to the MBT in the block level DSM system, maintains the current owner of all default owned pages that have been migrated away. At any time, any node can determine the current owner of any page through the default owner of the page.

6.4.2. Tracking Interprocess Communication

As described earlier, in a DSM system, processes communicate by the read-write sharing of data and dependencies can be tracked by monitoring the exchange of dirty data. In the one-level DSM system, these dependencies were tracked by associating a dirty-since-checkpoint (dsc) bit with each block of data. In the hierarchical DSM system, data is mapped at both the page and block levels — private data is mapped at the page level and shared data is mapped at both page and block levels. Hence, there are choices to be made regarding the granularity at which dependencies should be tracked in the hierarchical system.

To track dependencies at the page level, a dsc bit can be associated with each

page. Since the single page-level dsc bit is associated with all blocks of the page, processes that have accessed different blocks of the page will be treated as having shared data. Hence, checkpoints and rollbacks will be unnecessarily propagated to processes that do not have dependent states, forcing them to be checkpointed and rolled back together. The resulting larger interacting sets lead to higher performance overhead than necessary.

Dependencies can be tracked at the block level by maintaining a dsc bit for every block in memory. This can eliminate the unnecessary checkpoint and rollback propagation of page-level dependency tracking. However, such a scheme incurs significant storage overhead over maintaining a dsc bit per page, and may be wasteful. Specifically, a large fraction of the data resident in memory is dynamically private and thus mapped only at the page level. Associating a dsc bit with every block of these dynamically private pages may not provide much benefit in reducing checkpoint and rollback propagation since these pages are not currently being accessed by multiple nodes.

In order to keep both the performance and the storage overheads low, block-level dsc bits are maintained for blocks in pages that are dynamically shared while only page-level dsc bits are maintained for dynamically private pages. Since shared pages are also mapped at the block level, integrating the dsc bits with the block level data structure maintained for shared pages by the memory management mechanism is simple. Since pages dynamically change between private and shared status (depending on the pattern of accesses to the pages at execution), the system dynamically transforms between the per-page dsc bits of private pages and the per-block dsc bits of shared pages when a page changes status. When a private

page with its dsc bit set becomes shared, the dsc bit is set for each of its blocks. The dsc bit of a page that changes from shared to private status is set if the dsc bit was previously set for any of the blocks of the page. The use of these dsc bits to track dependencies is identical to the one-level scheme. The dsc bit is transferred along with the data (block or page). When data is transferred with its dsc bit set, the node updates its list of process clusters it has directly communicated with.

6.4.3. Handling of Pending Transactions

As discussed in Section 6.3.2, the handling of coherency transactions that are in progress at checkpoint or recovery is a key issue. As with the one level scheme, we adopt the approach of ensuring that all such transactions are completed (successfully or unsuccessfully) before proceeding with the checkpoint or recovery. The hierarchical DSM system performs additional, more complex, coherency transactions which must also meet the requirement of guaranteed completion. For example, in the hierarchical system, a node requesting a copy of a block may send the request to the default owner of the page to which the block belongs. This request may be forwarded by the default owner to the current owner of the page, which may, in turn, forward the request to the current owner of the block, who will service the read request directly. Similar to the scenario discussed in Section 6.3.2, the node that originated the request is unaware of the identity of the current page owner, or the current block owner, and, hence, will be unable to associate the failure of either of these nodes with the failure of the transaction.

Coherency transactions in the hierarchical DSM system can be restructured exactly as in a one-level system to guarantee their completion (successfully or

unsuccessfully) across failures. The coherency protocol is modified so that the reply is returned through the page owner and the default owner, so that the transaction can be aborted by the requesting node upon failure of any of these participant nodes. The resulting transactions are similar to the transactions on the one-level coherent system except for the presence of a page owner in addition to a block owner, and can be guaranteed to complete before checkpoint and recovery.

6.4.4. Checkpointing and Recovery of Process States

The checkpointing and recovery mechanisms respectively checkpoint and recover consistent states of an interacting set of process-clusters using a two-phase protocol, exactly as with the one-level system. The only difference that must be addressed is that in the hierarchical system some of the memory blocks are in dynamically private pages. Fortunately, the mechanism we developed for the one-level system extends naturally for the hierarchical system.

The checkpointing mechanism saves data that has been modified to stable storage. In the one-level scheme, to ensure that only a single committed copy of each memory block is maintained in stable storage, a memory block is saved only by its owner. As we discussed earlier, the hierarchical system maintains dsc bits for every block of shared pages and a single dsc bit for each private page. During checkpointing, nodes checkpoint their modified private pages and modified locally-owned blocks in shared pages. The dsc bit of private pages (subsection 6.4.2) is used to identify private pages that need to be checkpointed. As in the one-level scheme, during recovery the faulty computation is undone in the recovering nodes by locally invalidating all read-only copies of blocks that

have their dsc bit set and by restoring all modified locally-owned blocks from stable storage. In addition, all modified private pages are restored from stable storage.

6.4.5. Directory Repair

The modifications to the directory repair mechanism address two key differences between the organization of the directory in the hierarchical DSM system and its organization in the one-level (block-level) system. First, in addition to ownership at the block level, memory is also owned at the page level. The page owner maintains an additional copy directory, which identifies the current owner of each of the blocks of the page. The repair mechanism should address the possible loss of this data structure maintained by the page owner. Second, nodes that possess read-only copies of a block do not store the identity of the block owner. In the one-level scheme, when the block owner fails, nodes that possess read-only copies recognize the loss of the block owner's copy directory (which maintains the location of read-only copies) and can repair it by invalidating their read-only copies themselves. In the hierarchical DSM system, nodes that possess read-only copies cannot unilaterally identify the failure of the block owner. Rather, since only the page owner can identify the owner of each block of the page, the cooperation of this page owner is necessary to repair the loss of the copy directories.

The distributed coherency directory in the hierarchical DSM system can incur the following losses due to node failure: (i) loss of the MPT maintained in the failed node, (ii) loss of read-only copies held by the failed node, (iii) loss of the

directory which identifies the owners of blocks belonging to pages owned by the failed node, and (iv) loss of block copies owned by the failed node and their associated copy directories. The loss of the MPT and the loss of read-only copies present scenarios that are identical to the equivalent cases addressed by the one-level DSM recovery scheme and require no new solutions. As with the restoration of the MBT in the one-level system, the Migrated Page Table (MPT) is restored in the failed node using update messages. As in the one-level system, the block owner is aware of the location of read-only block copies. Hence, a block owner can repair its directory by eliminating references to read-only copies which were on the failed node.

The loss of the list of block owners on a failed page owner is equivalent to the loss of the MBT in the one-level scheme. The list of block owners serves the same function in the hierarchical DSM system as the MBT in the one-level system. It tracks the current owners of blocks, so that requests for a block can be forwarded to the current owner. In both cases block requests cannot be forwarded to the block owner if there is a loss. The page owner's directory of block owners is rebuilt using update messages just as the MBT is rebuilt in the one-level system. Any block whose ownership is not claimed by an update message is owned by the page owner.

When a node fails, it loses locally owned blocks along with their copy directories which identified the locations of read-only copies of these blocks. To repair this loss in the one-level scheme, each node in the system invalidated *its* local read-only copies of such blocks (blocks owned by the failed node). This allowed the failed node to recover with null copy directories for its owned blocks.

