Designing Efficient Communication Subsystems for Distributed Shared Memory (DSM) Systems

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Donglai Dai, B.S., M.S.

* * * * *

The Ohio State University

1999

Dissertation Committee:

Prof. Dhabaleswar K. Panda, Adviser

Prof. Ponnuswamy Sadayappan

Prof. Wu-chi Feng

Approved by

Adviser

Department of Computer and Information Science

ABSTRACT

Programmability has proved to be the biggest obstacle to ubiquitous use of scalable high performance computing systems. The emerging distributed shared memory (DSM) systems, broadly classified into hardware DSM systems and software DSM systems, provide good programmability and scalability. The key to materializing the potential of DSM systems is to ensure low latency for various remote memory and synchronization operations. In this thesis we make four important contributions towards building efficient communication subsystems for DSM systems. First, we categorize various types of network contention and evaluate their impact on the performance of DSM systems. We show that network contention can affect DSM system performance significantly. Next, we develop a parameterized analytical model for estimating the performance of a DSM system by characterizing its key components and their interactions. This model provides system architects with a fast and economical method for identifying the bottlenecks in existing or future DSM systems. Based on this model and detailed simulations, a set of network design guidelines are established. Third, we propose two new designs to improve the efficiency of node-network interfaces (NNI): (i) a pipelined NNI supporting cut-through delivery and partial cache-filling and (ii) a novel block correlated FIFO strategy and its implementation exploiting multiple paths in interconnects. These designs can significantly reduce remote memory access latencies and the complexity of NNI. Finally, we propose two kinds of architectural enhancements to the networks in DSM systems: (i) multidestination messaging mechanisms for reducing invalidation overhead for full-map cache coherence schemes and limited directory schemes, and (ii) unbalanced network designs exploiting the different characteristics of request and reply traffic. The effectiveness of these new designs and enhancements has been evaluated extensively using a simulation-based testbed and benchmark applications. The experimental results demonstrate that the overall performance of current and future DSM systems can be improved significantly by using these novel designs and enhancements in the communication subsystems.

Designing Efficient Communication Subsystems for Distributed Shared Memory (DSM) Systems

By

Donglai Dai, Ph.D.

The Ohio State University, 1999

Prof. Dhabaleswar K. Panda, Adviser

Programmability has proved to be the biggest obstacle to ubiquitous use of scalable high performance computing systems. The emerging distributed shared memory (DSM) systems, broadly classified into hardware DSM systems and software DSM systems, provide good programmability and scalability. The key to materializing the potential of DSM systems is to ensure low latency for various remote memory and synchronization operations. In this thesis we make four important contributions towards building efficient communication subsystems for DSM systems. First, we categorize various types of network contention and evaluate their impact on the performance of DSM systems. We show that network contention can affect DSM system performance significantly. Next, we develop a parameterized analytical model for estimating the performance of a DSM system by characterizing its key components and their interactions. This model provides system architects with a fast and economical method for identifying the bottlenecks in existing or future DSM systems. Based on this model and detailed simulations, a set of network design guidelines are established. Third, we propose two new designs to improve the efficiency of node-network interfaces (NNI): (i) a pipelined NNI supporting cut-through delivery and partial cache-filling and (ii) a novel block correlated FIFO strategy and its implementation exploiting multiple paths in interconnects. These designs can significantly reduce remote memory access latencies and the complexity of NNI. Finally, we propose two kinds of architectural enhancements to the networks in DSM systems: (i) multidestination messaging mechanisms for reducing invalidation overhead for full-map cache coherence schemes and limited directory schemes, and (ii) unbalanced network designs exploiting the different characteristics of request and reply traffic. The effectiveness of these new designs and enhancements has been evaluated extensively using a simulation-based testbed and benchmark applications. The experimental results demonstrate that the overall performance of current and future DSM systems can be improved significantly by using these novel designs and enhancements in the communication subsystems.

Designing Efficient Communication Subsystems for Distributed Shared Memory (DSM) Systems

By

Donglai Dai, Ph.D.

The Ohio State University, 1999

Prof. Dhabaleswar K. Panda, Adviser

Programmability has proved to be the biggest obstacle to ubiquitous use of scalable high performance computing systems. The emerging distributed shared memory (DSM) systems, broadly classified into hardware DSM systems and software DSM systems, provide good programmability and scalability. The key to materializing the potential of DSM systems is to ensure low latency for various remote memory and synchronization operations. In this thesis we make four important contributions towards building efficient communication subsystems for DSM systems. First, we categorize various types of network contention and evaluate their impact on the performance of DSM systems. We show that network contention can affect DSM system performance significantly. Next, we develop a parameterized analytical model for estimating the performance of a DSM system by characterizing its key components and their interactions. This model provides system architects with a fast and economical method for identifying the bottlenecks in existing or future DSM systems. Based on this model and detailed simulations, a set of network design guidelines are established. Third, we propose two new designs to improve the efficiency of node-network interfaces (NNI): (i) a pipelined NNI supporting cut-through delivery and partial cache-filling and (ii) a novel block correlated FIFO strategy and its implementation exploiting multiple paths in interconnects. These designs can significantly reduce remote memory access latencies and the complexity of NNI. Finally, we propose two kinds of architectural enhancements to the networks in DSM systems: (i) multidestination messaging mechanisms for reducing invalidation overhead for full-map cache coherence schemes and limited directory schemes, and (ii) unbalanced network designs exploiting the different characteristics of request and reply traffic. The effectiveness of these new designs and enhancements has been evaluated extensively using a simulation-based testbed and benchmark applications. The experimental results demonstrate that the overall performance of current and future DSM systems can be improved significantly by using these novel designs and enhancements in the communication subsystems.

Dedicated to my parents Yuan Zhong Dai and Hai Yan Huang, for their love and faith

ACKNOWLEDGMENTS

I would like thank my advisor, Prof. D. K. Panda for his help, guidance, friendship over the last four years. He has spent an immense amount of time and energy in developing and polishing my research, writing, and presentation skills. His patience and dedication have transformed me from the diffident graduate student I was to what I am now.

I would like to thank my mother and father for their unquestioned love, and faith, and guidance over the years. It is impossible to do justice to the encouragement that they have given me to continue my education. I thank my sister for her love and encouragement. I thank my wife Wendy for her love, faith, and support. I thank my daughter Allison for making my life so joyful.