As mentioned earlier, this identical mechanism is not feasible in the hierarchical DSM system because nodes that possess read-only copies are not aware of the identity of the block owner. These nodes cannot *unilaterally* identify that the node owning the block has failed and thus cannot unilaterally invalidate their copy.

Since the page owner is the only node that can identify the owners of that page's blocks, the page owner can initiate the repair of directory inconsistencies caused by the failure of block owners. Specifically, if the page owner has not failed, it can determine that the block owner has failed and initiate the invalidation of all read-only copies of the block. Since the page owner cannot identify the location of these read-only copies, this invalidation can only be achieved by a broadcast. The failed node can subsequently recover with a null copy directory. (An alternative solution is to have the page owner assume exclusive ownership of these blocks. This solution will defer the broadcast invalidate traffic to the time the block is next written).

Solutions that rely on the directory of the page owner to recover from the failure of the block owner are not useful if the page owner has also failed. Hence, if the page owner fails, the copy directories of *all* the blocks of the page are invalidated (set to null). In order to permit the block owners to clear their copy directories, all nodes in the system locally invalidate their read-only copies of all the blocks of a page whose page owner has failed.

The directory repair mechanisms are illustrated in Figures 6.4 and 6.5. Figure 6.4 shows the repair of two types of losses — the loss of read-only copies and the loss of the block owner's copy directory when the block owner is distinct from the page owner. The repair of the loss of the page owner's directory of block

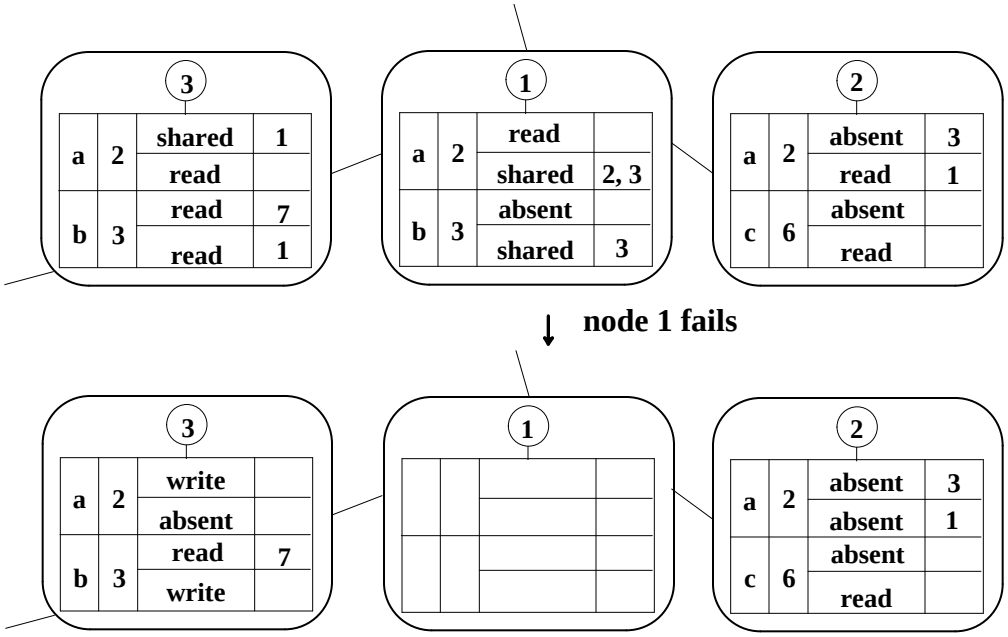


Figure 6.4: Repair of the coherency directory in a hierarchical DSM system — non-page-owner failure. The directory maintains the page id, the page owner, block access permissions, and the copy directory for each block. A page is comprised of two blocks (block *0* and block *1*). A node which possesses a block in the ‘‘write’’ or ‘‘shared’’ state is the block owner. If a node is a page owner, for every block that it does not own it maintains the identity of the block’s owner (e.g., node 2, page *a*, block *1*). Pages *a* and *b* are present but not owned in node 1 prior to failure. Node 1 fails and recovers with an empty memory state. Node 3 removes node 1 from the copy directory of block *0* of page *a* to reflect the loss of node 1’s copy. Node 2 owns page *a*, and determines that node 1 was the owner of block *1* of page *a*. Hence, node 2 broadcast invalidates all copies of block *1* of the page, and the block is restored in the exclusively owned state in node 1. The case of block *1* of page *b* is similar but here the page owner node 3 chooses to assume exclusive ownership of the block.

owner identities and the loss of the block owner’s copy directory when it is also the page owner is shown in Figure 6.5.

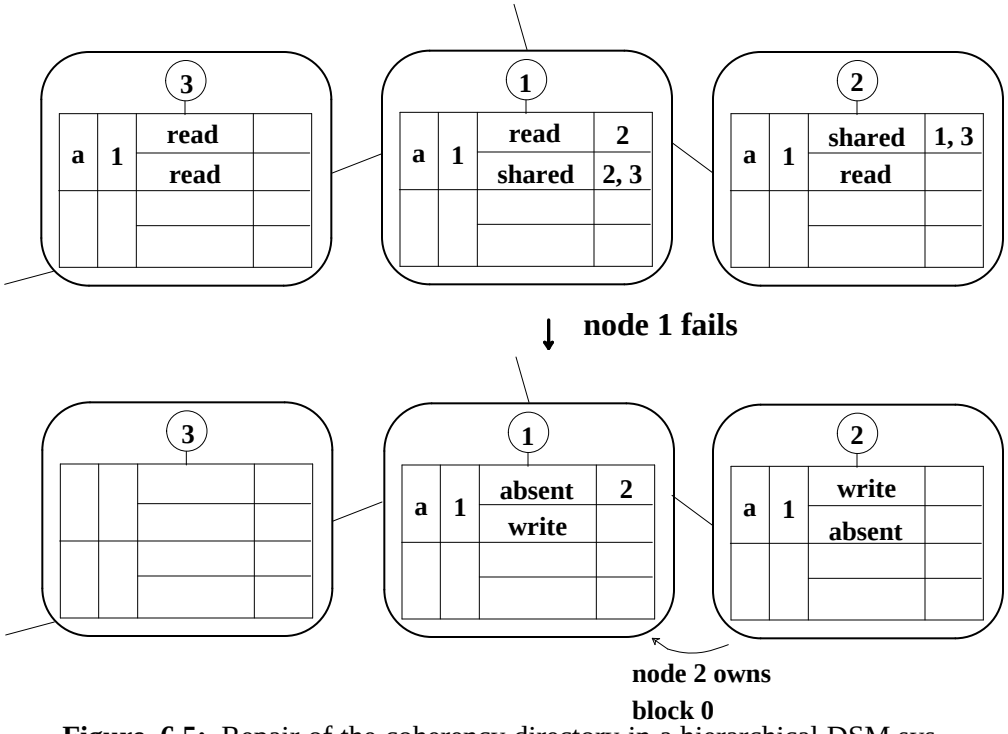


Figure 6.5: Repair of the coherency directory in a hierarchical DSM system — page owner failure. Node 1 owns page *a* prior to failure. Hence, nodes 2 and 3 invalidate their local read-only copies of all blocks of page *a* during recovery. Node 2 transfers block *0* of page *a* to the exclusively owned state, and informs recovering node 1 that it owns the block. The ownership of block *1* is unclaimed, and node 1 assumes exclusive ownership of the block.

6.5. Previous DSM Recovery Schemes

Many recovery schemes for DSM systems focus on avoiding rollback propagation, at the cost of frequent checkpointing [Wu90a, Tam90, Stum90, Jans93] or high overhead for maintaining logs [Rich93]. Due to the absence of rollback propagation, these schemes have the advantage of fast recovery, at the cost of high overhead during normal operation.

Consider a process Q that reads data modified by a process P, so that the state

of Q depends on the state of P. Rollback propagation occurs if P fails and recovers by rolling back, forcing Q to also roll back since P may compute a different value for the data during re-execution.

Wu and Fuchs [Wu90a] have proposed a scheme where rollback propagation is avoided by forcing processes to checkpoint before providing modified data to any other process. Checkpointing a process involves the flushing of pages modified by the process since the last checkpoint and saving the process state onto disk. At recovery, the processes that were running on the failed node are restarted from their checkpointed states, and re-acquire memory pages on demand. If the state of the centralized directory is lost, the directory information is reconstructed on demand. When a page is requested by a node, a special message is broadcast requesting the owner and the copy holders of the page to identify themselves to the new centralized manager. If no node claims ownership of the page, the new centralized manager assumes ownership of the page. The requested page is provided by the owner of the page, unless the owner is recovering, in which case the page is fetched from stable storage. In [Jans93], this scheme is extended to systems supporting weak consistency by checkpointing a process only when a synchronization variable is to be read by another process.

Tam and Hsu [Tam90] propose a scheme where a modified page is permitted to migrate only after all modified pages are logged to stable storage (as in [Wu90a]). At recovery, the process' state is restored from the checkpoint. The key contribution of this scheme is the use of a ''token database'' that allows the coherency directory to be repaired without consulting all the directory fragments on the system (as required by our scheme). Stumm and Zhou [Stum90] propose a

variation that tolerates only single faults. When a node is about to migrate modified data to another node, all modified data from the first node is checkpointed in the local memory of the second node.

Richard and Singhal present a recovery scheme for an environment where checkpoints are costly [Rich93]. All pages read by a process are logged to volatile storage. When a modified page is to be read by another process, the log is committed to stable storage. Upon failure of a process, the process is restored from the checkpoint, and its up-to-date state is recomputed using the logged pages. With this scheme there is no rollback propagation and no need for frequent checkpointing. However, there is high overhead for collecting, committing, and storing the logs.

6.6. Evaluation

A key measure of error recovery schemes based on checkpointing and rollback is the overhead they incur during normal operation. This overhead includes the cost of checkpointing as well as any other special processing, such as for keeping track of dependencies, logging messages, etc. Our evaluation is focused on the overhead for checkpointing, specifically on the extra interconnection network traffic due to checkpointing. We compare our coordinated checkpointing scheme to the scheme proposed by Wu and Fuchs [Wu90a].

Our evaluation is based on trace-driven simulation, using address traces from a possible execution on a 64 node multiprocessor of three parallel applications: Weather, Speech, and FFT. Weather uses finite difference methods to solve a set of partial differential equations describing the state of the atmosphere. Speech uses

a modified Viterbi search algorithm to find the best match between paths through a directed graph representing a dictionary and another through a directed graph representing the input. FFT is a radix-2 fast Fourier transform. Details about the applications and the traces can be found in [Chai90].

The simulator faithfully implements the checkpointing phase of both recovery schemes. We have implemented the simulator both for the block-level DSM system and the hierarchical DSM system. The interconnection network traffic includes interprocessor traffic as well as traffic to stable storage. This traffic is calculated assuming that the size of a control message is 16 bytes, the size of data messages is 16 bytes plus the size of the data being transmitted, and that the size of a process' register state is 128 bytes. Simulations were performed for a range of block sizes to evaluate the block-level schemes. For the hierarchical DSM system, simulations were performed for a page size of 1 KB and various block sizes. For coordinated checkpointing the impact of varying the checkpointing frequency was measured. With the block-level scheme, the system allows a maximum of 512 blocks per node to be in a shared state, using a 512 entry Shared Blocks Table (SBT). Similarly, with the hierarchical system, a maximum of 64 pages per node may contain a block in a shared state, using a 64 entry Shared Pages Table (SPT) [Tami92].

6.6.1. Results for the Block-Level DSM system

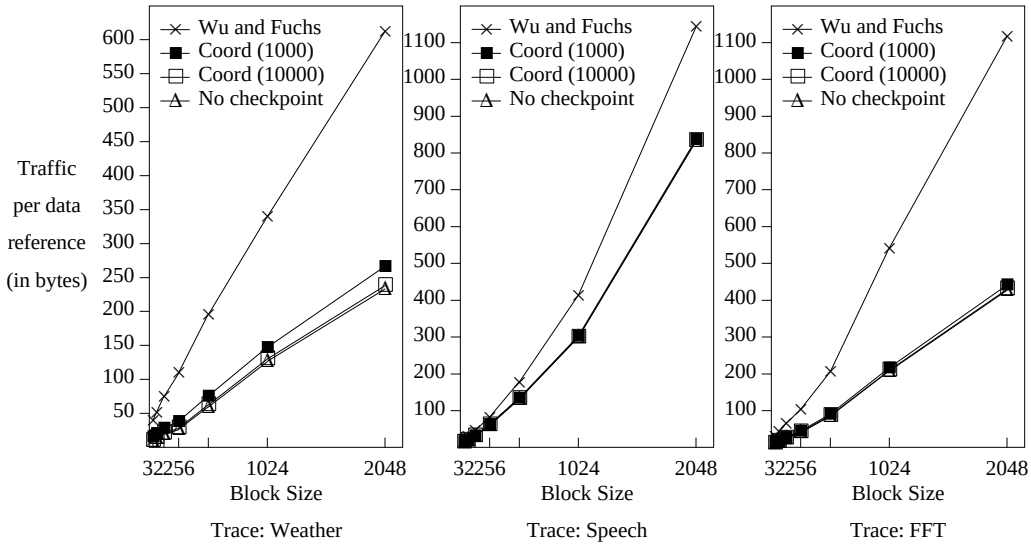
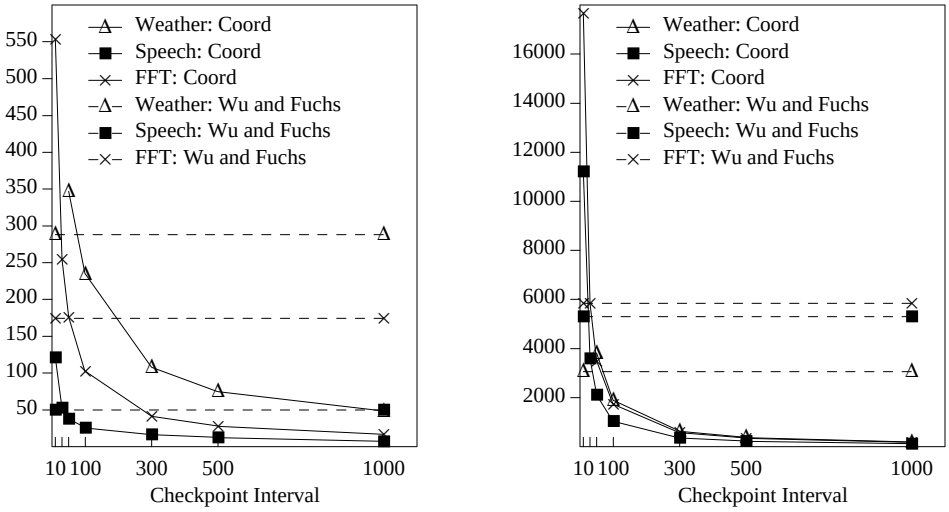


Figure 6.6: Network traffic in a block-level system as a function of block size. Results for a system without checkpointing, with the Wu and Fuchs [Wu90a] scheme, and with coordinated checkpointing. With coordinated checkpointing, results for two checkpointing intervals: every 1000 and every 10000 references by any one processor.