I would like to thank current and previous members of the PAC group, Prof. P. Sadayappan, Prof. T. Page, Prof. W.-C. Feng, and other members of the NOW group for the valuable suggestions, feedback, and help. I thank Profs. Mike M.-T. Liu, Steve T.-H. Lai, Phil Krueger, C.-H. Huang, Anish Arora, Neelam Soundarajan, and D. N. Jayasimha for inspiring discussions, help, and friendship. I thank Debashis Basak for invaluable advice and help during the initial years of my Ph.D. I thank Rajeev Sivaram of IBM for help with Chapter 2, Mike Galles of SGI for help with Chapter 4, and Prof. Jose Duato of Politec. de Valencia Univ. for inspiration on Chapter 5. I thank the great friends I have made at Ohio State: Debashis Basak, Rajeev Sivaram, Ram Kesavan, Sri Subramanian, Mohammad Banikazemi, Scott King, Matt Jacunski, Darius Buntinas, Pete Ware, Vijay Moorthy, Sandeep Gupta, Shiv Kaushik, Gopal Dommety, Sandeep Kulkarni, Ravi Prakash, Mangesh Ghiware, Jayanthi Sampathkumar, Sandeep Prabhu, Rohit Goyal, Manoj Pillai, Steve and Marika Fridella, Bobby Vandalore, Pradeep Chordia, Po-Wen Shih, Chao-Hui Wu, Jun Xu, Junfeng He, Ming Liu, Cho-yu Chiang, and Min-te Sun. I thank my teachers and friends from high school, Xian Jiaotong University, and Florida Atlantic University.

I would like to thank Drs. Craig B. Stunkel, Ashwini Nanda, Bulent Abali, Micheal Maged, Micky Tsao, Doug Joseph, and Micheal Rosenfield for their help and friendship during my summer internship at IBM T. J. Watson Research Center. I would like to thank the National Science Foundation for providing funds for my advisor to hire me as a Research Associate, and thank the Department of Computer and Information Science and The Ohio State University for financial support and a great environment during my stay here.

Last, but most importantly, I thank God for leading me through all the highs and lows during my PhD study and life in general.

VITA

June 5th, 1963	Born - Wenzhou, Zhejiang, China
July 1985	B.S., Computer Science and Engg., Xian Jiaotong University.
Fall 1985 - Spring 1988	University Graduate Fellow, Xian Jiaotong University.
June 1988	M.S., Computer Science and Engg., Xian Jiaotong University.
June 1988 - April 1990	Instructor and Research Staff Member, Xian Jiaotong University.
Summer 1990 - Spring 1991	Graduate Teaching Assistant, Florida Atlantic University.
June 1991 - January 1993	Testing Programmer, IBM Boca Raton Division.
Fall 1993 - Winter 1994	Graduate Research Associate, The Ohio State University.
March 1994	M.S., Computer and Info. Science, The Ohio State University.
Spring 1994 - Winter 1999	Graduate Teaching/Research Associate, The Ohio State University.
Summer 1996	Research Intern, IBM T.J.Watson Research Center.

PUBLICATIONS

Research Publications

D. Dai and D. K. Panda. "Exploiting the Benefits of Multiple-Path Network in DSM Systems: Architectural Alternatives and Performance Evaluation." *IEEE Transactions on Computers*, Special Issue on Cache Memory and Related Problems, pp. 236-244, Vol. 48, No. 2, February 1999.

F. Silla, M. P. Malumbres, J. Duato, D. Dai, and D. K. Panda. "Impact of Adaptivity on the Behavior of Networks of Workstations under Bursty Traffic." *Proceedings of the 27th International Conference for Parallel Processing*, pp. 88-95, August 1998.

D. Dai and D. K. Panda. "Evaluating Pipelined Node-Network Interface Designs for DSM Systems." *Technical Report OSU-CISRC-8/98-TR36*, The Ohio State University, August 1998.

D. Dai and D. K. Panda. "How Much Does Network Contention Affect Distributed Shared Memory Performance?" *Proceedings of the 26th International Conference for Parallel Processing*, pp. 454-461, August 1997.

D. Dai and D. K. Panda. "How Can We Design Better Networks for DSM Systems?" *Proceedings of 2nd Parallel Computing Routing and Communication Workshop, Lecture Notes in Computer Science (LNCS) #1417*, pp. 171-184, June 1997, Springer Verlag.

D. K. Panda, D. Basak, D. Dai, R. Kesavan, R. Sivaram, M. Banikazemi, and V. Moorthy. "Simulation of Modern Parallel Systems: A CSIM-Based Approach." *Proceedings of Winter Simulation Conference*, pp. 1013-1020, December 1997.

D. Dai and D. K. Panda. " Effective Use of Virtual Channels in Wormhole Routed Distributed Shared Memory Systems." *Technical Report OSU-CISRC-10/97-TR46*, The Ohio State University, October 1997.

D. Dai and D. K. Panda. "Reducing Cache Invalidation Overheads in Wormhole DSMs using Multidestination Message Passing." *Proceedings of the 25th International Conference for Parallel Processing*, pp. I:138-145, August 1996.

D. Dai and D. K. Panda. " Efficient Schemes for Limited Directory-based DSMs Using Multidestination Message Passing." *Technical Report OSU-CISRC-11/96-TR61*, The Ohio State University, November 1996.

J. Wu, E. B. Fernandez, and D. Dai. "Optimal Fault-Secure Scheduling." *ISCA International Conference on Parallel and Distributed Computing*, pp. 603-608, 1995.

D. Dai, S. K. S. Gupta, S. D. Kaushik, J. H. Lu, R. V. Singh, C.-H. Huang, P. Sadayappan, and R. W. Johnson. "EXTENT: A Portable Programming Environment for Designing and Implementing High-Performance Block Recursive Algorithms." *Proceedings of Supercomputing'94*, pp. 49-58, 1994.

D. Dai, S. Q. Zheng, D. G. Shan, D. P. Qian, B. Jia, and Y. L. Zhao. "Investigation on Implementation of Evaluator on Lisp Machine Lisp-M1." *Journal of Xian Jiaotong University*, pp. 31-36, Vol. 22, No. 6, June 1988.

S. Q. Zheng, B. Jia, D. G. Shan, D. P. Qian, Y. L. Zhao, and D. Dai. "AI Computer System LISP-M1." *Journal of Xian Jiaotong University*, pp. 1-8, Vol. 22, No. 6, June 1988.

D. P. Qian, D. G. Shan, S. Q. Zheng, B. Jia, D. Dai, and Y. L. Zhao. "Design and Implementation of a List Processor in Lisp Machine Lisp-M1." *Journal of Xian Jiaotong University*, pp. 15-22, Vol. 22, No. 6, June 1988.

D. G. Shan, S. Q. Zheng, B. Jia, D. P. Qian, Y. L. Zhao, and D. Dai. "Memory Hierarchical Structure in AI Computer Lisp-M1." *Journal of Xian Jiaotong University*, pp. 9-14, Vol 22, No. 6, June 1988.