With a block-level DSM system, the simulation results for coordinated checkpointing and for the Wu and Fuchs scheme [Wu90a] are shown in Figures 6.6 and 6.7. Figure 6.6 shows the network traffic per data reference as a function of the block size. In all three applications, the additional traffic due to coordinated checkpointing is low relative to the traffic without checkpointing for a wide range of block sizes. The Wu and Fuchs scheme [Wu90a] generates significantly higher network traffic than the coordinated checkpointing scheme, especially with larger blocks. Larger blocks increase false-sharing, thus increasing the frequency of read-write sharing. With the Wu and Fuchs scheme, this forces an increase in the frequency of checkpointing and thus increased overhead. Hence, the Wu and

Fuchs scheme will not be a good choice for DSM systems that use large blocks (e.g., 1 KB blocks are used in [Li89a] and [Wu90a]).



a) Relative traffic overhead as percentage of normal traffic

b) Node checkpoints per 10^5 system references

Figure 6.7: The impact of the checkpointing interval in a block-level system with 128 byte blocks. The relative traffic overhead is the additional traffic due to checkpointing relative to the traffic without checkpointing. The checkpointing interval is the maximum number of references made by any processor between checkpoints — it is an approximate measure of the time between checkpoints. For the Wu and Fuchs scheme, the checkpointing interval is determined by the behavior of the application and cannot be set by the system.

Figure 6.7 shows the impact of changing the checkpointing interval for a system with 128 byte blocks. For coordinated checkpointing, as the checkpointing frequency is decreased, the overhead decreases approximately linearly. Hence, if the requirements of the application (e.g., real-time response specifications) allow, the overhead for coordinated checkpointing can be reduced by simply lowering the checkpointing frequency. With the Wu and Fuchs scheme, the checkpointing

interval is a characteristic of the application and cannot be set by the system. Hence, the checkpointing overhead is fixed and high regardless of the needs of the application. Thus, these results demonstrate the benefits of being able to tune the frequency of checkpoints in an *application-independent* manner.

The Wu and Fuchs scheme has the advantage of checkpointing traffic which is evenly distributed over time. With coordinated checkpointing, the checkpointing traffic is likely to be burstier. This disadvantage of coordinated checkpointing can be eliminated through the use of asynchronous coordinated checkpointing with a memory system that supports copy-on-write pages [Tami91]. In this case, volatile checkpoints are taken locally on each node and are spooled gradually to stable storage once normal operation resumes.

6.6.2. Results for the Hierarchical DSM System

As shown in Figure 6.8, in terms of network traffic overhead, the coordinated checkpointing scheme retains its advantage over the Wu and Fuchs scheme even on the hierarchical DSM system.

Even without checkpointing, for the same block size, the hierarchical DSM system often requires higher network bandwidth than a block-level system. As shown in Figure 6.8, this disadvantage of the hierarchical system is often exacerbated in a system with checkpointing. Both coordinated checkpointing and the Wu and Fuchs scheme incur higher checkpointing traffic on the hierarchical system. The additional traffic on the hierarchical DSM system is due to its support for page-level mapping and transfers. Since private data is mapped in pages, they must be checkpointed in pages rather than blocks. Also, when a private page that

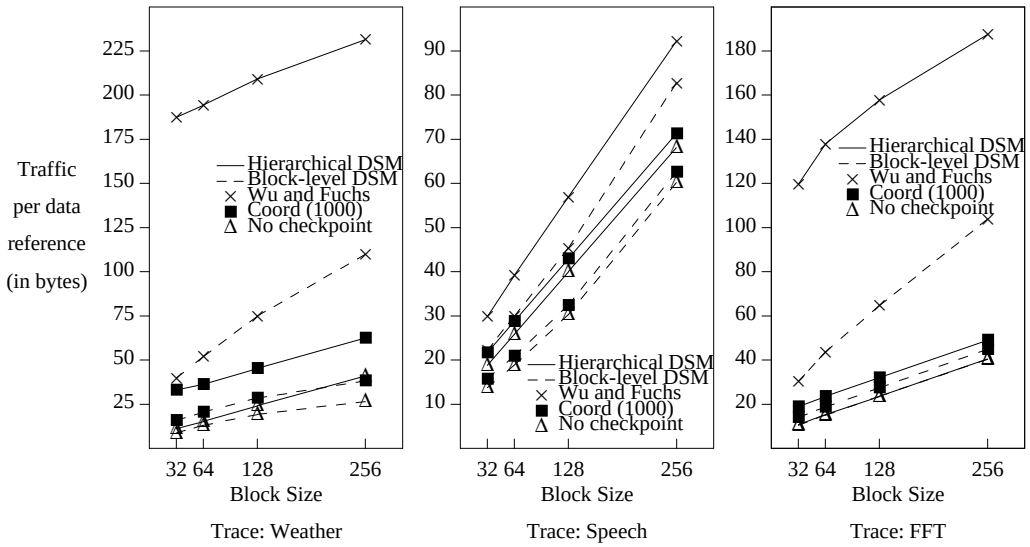


Figure 6.8: The impact of block size on network traffic for the hierarchical and the block level systems with coordinated checkpointing and with the Wu and Fuchs scheme. The hierarchical system uses 1 K byte pages. A checkpointing interval of 1000 references is used with coordinated checkpointing. For a given block size, network traffic, with and without checkpointing, is higher in the hierarchical system than the block-level system.

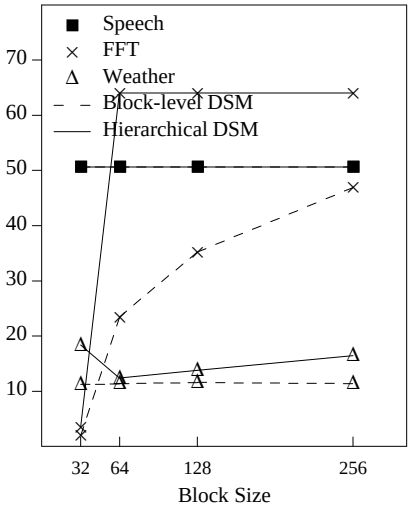
has its dsc bit set moves to the shared state, the dsc bits of all blocks are set, even if only a few blocks were modified. In both cases, a page of blocks may be checkpointed when only a few of the blocks actually require checkpointing.

The absence of block-level dsc bits for private pages in the hierarchical system can lead to larger interacting sets with coordinated checkpointing and more frequent checkpointing with the Wu and Fuchs scheme. The dsc bit of a private page is set when it is modified. If the page is subsequently read by another process, the page is transferred to a shared state and the dsc bits of all the blocks of the page are set. With the Wu and Fuchs scheme, the process that modified the page must checkpoint. With coordinated checkpointing, the process reading the

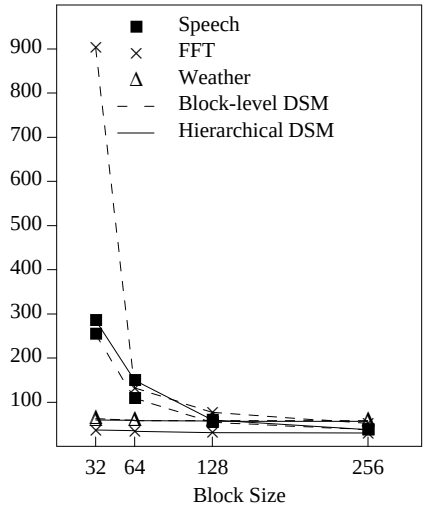
page must be included in the interacting set. In the block-level DSM system, a read issued from another process would necessitate these actions only if the block being read was modified.

Figure 6.9 shows that the average size of the interacting set with coordinated checkpointing and the checkpointing frequency with the Wu and Fuchs recovery scheme are generally higher on the hierarchical system than on the block-level system using the same block size. With coordinated checkpointing at an interval of 10000 processor references (Figure 6.9-a), both FFT and Weather have larger interacting sets on the hierarchical system, especially at larger block sizes. With the Weather application, there is the somewhat surprising phenomenon of the interacting set size initially *decreasing* when the block size of the hierarchical system is increased. Such behavior is possible when false sharing forces the checkpoint of the sub-blocks that comprise the large block. This effectively clears the dsc bit that would have been associated with each of the sub-blocks if the system used smaller blocks. With smaller blocks, the dsc bits of these sub-blocks would remain set, causing the interacting set of subsequent checkpoint sessions to be larger than when the large blocks are used. With Speech, within the range of block sizes studied, larger blocks do not cause significant additional false sharing. Hence, with coordinated checkpointing, the interacting set size is not affected by the block size or by the use of the hierarchical scheme versus the block-level scheme. With the Wu and Fuchs scheme, the expected decrease in checkpointing interval (increase in checkpointing frequency) on the hierarchical system is observed only for FFT (Figure 6.9-b).

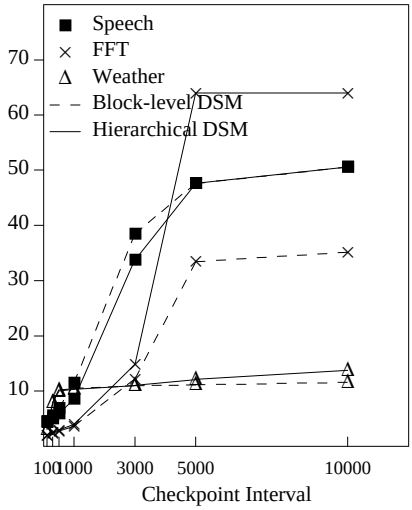
Without tracking interprocess dependencies, every checkpoint or rollback



a) Interacting set size, coordinated checkpointing, checkpoint interval of 10000 references



b) References per processor between checkpoints, Wu and Fuchs scheme



c) Interacting set size, coordinated checkpointing, 128 byte blocks

Figure 6.9: The impact of the hierarchical scheme on the interacting set size with coordinated checkpointing and on checkpointing frequency with the Wu and Fuchs scheme. The hierarchical system uses 1 KB pages.

must be global — involving all the processes of the task. Figures 6.9-b and 6.9-c demonstrate that with real applications tracking dependencies can reduce the size of the set of processes that need to be checkpointed or rolled back together. As shown in these figures, the precise beneficial impact (if any) of tracking dependencies depends on the application, the checkpointing interval, the block size, and whether a hierarchical or block-level scheme is used.

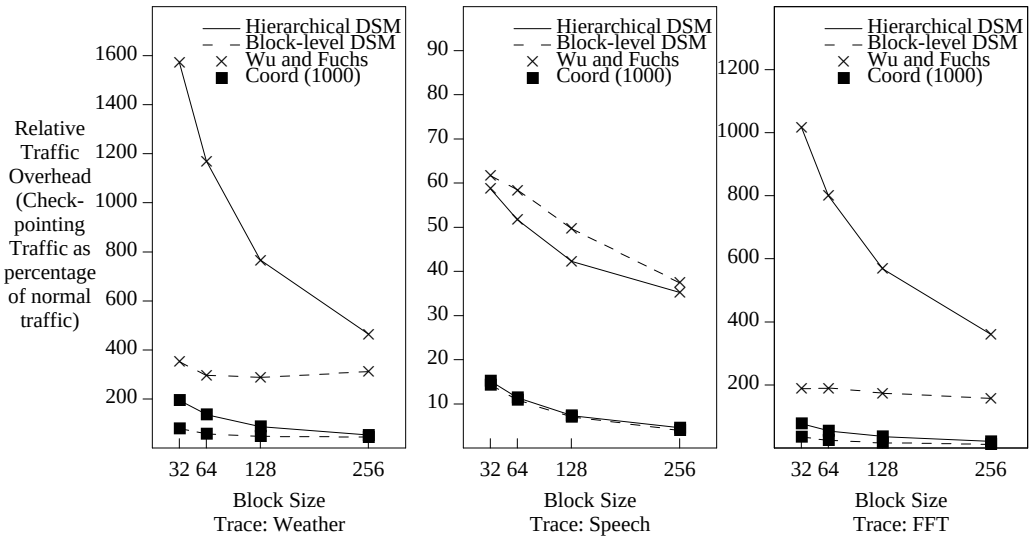


Figure 6.10: Comparison of relative traffic overhead in the hierarchical and block-level systems at various block sizes. The relative traffic overhead is the additional traffic due to checkpointing expressed as a percentage of the traffic without checkpointing. Results for the Wu and Fuchs scheme and the the coordinated checkpointing scheme. Coordinated checkpointing scheme results are shown at a checkpoint interval of 1000 per-processor references. The hierarchical system uses 1K byte pages.

Figure 6.10 compares the relative traffic overhead due to checkpointing in the hierarchical and block-level DSM systems. The relative traffic overhead is the additional traffic due to checkpointing expressed as a percentage of traffic without checkpointing. For Weather and FFT, the relative traffic overhead with both

checkpointing schemes is higher in the hierarchical system. As previously discussed, one of the key reasons for the increase in checkpointing traffic in the hierarchical system is that often whole pages instead of blocks are checkpointed. Specifically, with these applications, 80-90% of the data that is checkpointed is checkpointed as pages. With the Speech application, traffic due to checkpointing is small in proportion to the traffic on a system without checkpointing. The heavy read-sharing of a dictionary in Speech causes the traffic without checkpointing to be higher on the hierarchical system than on the block-level system. However, since this data is not modified during most of the application, it does not need to be checkpointed so there is little change in checkpointing traffic between the hierarchical and block-level systems.

Figure 6.7-b, shows that checkpoints occur far less frequently with coordinated checkpointing using an interval of 1000 references than with the Wu and Fuchs scheme. As a result, as shown in Figure 6.10, the increase in overhead in the hierarchical system is significantly smaller with coordinated checkpointing than with the Wu and Fuchs scheme.

The difference between the relative traffic overhead of checkpointing on the hierarchical system and on the block-level system decreases with increasing block size (Figure 6.10). This is due to the fact that one of the key reasons for the increase in checkpointing traffic in the hierarchical system is that often whole pages instead of blocks are checkpointed. As the block size decreases, this effect becomes more significant. Thus, for small blocks the negative impact of page-level management in the hierarchical system is more severe on the checkpointing performance than on the performance of the non-checkpointing system.