Y. L. Zhao, S. Q. Zheng, D. G. Shan, D. P. Qian, B. Jia, and D. Dai. "Compact List Representation and Its Implementation on Lisp Machine Lisp-M1." *Journal of Xian Jiaotong University*, pp. 23-30, Vol. 22, No. 6, June 1988.

FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in:

Computer Architecture	Prof. Dhabaleswar K. Panda
Algorithm Theory	Prof. Ten-Hwang Lai
Operating Systems	Prof. Thomas W. Page Jr.

TABLE OF CONTENTS

Page

Abstract ii

Dedication iii

Acknowledgments iv

Vita v

List of Tables xiii

List of Figures xv

Chapters:

1. Introduction 1

    1.1 Scalable Parallel Computing Systems 1
    1.2 Communication Subsystems in Emerging DSM Systems 3
        1.2.1 Interconnection Layer 5
        1.2.2 Protocol Layer 7
    1.3 Problem Description 10
        1.3.1 Understanding the Impact of Network Contention and Developing a Comprehensive Analytical Model for DSM Performance 10
        1.3.2 Support for Efficient Node-Network Interface 11
        1.3.3 Network Support for the Protocol Layer 12
    1.4 Thesis Overview 13
        1.4.1 Assumptions 13
        1.4.2 Experimental Methodology 14
        1.4.3 Overview of the Proposed Solutions 16

2. Impact of Network Contention on DSM Performance 19

2.1 Message Transfer and Network Contention in DSMs 19
2.2 Methodology for Modeling Network Contention 21
2.3 Simulation Environment 22
2.3.1 The Simulated Architecture 22
2.3.2 Implementing Network Models 25
2.4 Simulation Results and Discussion 26
2.4.1 Impact of Network and Network Contention in Baseline System 27
2.4.2 Impact of Cache Organization 27
2.4.3 Impact of Memory System 30
2.4.4 Effect of Node Speed 31
2.4.5 Impact of Network Parameters 32
2.5 Related Work 34
2.6 Summary 34

3. Comprehensive Analytical Model for DSM Performance 36

3.1 Analytical Model for a Typical DSM System 36
3.1.1 Characterizing Application Behavior on a Processor 37
3.1.2 Characterizing Cache and Memory Hierarchy 39
3.1.3 Characterizing Interconnection Network 40
3.1.4 Putting All Components Together 40
3.2 Impact of Network Components 42
3.2.1 Impact of Link Speed 43
3.2.2 Impact of Link Width 44
3.2.3 Impact of Topology 44
3.3 Simulation Results 45
3.3.1 Simulation Environment 45
3.3.2 Performance Metrics 45
3.3.3 Impact of Network Speed 46
3.3.4 Impact of Link Width 48
3.3.5 Impact of Topology 48
3.4 Discrepancy Between Analytical and Simulation Models 51
3.5 Summary 51

4. Pipelined Node-Network Interface Designs 52

4.1 System Model 53
4.2 Role of NNIs in Remote Memory Operations 54
4.3 Conventional Block-Based NNI Design 56
4.4 Pipelined NNI Designs 57
4.4.1 NNI Supporting Partial Cache-Filling 57

4.4.2 NNI Supporting Cut-Through Delivery 57
4.4.3 Integrating CTD and PCF 59
4.5 Implementation of BTU 60
4.6 Performance Evaluation 62
4.6.1 Simulation Environment 62
4.6.2 Latency of Remote Memory Operations 63
4.6.3 Application Performance 65
4.7 Related Work 69
4.8 Summary 70

5. Incorporating Multiple-Path Networks by Using Efficient Node-Network Interface 71

5.1 Relevant Design Issues 72
5.2 Intelligent Cache Coherence Protocol Strategy 73
5.3 Total FIFO Channel Strategy 74
5.4 Block Correlated FIFO Channel Strategy 75
5.4.1 Eliminating Race Conditions 75
5.4.2 Block Correlated FIFO Channel 76
5.4.3 The Strategy and Implementation 76
5.5 Performance Evaluation 76
5.5.1 Methodology 76
5.5.2 Results and Discussions 79
5.5.3 Impact of Cache Design Parameters 82
5.5.4 Impact of Network Topology 83
5.6 Summary 85

6. Reducing Invalidation Overhead in Full-map Coherence Schemes 87

6.1 Cache Invalidation Overhead in DSM systems 88
6.1.1 Characteristics of Invalidations 88
6.1.2 Latency and Traffic Estimates 89
6.2 Multidestination Message Passing 89
6.2.1 The BRCP Model 90
6.2.2 Multicast and Gather Worms 90
6.3 Frameworks for Reducing Overhead 90
6.3.1 MI-UA Framework 91
6.3.2 Three Novel Mechanisms 92
6.3.3 MR-MA Framework 93
6.4 Grouping Schemes 94
6.4.1 Associated Issues 94
6.4.2 Dimensional Order Routing in 2D Mesh 98
6.4.3 Turn Model Adaptive Routing in 2D Mesh 99
6.4.4 Comparison of Grouping Schemes 100

6.4.5 Systems with Other Topologies . . . 100
6.4.6 Plane-Based Two-step Reservation Scheme . . . 103
6.5 Simulation Experiments and Results . . . 104
6.5.1 Simulation Setup . . . 104
6.5.2 Results and Discussions . . . 105
6.6 Summary . . . 107

7. Reducing Invalidation Overhead in Limited Directory Schemes . . . 108

7.1 Limited Directory Schemes . . . 109
7.1.1 Motivation Behind Limited Directories . . . 109
7.1.2 Broadcast Scheme (dir_iB) . . . 110
7.1.3 Coarse Vector Scheme (dir_iCV_r) . . . 110
7.2 Limited Directory Schemes with Multidestination Messages . . . 111
7.2.1 Enhanced Broadcast Scheme . . . 111
7.2.2 Enhanced Coarse Vector Schemes . . . 113
7.2.3 Overall Comparison . . . 116
7.2.4 Systems with Other Topologies . . . 117
7.3 Performance Evaluation . . . 119
7.3.1 Objectives and Simulation Experiments . . . 119
7.3.2 Simulation Results . . . 120
7.4 Related Work . . . 128
7.5 Summary . . . 133

8. Designing Unbalanced Request and Reply Networks . . . 135

8.1 Limitations of Current Approach . . . 136
8.1.1 Unbalanced Usage of Request and Reply Networks . . . 136
8.1.2 Bottleneck at the Network Interface . . . 139
8.2 Proposed Solutions . . . 140
8.2.1 Unbalanced Partitioning of Virtual Channels . . . 140
8.2.2 Supporting Virtual Injection/Consumption Channels at the NI . . . 141
8.3 Simulation Experiments and Results . . . 141
8.3.1 Simulation Experiments . . . 141
8.3.2 Simulation Results . . . 143
8.4 Related Work . . . 150
8.5 Summary . . . 152

9. Conclusions and Future Research Directions . . . 154

9.1 Summary of Research Contributions . . . 154
9.2 Suggestions for Future Research . . . 155

Bibliography . . . 157

LIST OF TABLES

Table **Page**

1.1 Communication characteristics of benchmark applications. 16

2.1 Default sizes of various queues in a simulated node controller. 23

2.2 Default system parameters used in the simulation. 24

2.3 Contention-free latencies of typical memory accesses. The remote nodes are assumed to be neighboring nodes. All latency values are given in processor cycles (5ns per cycle). The fourth column (with title "# Inv or CpyBack") shows the number of invalidation or copy-back messages needed to be sent, as dictated by the cache coherence protocol being used [90, 24]. 26

2.4 Latency break down of a clean read-miss to a neighboring node in processor cycles (5ns per cycle). 27

3.1 Summary of the main parameters and notations used in the analytical performance model for DSM systems. 41

3.2 Predicted performance of three different application types under different network design choices using Eqns. 3.16-3.20. Note $T_s = T_{snet}$, $T_l = T_{lnet}$, pc = processor cycles, nc = network cycles. 43

3.3 Default system parameters used in the simulation. 46

4.1 Breakdown of the contentionless latency of a clean read-miss to a neighboring home node. Values are counts in 500 MHz processor cycles. CBB = CBB-NNI, PCF = PCF-NNI, CTD = CDT-NNI, and ICP = ICP-NNI. 64

5.1 Default system parameters used in the simulation. 78

5.2 Total number of network transactions (in thousands). 80

5.3 Summary of out-of-order (OoO) messages per pair of processing nodes in a total FIFO channel system (T-FIFO-1 configuration). 82

6.1 Comparison of different grouping schemes. 100

6.2 Default System parameters used in the simulation. 104

7.1 Directory memory usage for different schemes. 121

7.2 Default system parameters used in the simulation. 122

8.1 The characteristics of traffic generated by each of the four generic categories of cache coherence messages in the request network and the reply network. 137

8.2 Default system parameters used in the simulation. 142

LIST OF FIGURES

Figure Page

1.1 Schematic representation of a distributed memory (DM) system. 2

1.2 Schematic representation of a distributed shared memory (DSM) system. 3

1.3 Factors relevant to communication in a DSM system. The shaded area represents the communication subsystem. 4

1.4 Performance impact of the ratio of the remote memory latency to the local memory latency. 5

1.5 A generic CC-NUMA architecture. 14

1.6 Execution-driven DSM simulation framework used in this thesis. 15

2.1 Various types of network contention in a CC-NUMA system. 20

2.2 Differences and relations among the network models. 21

2.3 The CC-NUMA architecture simulated. 23

2.4 Overall execution time and its breakdown on the baseline system for three different network models. 28

2.5 Effect of cache size per node (32/64/128/256 KB) on network contention. Every system configuration has three normalized execution times (from left to right) corresponding to the NCM, NIM, and DNM model, respectively. The breakdown of the execution time is similar to the ones used in Fig. 2.4. 29

2.6 Effect of cache line size (16/32/64/128 bytes) on network contention. 29

2.7 Effect of cache associativity (1/2/4-way) on network contention. 30

2.8 Effect of memory response time (20/30/40 processor clock cycles) on network contention. 31

2.9 Effect of memory bus width (4/8/16 bytes) on network contention. 32

2.10 Effect of node speed (100/200/400 MHz) on network contention. 33

2.11 Effect of network speed (100/200/400 MHz) on network contention. 33

2.12 Effect of flit width (8/16/32 bits) on network contention. 34

3.1 A programmer's point of view to the execution behavior of a typical DSM application. 37

3.2 A processor's point of view to the execution behavior of a typical DSM application. . 38

3.3 Impact of network speed on the overall execution times of benchmark applications and on the message latencies. 47

3.4 Impact of link width on the overall execution times of benchmark applications and on the message latencies. 49

3.5 Impact of network topology on the overall execution times of benchmark applications and on the message latencies. 50

4.1 The basic structure of a node controller. 54

4.2 The timeline of a read miss to a clean remote memory block using the CBB-NNI design of node-network interface. 56

4.3 The timeline of a read miss to a clean remote memory block using pipelined node-network interfaces (NNIs): (a) PCF-NNI, (b) CTD-NNI, and (c) ICP-NNI. 58

4.4 The organization of the block transfer unit (BTU). 61

4.5 The association between the data FIFO and the descriptor table in the BTU. 61

4.6 The performance of the four alternative node-network interface (NNI) designs for benchmark applications. CBB = CBB-NNI, PCF = PCF-NNI, CTD = CDT-NNI, and ICP = ICP-NNI. 65

4.7 Sensitivity to the size of sub-block. The block size is fixed to 256 bytes. The sub-block size changes from 32 to 64 and to 128 bytes. 67

4.8 Sensitivity to the size of block. The sub-block size is fixed to 32 bytes. The block size changes from 64 to 128 and to 256 bytes. 68

4.9 Sensitivity to the occupancy on the coherence controller. The occupancy changes from 3 to 6 and to 12 bus cycles. 68

4.10 Sensitivity to cluster size. The number of compute processors connected to a single switch changes from 1 to 2 and to 4. 69

5.1 An example of out-of-order (OoO) arrivals of messages between a source and destination pair. 72

5.2 An example race condition caused by an OoO event. 73

5.3 The networks between two nodes in a system using: (a) the I-CC, (b) the T-FIFO, and (c) the C-FIFO strategies. 74

5.4 Pseudo-code description of the NI used in the total FIFO channel strategy. 'Dispatch' means delivering 'msg' to the cache coherence controller. The comments show the time delays for reordering operations.. 75

5.5 A qualitative comparison of alternative strategies for incorporating multiple-path networks in DSM systems along three fundamental design axes. The 'Use of Multiple Paths' axis indicates the flexibility for transferring a given message. 77

5.6 The overall execution times of benchmark applications on different configurations. . 80

5.7 The average latency of a network transaction on different configurations. 81

5.8 Impact of smaller cache line (128 bytes) on the overall execution times of benchmark applications on three configurations. 83

5.9 Impact of larger cache size per node (256K) on the overall execution times of benchmark applications on three configurations. 84

5.10 Impact of larger cluster size (4 processing nodes per switch) on the overall execution times of benchmark applications on three configurations. 85