Chapter Seven
Summary

For many applications that target distributed memory multicomputers for their scalability and high-performance, it is desirable to provide transparent Distributed Shared Memory (DSM). For such applications, the hardware and system software must maintain coherency among the local memories. To assure sustained progress of large applications through the failures of nodes and their associated local memory, the system must also incorporate a mechanism that ensures the continued availability of the entire shared address space in spite of these failures.

Most existing techniques for providing DSM on multicomputers perform the mapping, data transfers, and ownership management at a single granularity of blocks (or pages). If large blocks are used, these techniques suffer from long latencies for remote requests, they require higher network bandwidth than is inherently needed by the application, and they lead to unnecessary transfers due to false sharing. Smaller blocks lead to a decrease in miss ratio [Dubo88], a reduction in remote access latencies, and a reduction in the required network bandwidth. However, for small blocks, management of DSM at the block level results in unacceptably high storage overhead for the various mapping tables.

We have proposed the use of multiple levels of mapping and coherency management, a new organization for the copy directory, and inverted tables as a solution to this problem. The proposed scheme dynamically distinguishes between shared and private pages in order to manage each class efficiently. Ownership is maintained at both the block and page level for shared pages while ownership is

maintained only at the page level for private pages. In order to limit the storage overhead for coherency management, tables that keep track of block copies are not allowed to grow beyond a static bound. In a realistic example of a large multicomputer, less than 4% of the total system storage will be dedicated to tables needed for coherency management.

The design of correct distributed directory-based coherence protocols is one of the main tasks in DSM implementation. This design is significantly more complicated than the design of snoopy coherence protocols due to the absence of a serializing medium such as a shared bus and the non-atomicity of accesses. These factors introduce problems associated with deadlocks, race conditions, and the handling replacements.

We have designed a DSM protocol for hierarchically managed DSM as well as DSM managed at a single block level. We employ standard mechanisms, similar to those used in [Leno93], to handle deadlocks and replacement hints. Deadlocks are avoided by requiring that invalidate messages and reply messages are always accepted and processed (‘‘sunk’’) at their destinations. Update messages due to replacements are handled with additional states in the coherence protocol. We developed the use of *bilateral exchanges* as opposed to forwarding as a solution to the problem of races. Specifically, in a pairwise-FIFO network, the race problem is avoided by restricting all protocol exchanges to be bilateral. Since the use of bilateral exchanges involves a slight increase in message count, we have developed a hybrid protocol, where read accesses use forwarding and write accesses use bilateral exchanges.

Using the Proteus [Brew92b] execution-driven simulation environment, we

have implemented detailed simulations of our DSM protocol for hierarchically managed DSM as well as DSM managed at a single block level. We have evaluated the performance of the hierarchical management scheme using execution-driven simualtions and trace-driven simulations. In general, the results indicate that significant performance improvement over conventional page-level schemes is possible with our two-level scheme, without incurring the storage overhead that makes uniform block-level schemes impractical in large systems. The results verify that only a small fraction of the shared address space is actually shared at any time. This validates the benefits of a scheme, such as our hierarchical scheme, where coherency tables (e.g., our shared pages table) can be kept small since they maintain information only for dynamically shared data. The results also show that our new copy directory organization is more space-efficient than the conventional organization used in most DSM systems.

Many error recovery schemes for multiprocessors and multicomputers are based on checkpointing a process whenever it communicates with another process or on the use of message logging techniques. For scalable high performance systems, used for communication-intensive applications, all of these schemes incur unacceptably high overhead.

We have developed an error recovery scheme for DSM multicomputers based on coordinated checkpointing and rollback. Our scheme is an adaptation of coordinated checkpointing and rollback for message-passing systems. This adaptation efficiently solves several potential difficulties with DSM systems which are due to the lack of explicit messages, the presence of a shared replicated address space, and the distributed coherency directory. We have developed mechanisms to

identify and track read-write communication in the DSM system and to checkpoint and recover the shared replicated address space. Our scheme does not checkpoint and rollback the coherency directory. Instead, at recovery, the distributed coherency directory is ‘‘repaired’’ to restore it to a correct state. Trace-driven simulations have demonstrated the significant performance advantages of our recovery scheme. By allowing lower checkpointing frequency, leading to lower overall checkpointing traffic, our approach can result in dramatically lower overhead than alternate schemes, leading to minimal impact on the normal operation of the system.

Bibliography

[Adve90] S. V. Adve and M. D. Hill, ‘‘Weak Ordering - A New Definition,’’ *17th Symposium on Computer Architecture*, pp. 2-14 (May 1990).

[Agar88] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, ‘‘An Evaluation of Directory Schemes for Cache Coherence,’’ *15th Symposium on Computer Architecture*, pp. 280-289 (May 1988).

[Agar95] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, ‘‘The MIT Alewife Machine: Architecture and Performance,’’ *22nd Symposium on Computer Architecture*, pp. 2-13 (June 1995).

[Ahme90] R. E. Ahmed, R. C. Frazier, and P. N. Marinos, ‘‘Cache-Aided Rollback Error Recovery (CARER) Algorithms for Shared-Memory Multiprocessor Systems,’’ *20th Fault-Tolerant Computing Symposium*, (1990).

[Arch85] J. Archibald and J.-L. Baer, ‘‘An Economical Solution to the Cache Coherence Problem,’’ *15th International Symposium on Computer Architecture*, pp. 355-362 (June 1985).

[Arch87] J. Archibald, ‘‘The Cache-Coherence Problem in Shared-Memory Multiprocessors,’’ Ph.D. dissertation, University of Washington (March 1987).

[Atha88] W. C. Athas and C. L. Seitz, ‘‘Multicomputers: Message-Passing Concurrent Computers,’’ *Computer* **21**(8) pp. 9-24 (August 1988).

[Bari83] G. Barigazzi and L. Strigini, ‘‘Application-Transparent Setting of Recovery Points,’’ *13th Fault-Tolerant Computing Symposium*, pp. 48-55 (June 1983).

[Bayl89] S. J. Baylor and B. D. Rathi, ‘‘A study of the Memory Reference Behavior of Engineering/Scientific Applications in Parallel Processors,’’ *International Conference on Parallel Processing*, pp. I/78-I/82 (August 1989).

[Bern88] P. A. Bernstein, ‘‘Sequoia: A Fault-Tolerant Tightly Couple Multiprocessor for Transaction Processing,’’ *IEEE Computer* **21** pp. 37-45 (February 1988).

[Blel91] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha, ‘‘A Comparison of Sorting Algorithms for the Connection Machine CM-2,’’ *Proceedings of the Symposium on Parallel Algorithms and Architectures*, pp. 3-16 (July 1991).

[Bolo89] W. J. Bolosky, R. P. Fitzgerald, and M. L. Scott, ‘‘Simple But Effective Techniques for NUMA Memory Management,’’ *Twelfth ACM Symposium on Operating Systems Principles*, pp. 19-31 (December 1989).

[Borg83] A. Borg, J. Baumbach, and S. Glazer, ‘‘A Message System Supporting Fault-Tolerance,’’ *9th Symposium on Operating Systems Principles*, pp. 90-99 (October 1983).

[Brew92a] E. A. Brewer, A. Colbrook, C. N. Dellarocas, and W. A. Weihl, ‘‘PROTEUS: A High-Performance Parallel-Architecture Simulator,’’ *1992 ACM Sigmetrics and Performance*, pp. 247-8

(June 1992).