5.11 The overall execution times of benchmark applications on three configurations. Switches are connected using a 3D mesh topology. 86

5.12 The overall execution times of benchmark applications on three configurations. Switches are connected using a 2D mesh topology. 86

6.1 Example of an invalidation in an dimensional order routed mesh DSM supporting unicast communication: (a) the request phase and (b) the acknowledgment phase. . 88

6.2 Feasible paths for multidestination worms in a 2D mesh [112]. 90

6.3 Comparing the communication traffic for an invalidation transaction in two frameworks: (a) UI-UA and (b) MI-UA. 91

6.4 Enhanced node organization and the associated router interface with the i-ack buffer. 92

6.5 An example of a portion of invalidation using the MR-MA framework. 94

6.6 The routing algorithm for a reservation worm. 95

6.7 The routing algorithm for a gather worm. 96

6.8 Invalidation handler algorithm at the NNI. 97

6.9 Invalidation acknowledgment handler algorithm at the NNI. 97

6.10 Dimensional order routing based grouping schemes. 98

6.11 Turn model routing grouping schemes. 99

6.12 Axis-based one-step reservation (AB1R) scheme in a 3D mesh supporting dimensional order routing using full-map directory. The two phases of an invalidation transaction: (a) request and (b) acknowledgment. 101

6.13 Axis-based two-step reservation (AB2R) scheme in a 3D mesh supporting *all-but-X-negative-first turn model* routing using full-map directory. The two phases of an invalidation transaction: (a) request and (b) acknowledgment. 102

6.14 Plane-based two-step reservation (PB2R) scheme in a 3D mesh supporting all-but-X-negative-first turn model routing using full-map directory. The two phases of an invalidation transaction: (a) request and (b) acknowledgment. 103

6.15 Comparing the number of invalidation messages required under the unicast-based and multidestination-based schemes. 105

6.16 Comparing the total number of messages. 106

6.17 Comparing the overall execution time. 106

6.18 The impact of i-ack buffer size in Barnes application. 106

7.1 An example of invalidation broadcast using the m_dir_iB scheme. 112

7.2 Examples of different region partitioning schemes under multidestination message passing for an 8×8 mesh. 114

7.3 Illustration of an example invalidation event using the $m\_dir_iCB_1$ scheme on an 8×8 mesh. The home node and the sharers are assumed to be the same as shown in Fig. 7.1.115

7.4 Illustration of an example invalidation event using the $m\_dir_iCB_2$ scheme on an 8×8 mesh. The home node and the sharers are assumed to be the same as shown in Fig. 7.1.116

7.5 Axis-region one-step reservation (AR1R) scheme in a 3D mesh supporting e-cube routing using limited directory. The two phases of an invalidation transaction: (a) request and (b) acknowledgment. 117

7.6 Axis-region two-step reservation (AR2R) scheme in a 3D mesh supporting all-but-X-negative-first turn model routing using limited directory. The two phases of an invalidation transaction: (a) request and (b) acknowledgment. 118

7.7 Plane-region two-step reservation (PR2R) scheme in a 3D mesh supporting all-but-X-negative-first turn model routing using limited directory. The two phases of an invalidation transaction: (a) request and (b) acknowledgment. 119

7.8 Performance comparison of different schemes on a 64 processor system using Barnes. 124

7.9 Performance comparison of different schemes on a 32 processor system using Barnes. 125

7.10 Performance comparison of different schemes on a 64 processor system using LU. . . 126

7.11 Performance comparison of different schemes on a 32 processor system using LU. . . 127

7.12 Performance comparison of different schemes on a 64 processor system using Radix. 129

7.13 Performance comparison of different schemes on a 32 processor system using Radix. 130

7.14 Performance comparison of different schemes on a 64 processor system using Water. 131

7.15 Performance comparison of different schemes on a 32 processor system using Water. 132

8.1 Comparing the theoretical latencies of messages in the request and reply networks by assuming these two networks being physically separated. 139

8.2 An example snapshot of a typical router and the network interface supporting one physical injection/consumption channel. There are three concurrently bypassing worms in the router. One worm is being injected from the NI to the router into the network. Two worms are terminating at this router. 140

8.3 The effect of virtual injection and virtual consumption channels on the overall execution time of benchmark applications. 144

8.4 The effect of virtual injection and virtual consumption channels on the average message latencies in the request and reply networks. 144

8.5 The effect of virtual channel partitioning on the overall execution time. 145

8.6 The effect of virtual channel partitioning on the average message latencies in the request and reply networks. 146

8.7 Changing trend of the effect of virtual channels on the overall execution times under different cache line sizes. 148

8.8 Changing trend of the effect of virtual channels on the average message latencies under different cache line sizes. 149

8.9 Impact of switch routing delay on the overall execution times. 149

8.10 Impact of switch routing delay on the average message latencies. 150

8.11 Impact of network topology on the overall execution times. 151

8.12 Impact of network topology on the average message latencies. 152

CHAPTER 1

INTRODUCTION

1.1 Scalable Parallel Computing Systems

The endless quest for more computing power from a variety of application domains, such as advanced scientific computing, engineering computing, commercial computing, etc., has driven the computer industry to build ever-increasingly powerful computing systems. Scientific computing applications typically involve modeling and simulating physical phenomena, in sufficient detail, that are either too complex to use analytical methods or impossible/too costly to use measuremental methods. Examples of scientific computing applications include modeling global weather, molecular dynamics, fluid dynamics, etc. Engineering computing applications typically involve computer-aided-design (CAD) and computer-aided-manufacturing (CAM). Examples of engineering computing applications include design automation, industrial process control, damage analysis on automobile crashes, etc. Commercial computing applications typically involve on-line transaction processing to enormous databases. Examples of commercial computing applications include enterprise decision supporting, data mining, servicing web accesses, maintaining on-line reservation systems, etc.

These applications demand computing power which is orders of magnitude higher than the capability of any single modern processor system. Therefore, parallel computing systems with multiple processors (or processing nodes) become the natural choice to meet such a demand. Examples of such systems include the Thinking Machines CM-5 [89, 137], Meiko CS-2 [18, 98], Intel Paragon [68], nCube/3 [46], NEC Cenju [79], Cray T3D/T3E [124], IBM SP [4], SGI Origin [88], HP SPP [22], Sequent NUMA-Q [94], and HAL Mercury [144].

The fundamental issue in designing scalable parallel computing systems, also being known as *massively parallel processing* (MPP) systems, is to achieve efficient communication among the cooperative processing nodes which collectively solve one application problem. This requires that the processors in such a system be efficiently interconnected with each other and with the other components of the system such as memory, disks, and I/O devices. For this purpose, various system architectures have been developed in the past. These architectures basically support two different kinds of programming paradigms, namely, message passing paradigm and shared memory paradigm. In the message passing paradigm, any communication between processors must be coded explicitly in the application programs. In the shared memory paradigm, communication between processors is achieved by accesses to shared variables in the programs.

Scalable parallel computing systems broadly fall into two categories: distributed memory systems and distributed shared memory systems. *Distributed memory* (DM) systems consist of a set of processors, each connected to its local memory and attached disks or I/O peripherals, and connected to each other through an interconnection network. Figure 1.1 shows a schematic DM architecture. DM systems support the message passing programming paradigm. Application programmers are required to partition and distribute the target tasks and data among the processing nodes of a system. The advantages of such systems are the simplicity of design/implementation and the inherent scalability. The severe disadvantage is the burden imposed on the programmers to partition the problem onto multiple processors. The Thinking Machines CM-5, Meiko CS-2, Intel Paragon, nCube/3, NEC Cenju, and IBM SP are examples of DM systems.

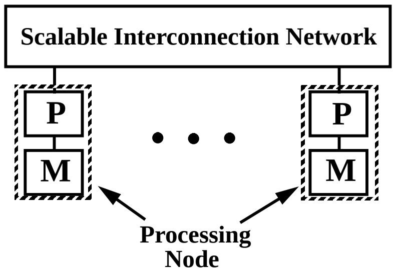


Figure 1.1: Schematic representation of a distributed memory (DM) system.

Recently, to alleviate the programmer's burden, *distributed-shared memory* (DSM) systems are emerging as the trend in building modern scalable parallel computing systems [73]. DSM systems support the shared memory programming paradigm while maintaining good scalability. In particular, memory is physically distributed among processing nodes, and a collection of the memory on each node forms a global address space accessible by all processors, as shown in Fig. 1.2. Memory accesses from a processor to a remote location is fulfilled by the communication assistant at each node and the underlying network. For a large variety of applications, the conversion from sequential programs to their parallel equivalents is much easier on DSM systems than on DM systems. Such improved programmability is the obvious advantage of DSM systems. On the other hand, the disadvantage is the complexity of the communication assistant required for providing the global shared memory on top of the physically distributed memories. Examples of such systems are the Stanford DASH/FLASH [91, 92, 59], MIT Alewife [1], Cray T3D/T3E, SGI Origin, HP SPP, Sequent NUMA-Q, and HAL Mercury, etc.

Most existing DSM systems employ the cache coherent *Non-Uniform Memory Access* (CC-NUMA) architecture. The naming is due to two facts: 1) the disparity in access times between local and remote memory and 2) system-wise coherence enforced among the processor's caches. CC-NUMA systems automatically replicate remote data at the caches when needed, and use a directory-based, write-invalidate cache coherence protocol. The protocol ensures that accesses to shared data by any processor always get the latest copy which the processor is aware of. Except the Cray T3D/T3E, all example DSM systems listed above use the CC-NUMA architecture.

More recently, as workstations and personal computers (PCs) are becoming more powerful, software DSM systems (also called *virtual shared memory* (VSM) systems) running on a cluster of workstations [6] are emerging as inexpensive high performance computing systems. Unlike the hardware CC-NUMA systems where the coherence is maintained at the cache line level, software DSM systems normally maintain coherence at the page level by exploiting the standard virtual

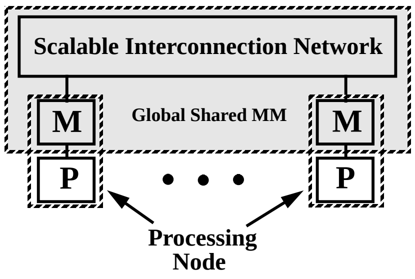


Figure 1.2: Schematic representation of a distributed shared memory (DSM) system.

memory management mechanism of the workstations. Due to the volume production, modern commercial workstations and high performance switches like the Autonet [122], Myrinet [19], and ServerNet [50, 63], offer a much better price-to-performance ratio than the hardware DSM systems. However, current generation software DSM systems like the TreadMarks [5], SHRIMP [93, 48] (with limited special hardware support), Shasta [121], etc., only show competitive performance on small scale systems with four to eight nodes. For a larger scale system, the overall performance is limited by severe load imbalance induced by false sharing and the high overhead of communication and synchronization operations [67].

In summary, regardless of the distinctions between message passing and shared memory programming paradigms, the underlying parallel architecture is converging towards a collection of processing nodes interconnected by a scalable network. As processor speed and system size keep on increasing, communication between the nodes is becoming a critical factor for obtaining the maximum performance from a scalable parallel system. In this thesis, we focus on communication problems in hardware DSM systems. However, most of the methods proposed can also be applied to DM systems and software DSM systems.

1.2 Communication Subsystems in Emerging DSM Systems

As discussed above, communication in DSM systems are triggered implicitly by shared memory accesses. To fulfill the desired transformation, in addition to the interconnection network, dedicated hardware logic (i.e., the communication assistant) is needed, for transferring the resultant message. Figure 1.3 shows the main factors relevant to communication in a DSM system. These factors can be broken into three layers: the application layer, the protocol layer, and the interconnection layer. The protocol layer and the interconnection layer comprise the communication subsystem in a DSM system. Even though the application layer is not a part of the communication subsystem, it determines the frequency and volume of communication [27]. Therefore, it affects the impact of communication subsystem on the overall performance of a DSM system.

Before we examine the factors of the communication subsystem in detail, it will be helpful to know the importance of communication subsystem on the overall performance of a DSM system. An often used indicator is the impact of the ratio of the remote memory latency (with inter-node communication) to the local memory latency (without inter-node communication) on the performance [88, 144, 94, 27]. Figure 1.4 shows this impact in two representative applications