[Brew92b] E. A. Brewer and C. N. Dellarocas, ‘‘Proteus User Documentation,’’ Technical Document, Massachusetts Institute of Technology, Cambridge, MA (March 1992).

[Burk92] H. Burkhardt III, ‘‘Overview of the KSR1 Computer System,’’ Technical Report KSR-TR-9202001, Kendall Square Research, Boston (February 1992).

[Cens78] L. Censier and P. Feautrier, ‘‘A New Solution to Coherence Problems in Multicache Systems,’’ *IEEE Transacations on Computers* **C-27**(12) pp. 1112-1118 (December 1978).

[Chai90] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, ‘‘Directory-Based Cache Coherence in Large Scale Multiprocessors,’’ *IEEE Computer* **23**(6) pp. 49-58 (June 1990).

[Chai91] D. Chaiken, J. Kubiatowicz, and A. Agarwal, ‘‘LimitLESS Directories: A Scalable Cache Coherence Scheme,’’ *Architecture Support for Programming Languages and Operating Systems-IV*, (April 1991).

[Chan85] K. M. Chandy and L. Lamport, ‘‘Distributed Snapshots: Determining Global States of Distributed Systems,’’ *ACM Transactions on Computer Systems* **3**(1) pp. 63-75 (February 1985).

[Chan88] A. Chang and M. F. Mergen, ‘‘801 Storage: Architecture and Programming,’’ *ACM Transactions on Computer Systems* **6**(1) pp. 28-50 (February 1988).

[Cher91] D. R. Cheriton, H. A. Goosen, and P. D. Boyle, ‘‘Paradigm: A Highly Scalable Shared-Memory Multicomputer Architecture,’’ *IEEE Computer* **24**(2) pp. 33-48 (February 1991).

[Chow88] E. Chow, H. Madan, J. Peterson, D. Grunwald, and D. Reed, ‘‘Hyperswitch Network for the Hypercube Computer,’’ *15th Symposium on Computer Architecture*, pp. 90-99 (May 1988).

[Dare87] F. Darema-Rogers, G. F. Pfister, and K. So, ‘‘Memory Access Patterns of Parallel Scientific Programs,’’ *Performance Evaluation Review* **15**(1) pp. 46-58 (May 1987).

[Dubo86] M. Dubois, C. Scheurich, and F. A. Briggs, ‘‘Memory Access Buffering in Multiprocessors,’’ *13th Symposium on Computer Architecture* **14**(2) pp. 434-442 (June 1986).

[Dubo88] M. Dubois and J.-C. Wang, ‘‘Shared Data Contention in a Cache Coherence Protocol,’’ *International Conference on Parallel Processing* **1** pp. 146-155 (August 1988).

[Fraz89] T. M. Frazier and Y. Tamir, ‘‘Application-Transparent Error-Recovery Techniques for Multicomputers,’’ *The Fourth Conference on Hypercubes, Concurrent Computers, and Applications* **1** pp. 103-108 (March 1989).

[Fraz95] T. M. Frazier, ‘‘Application-Transparent Error Recovery Techniques for Multicomputers,’’ Ph.D. Dissertation, Computer Science Department Technical Report CSD-950012, University of California, , Los Angeles, CA (January 1995).

[Full78] S. H. Fuller, J. K. Ousterhout, L. Raskin, P. I. Rubinfeld, P. J. Sindhu, and R. J. Swan, ‘‘Multi-Microprocessors: An Overview and Working Example,’’ *Proceedings of the IEEE* **66**(2)(February 1978).

[Ghar90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, ‘‘Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors,’’ *17th Symposium on Computer Architecture*, pp. 15-26 (May 1990).

[Ghar91] K. Gharachorloo, A. Gupta, and J. Hennessy, ‘‘Performance Evaluation of Memory Consistency Models for Shared Memory Multiprocessors,’’ *Architecture Support for Programming Languages and Operating Systems-IV*, pp. 245-257 (April 1991).

[Good83] J. R. Goodman, ‘‘Using Cache Memory to Reduce Processor-Memory Traffic,’’ *10th Annual Symposium on Computer Architecture*, pp. 124-131 (June 1983).

[Good88] J. R. Goodman and P. J. Woest, ‘‘The Wisconsin Multicube,’’ *15th Symposium on Computer Architecture*, pp. 422-431 (May 1988).

[Gray78] J. N. Gray, ‘‘Notes on Data Base Operating Systems,’’ pp. 393-481 in *Operating Systems: An Advanced Course*, ed. R. Bayer, R. M. Graham and G. Seegmuller, Springer-Verlag, Berlin (1978). Lecture Notes in Computer Science 60.

[Gupt90] A. Gupta, W.-D. Weber, and T. Mowry, ‘‘Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes,’’ *International Conference on Parallel*

Processing, pp. 312-321 (August 1990).

[Gust92] D. B. Gustavson, D. V. James, and H. A. Wiggers, ‘‘Overview of the Scalable Coherent Interface, IEEE STD 1596 (SCI).,’’ *1992 IEEE Nuclear Science Symposium and Medical Imaging Conference* **1** pp. 488-90 (October 1992).

[Hage92] E. Hagersten, A. Landin, and S. Haridi, ‘‘DDM - A Cache-Only Memory Architecture,’’ *IEEE Computer Magazine*, pp. 44-54 (September 1992).

[IEEE93] IEEE, *IEEE Standard for Scalable Coherent Interface (SCI)*. August 1993.

[Jana94] G. Janakiraman and Y. Tamir, ‘‘Coordinated Checkpointing-Rollback Error Recovery for Distributed Shared Memory Multicomputers,’’ *Symposium on Reliable Distributed Systems*, pp. 42-51 (October 1994).

[Jans93] B. Janssens and W. K. Fuchs, ‘‘Relaxing Consistency in Recoverable Distributed Shared Memory,’’ *23rd International Symposium on Fault-Tolerant Computing*, pp. 155-163 (Jun 1993).

[John87] D. B. Johnson and W. Zwaenepoel, ‘‘Sender-Based Message Logging,’’ *17th Fault-Tolerant Computing Symposium*, pp. 14-19 (July 1987).

[John88] D. B. Johnson and W. Zwaenepoel, ‘‘Recovery in DIstributed Systems Using Optimistic Message Logging and Checkpointing,’’ *7th Symposium on Principles of Distributed Computing*, pp. 171-

181 (August 1988).

[Katz85] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon, ‘‘Implementing a Cache Consistency Protocol,’’ *12th Symposium on Computer Architecture*, pp. 276-283 (June 1985).

[Koo87] R. Koo and S. Toueg, ‘‘Checkpointing and Rollback-Recovery for Distributed Systems,’’ *IEEE Transactions on Software Engineering* **SE-13**(1) pp. 23-31 (January 1987).

[Kopp95] D. Koppelman, ‘‘Version L3.5 Proteus Changes,’’ Technical Document, Louisiana State University (August 1995).

[Kusk94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharacharloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, ‘‘The Stanford FLASH Multiprocessor,’’ *21st Symposium on Computer Architecture*, pp. 302-313 (April 1994).

[Lamp79] L. Lamport, ‘‘How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs,’’ *IEEE Transactions on Computers* **C-28**(9) pp. 690-691 (September 1979).

[Leno90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, ‘‘The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor,’’ *17th Symposium on Computer Architecture*, pp. 148-159 (May 1990).