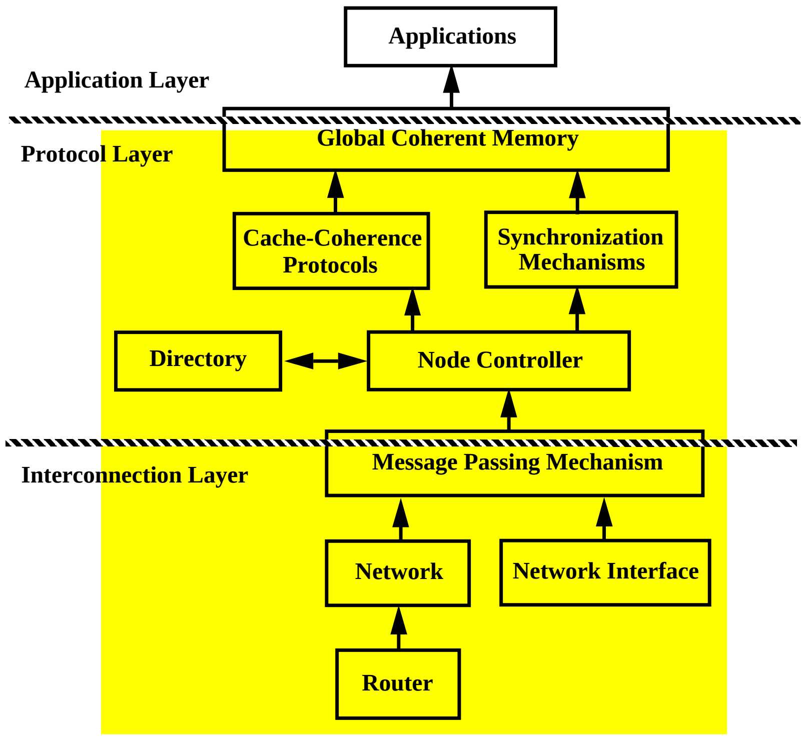


Figure 1.3: Factors relevant to communication in a DSM system. The shaded area represents the communication subsystem.

(FFT and Barnes)[1]. It can be observed that as the ratio changes from 1.5:1 to 4:1, the (normalized) execution times of applications increase more than 20%. The execution time increases significantly beyond the 4:1 ratio point. Therefore, efficient communication subsystems are very important for DSM systems. In particular, reducing remote memory access latency is crucial for the overall performance of such systems. Treating the remote memory access latency as a second order effect is a severe mistake sometimes made by the system designers, who concentrate on minimizing the local memory access latency alone. Generally, the remote memory access latency contains a minimum delay/overhead as well as contention at each of the factors of the communication subsystem shown in Fig. 1.3. In the rest of this section, we examine these factors and the associated issues in detail.

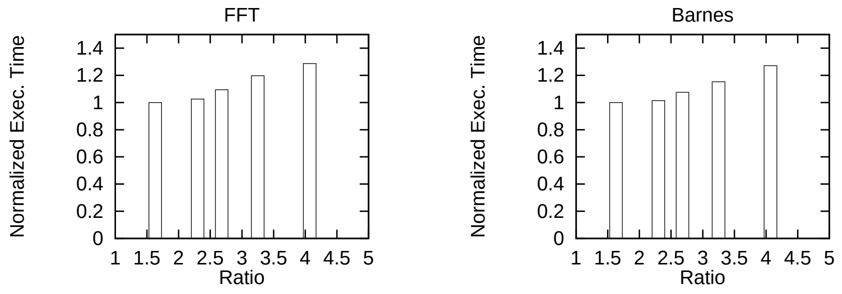


Figure 1.4: Performance impact of the ratio of the remote memory latency to the local memory latency.

1.2.1 Interconnection Layer

The interconnection layer is the bottom half of the communication subsystem in a DSM system, as shown in Fig. 1.3. This layer consists of many components such as the router/switch architecture, switching technique, routing scheme, network topology, and network interface. The following discussion describes these components and their functionalities in greater detail.

A. Router/Switch Architecture

The architecture of a generic router/switch can be described as follows. It consists of a set of input channels/ports, a set of output channels/ports, internal buffers associated with these ports, a switching fabric which can connect a subset of input ports to a subset of output ports at a given time, and a routing and arbitration unit which decides and sets the state of the switching fabric. If multiple input ports compete for the same output port, the arbitration unit assigns the output port to only one input port at a given time according to a predefined scheduling policy. In modern networks, long messages are normally partitioned into *packets*, which in turn may be broken into flow control units (*flits*[42]). Due to the limitations in channel width, a single flit transfer may require multiple physical channel cycles [44].

[1]These results are collected from realistic simulations (to be described later) using representative system and application parameters.

There are three basic organizations of buffers in switches: *input buffering*, *output buffering*, and *central shared-buffering*. The classification comes from whether the switch has the buffers associated with every input port, every output port, or one buffer shared by the input and output ports. The most popular buffering scheme is input (FIFO) buffering for implementation simplicity. Dynamic queuing (called DAMQ [136]) is a hybrid scheme of input buffering and central shared-buffering, where buffers associated with each input port are dynamically divided into logically separated sub-queues for each output port.

Two parameters associated with a router/switch are important to a designer. The *routing delay* is the time for the router to examine an incoming message and determine which output port to send the message. Once the path within the router has been established, the *switching delay* is the time required for the data to be transferred from the input port to the output port.

B. Switching Technique and Routing Scheme

The switching technique is the method used in selecting the path along which a message is transferred [113]. Switching techniques broadly fall into two categories: circuit switching and packet switching. In circuit switching, when a message needs to be sent, the path from a source to its destination must exist first; otherwise, such a circuit needs to be set up. Once a circuit is set up, the message is transferred as a continuous stream along the circuit. No links on the circuit can be used by another circuit before the former is torn down. In packet switching, a message is typically divided into smaller packets, whose maximum size is fixed to a given number of bytes. Each packet is transferred independently on a per link basis. A link is reserved and released for transferring a packet. Packet switching variants are used in most modern commercial parallel systems.

The routing scheme determines the output port of a switch that an incoming message/packet should be forwarded to reach its destination [44, 20, 41, 8]. Routing schemes can be broadly classified into two types: deterministic and adaptive. In deterministic routing, only one output port can be used to forward the incoming packet. The routing scheme determines this output port based on the destination of the packet. On the other hand, in adaptive routing, any one of the candidate output ports may be chosen. The decision is typically made based on the availability of the output ports and on other parameters like utilization of the output ports.

Adaptive routing schemes can be further divided into two classes: static and dynamic. In static schemes, a complete path from the source to the destination is decided at the time when a message is injected into the network. Different paths can be selected at different times between a pair of source and destination nodes. In dynamic schemes, the path is decided as the packet moves through the network. Static schemes require simpler switches and incur less switch delay. On the other hand, dynamic schemes are able to adjust to the temporal traffic load in the network. When virtual channels are supported, a path can be uniquely defined by a sequence of virtual channels from the source to the destination. Multiple virtual lanes [41] help to implement a simple dynamic adaptive scheme which allows a message to select one of the equivalent virtual channels over the same physical link. Adaptive routing schemes potentially allow messages along different paths to reach the destination in arbitrary order depending on many factors such as the lengths of the paths, congestion conditions along the paths, the lengths of messages, routing and arbitration algorithms used in the switches, etc. Thus, a good communication subsystem must consider all these aspects to deliver better performance to the application layer.