[Leno93] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, A. Gupta, and J. Hennessy, ‘‘The DASH Prototype: Logic Overhead and

Performance,'' *IEEE Transactions on Parallel and Distributed Systems* **4**(1) pp. 41-61 (January 1993).

[Li86a] K. Li and P. Hudak, ''Memory Coherence in Shared Virtual Memory Systems,'' *Fifth Symposium on Principles of Distributed Computing*, pp. 229-239 (August 1986).

[Li86b] K. Li, ''Shared Virtual Memory on Loosely Coupled Multiprocessors,'' Ph.D. Dissertation, YALEU/DCS/RR-492, Yale University (September 1986).

[Li88] K. Li, ''IVY: A Shared Virtual Memory System for Parallel Computing,'' *International Conference on Parallel Processing*, pp. II/94-II/101 (August 1988).

[Li89a] K. Li and R. Schaefer, ''A Hypercube Shared Virtual Memory System,'' *International Conference on Parallel Processing*, pp. I/125-I/132 (August 1989).

[Li89b] K. Li and P. Hudak, ''Memory Coherence in Shared Virtual Memory Systems,'' *ACM Transactions on Computer Systems* **7**(4) pp. 321-359 (November 1989).

[Lipt68] J. S. Liptay, ''Structural Aspects of the System/360 Model 85, Part II: The Cache,'' *IBM Systems Journal* **7**(1) pp. 15-21 (1968).

[Lisk87] B. Liskov, R. Scheifler, E. Walker, and W. Weihl, ''Orphan Detection,'' *17th Fault-Tolerant Computing Symposium*, pp. 2-7 (July 1987).

[Merl78] P. M. Merlin and B. Randell, ''State Restoration in Distributed

Systems,'' *8th Fault-Tolerant Computing Symposium*, pp. 129-134 (June 1978).

[O'Kr90] B. W. O'Krafka and A. R. Newton, ''An Empirical Evaluation of Two Memory-Efficient Directory Methods,'' *17th Symposium on Computer Architecture*, pp. 138-147 (May 1990).

[Pfis85] G. F. Pfister and V. A. Norton, '''Hot Spot' Contention and Combining in Multistage Interconnection Networks,'' *IEEE Transactions on Computers* **C-34**(10) pp. 943-948 (October 1985).

[Pink92] T. J. Pinkston and S. J. Baylor, ''Parallel Processor Memory Reference Analysis: Examining Locality and Clustering Potential,'' *Fifth SIAM Conference on Parallel Processing for Scientific Computing*, pp. 513-18 (1992).

[Powe83] M. L. Powell and D. L. Presotto, ''Publishing: A Reliable Broadcast Communication Mechanism,'' *9th Symposium on Operating Systems Principles*, pp. 100-109 (October 1983).

[Rama89] U. Ramachandran, M. Ahamad, and M. Y. A. Khalidi, ''Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer,'' *International Conference on Parallel Processing*, pp. II/160-II/169 (August 1989).

[Rand78] B. Randell, P. A. Lee, and P. C. Treleaven, ''Reliability Issues in Computing System Design,'' *Computing Surveys* **10**(2) pp. 123-165 (June 1978).

[Rich93] G. G. Richard III and M. Singhal, ''Using Logging and

Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory,'' *12th Symposium on Reliable Distributed Systems*, pp. 58-67 (October 1993).

[Sche87] C. Scheurich and M. Dubois, ''Correct Memory Operation of Cache-based Multiprocessors,'' *14th Symposium on Computer Architecture*, pp. 234-243 (Jun 1987).

[Schn84] F. B. Schneider, ''Byzantine Generals in Action: Implementing Fail-Stop Processors,'' *ACM Transactions on Computer Systems* **2**(2) pp. 145-154 (May 1984).

[Silb94] A. Silberschatz and P. B. Galvin, *Operating System Concepts*, Addison-Wesley (1994).

[Sing92] J. P. Singh, W.-D. Weber, and A. Gupta, ''SPLASH: Stanford Parallel Applications for Shared Memory,'' *Computer Architecture News* **20**(1) pp. 5-44 (March 1992).

[Sist89] A. P. Sistla and J. L. Welch, ''Efficient Distributed Recovery Using Message Logging,'' *8th Symposium on Principles of Distributed Computing*, pp. 223-238 (August 1989).

[Stro85] R. E. Strom and S. Yemini, ''Optimistic Recovery in Distributed Systems,'' *ACM Transactions on Computer Systems* **3(3)** pp. 204-226 (August 1985).

[Stum90] M. Stumm and S. Zhou, ''Fault Tolerant Distributed Shared Memory Algorithms,'' *Second Symposium on Parallel and Distributed Computing*, pp. 719-724 (December 1990).

[Tami83] Y. Tamir and C. H. Sequin, ‘‘Self-Checking VLSI Building Blocks for Fault-Tolerant Multicomputers,’’ *International Conference on Computer Design*, pp. 561-564 (November 1983).

[Tami84] Y. Tamir and C. H. Sequin, ‘‘Error Recovery in Multicomputers Using Global Checkpoints,’’ *13th International Conference on Parallel Processing*, pp. 32-41 (August 1984).

[Tami89] Y. Tamir and T. M. Frazier, ‘‘Application-Transparent Process-Level Error Recovery for Multicomputers,’’ *Hawaii International Conference on System Sciences-22*, pp. 296-305 (January 1989).

[Tami91] Y. Tamir and T. M. Frazier, ‘‘Error-Recovery in Multicomputers Using Asynchronous Coordinated Checkpointing,’’ Computer Science Department Technical Report CSD-910066, University of California, Los Angeles, CA (September 1991).

[Tami92] Y. Tamir and G. Janakiraman, ‘‘Hierarchical Coherency Management for Shared Virtual Memory Multicomputers,’’ *Journal of Parallel and Distributed Computing* **15**(4) pp. 408-419 (August 1992).

[Tam90] V.-O. Tam and M. Hsu, ‘‘Fast Recovery in Distributed Shared Virtual Memory Systems,’’ *10th IEEE International Conference on Distributed Computing Systems*, (1990).

[Tang76] C. K. Tang, ‘‘Cache Design in the Tightly Coupled Multiprocessor System,’’ *AFIPS Conference Proceedings, National Computer Conference*, pp. 749-753 (June 1976).

[Thak88] S. Thakkar, P. Gifford, and G. Fielland, ‘‘The Balance Multiprocessor System,’’ *IEEE Micro* **8**(1) pp. 57-69 (February 1988).

[Thap90] M. Thapar and B. Delagi, ‘‘Stanford Distributed Directory Protocol,’’ *Computer* **23**(6) pp. 78-80 (June 1990).

[Wils87] A. W. Wilson, ‘‘Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors,’’ *14th Symposium on Computer Architecture*, pp. 244-252 (June 1987).

[Woo95] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, ‘‘The SPLASH-2 Programs: Characterization and Methodological Considerations,’’ *22nd Symposium on Computer Architecture*, pp. 24-36 (1995).

[Wu90a] K.-L. Wu and W. K. Fuchs, ‘‘Recoverable Distributed Shared Virtual Memory,’’ *IEEE Transactions on Computers* **39**(4) pp. 460-469 (April 1990).

[Wu90b] K.-L. Wu, W. K. Fuchs, and J. H. Patel, ‘‘Error Recovery in Shared Memory Multiprocessors Using Private Caches,’’ *IEEE Transactions on Parallel and Distributed Systems* **1**(2) pp. 231-240 (April 1990).