C. Network Topology

Network topology defines how the elements of a network like routers/switches, links, and (processing) nodes are connected to each other. The topology determines the average and maximum distance a message needs to travel to reach the destination. The distance is measured in number of hops (links) on the path from the source to destination. This distance has strong impact on communication latency because the latency and possible resource contention are typically proportional to the number of links a message has to traverse on the average, The topology also determines the number of communications that can concurrently take place in the network, as measured by the bisection bandwidth. This bandwidth is defined to be the number of links that need to be broken to divide the system into two almost equal halves (in terms of the number of nodes) [44]. Therefore, network topology can significantly affect the inter-node communication latency and the overall communication performance of a scalable parallel system.

Networks can be classified into direct network and indirect networks depending on whether every switch in the system has at least one (processing) node attached to it or not. Examples of direct networks include k-ary n-cube networks such as meshes, tori, and hypercube [40]. In the mesh topology, routers and links are arranged to form a grid, with each router being connected to one node. In a torus, wraparound connections exist between the edges of the grid. In a hypercube with 2^n pairs of router and node, each such pair is connected to n other distinct pairs. Examples of indirect networks include a variety of multistage interconnection network based architectures such as the fat tree networks, omega, butterfly, cube, and other delta networks; bidirectional MINs, and some arbitrary irregular topologies that are currently becoming popular for networks of workstations environments [44].

D. Network Interface

Each processing node in a parallel system has dedicated communication logic which interfaces the router/switch with the node. The network interface (NI) connects either the processor/memory interconnect or a specialized I/O bus on the processing node to one of the output ports and one of the input ports in the switch. The typical NI has some memory to buffer incoming and outgoing messages, a few DMA engines which allow transfer of data between the NI memory and node's memory, and link interfaces which control the actual sending and receiving of data on the bidirectional links. The delay associated with the injection of a packet into the router at the sender side is called *injection delay.* Similarly, *consumption delay* is the delay associated with the consumption of a packet from the router into the processing node at the receiver side. The injection delay involves the time to DMA the message from node's memory to NI memory, the time for the NI to prepare the packet to be sent out, and the time to DMA the packet into the network through the link interface. On the receiver side, the transfer of data from the network to the NI memory and then to the node's memory is representative of the consumption delay. This involves the time to DMA the received packet from the link interface to NI memory, time for the NI to examine the correctness of the packet and to strip the routing header, and time to DMA the packet from NI memory to node's memory.

1.2.2 Protocol Layer

The other important component in the communication subsystem is the protocol layer which consists of the directory-based cache coherence protocol, organization of the directory, and synchronization mechanisms. The node controller is the main hardware logic of this layer which enforces the cache coherence protocol, consults the contents of directories, and implements the synchronization mechanisms, as illustrated in Fig. 1.3. The following discussion describes the functionalities of the node controller in these aspects.

A. Cache Coherence Protocol and Network Transaction

As mentioned before, a CC-NUMA system provides a coherent, global address space to all processing nodes. The *home node* of a memory block is a node in whose main memory the block is allocated. When a memory operation is issued to a location and the issuing node (which is not the home node) does not have a valid copy in its private cache, a request message is sent via the network to the home node. Eventually, a response message comes back to the issuing node with a copy of the memory block containing the content of the desired location. The entire communication process of constructing, sending, receiving, and dispatching a message is commonly abstracted as a single *network transaction* [27]. Thus, a simple request-response cycle contains two network transactions. At any time, information is maintained among a set of nodes about the operating states of copies of each memory block. A node in this set either has cached the block or is the home node of the block. A cache coherence protocol is a protocol guaranteeing that a *read* to any memory location always gets a copy of the content from the latest *write* which the reading processor is aware of. In a CC-NUMA system, such a consistent view of memory is ensured on the basis of each memory block. Various cache coherence protocols differ mainly on the types of block states being used, the types of network transactions being used, and the rules for state transitions.

B. Cache-based and Memory-based System

The cache coherence protocol of a CC-NUMA system is strongly influenced by the directory organization of the system. Depending on where the directory information is stored, CC-NUMA systems can be divided into two categories: (a) cache-based systems and (b) memory-based systems [27]. In cache-based systems, the information about current cached copies of a block is scattered along the caches with the copies forming a distributed linked list. The home node of the block contains a pointer to the head of the list. IEEE SCI standard [133] and most earlier commercial systems (e.g., the HP SPP [22] and Sequent NUMA-Q [94]) fall into this category. Alternatively, in memory-based systems, the information about current cached copies of a block is completely maintained at the home node. Typically, a variation of the presence bit-vector directory organization is used. Most research prototypes (the Stanford DASH/FLASH [91, 85] and MIT Alewife [1]) and many current-day commercial systems (the SGI Origin [88] and HAL Mercury [144]) fall into this category. The storage overhead of the directory used to be a major concern in designing memory-based systems. Now-a-days, practical techniques like limited pointers and directory caches have been developed to reduce this overhead considerably. Due to the inherent efficiency allowed by the associated cache coherence protocol [27], memory-based systems are emerging as the trend in building CC-NUMA systems. This thesis therefore focuses on the design of this second category of CC-NUMA systems.

C. Synchronization Mechanisms

In DSM systems, synchronization mechanisms like locks and barriers enforce specific conditions among a set of conflicting accesses to global data or memory. In many earlier shared memory systems like symmetric multiprocessor (SMP) systems, synchronization variables are treated in the same way as other shared data as far as the caching and cache-coherence is concerned [51, 70], except that simple atomic instructions like *Test-and-Set* are used to access the synchronization variables. Agarwal and Cherian observed that synchronization traffic accounted for as much as 49% of total network traffic [2]. Since accesses to synchronization variables display fundamentally different patterns from regular accesses to other shared data, it is desirable to have hardware provide more support specifically for synchronization operations. As a result, recent DSM systems support synchronization mechanisms using dedicated logic and specialized protocols like *Queue-On-Lock-Bit (QOLB)* [72]. Fortunately, these enhancements require only minor additions to the coherence mechanisms being used.

The implementation of synchronization mechanisms in a DSM system is determined by the memory consistency model [52] supported by the system. The two most popularly used memory consistency models are the sequential consistency model [87] and the release consistency model [53]. More advanced memory consistency models such as the release consistency model tend to allow more overlaps among shared memory operations. However, they also tend to require the application programmers to master more software skills and more knowledge about the application problems. Thus, it is a very important tradeoff for system designers to decide what type of memory consistency model a system supports. Fortunately, most methods we propose in this thesis will work well regardless of the consistency model being used, though the performance improvement may change from one model to another.

D. Request-Response Deadlock

In the design of the protocol layer, an important issue is the method adopted to deal with the potential request-response deadlock [91, 27]. Such a deadlock occurs when the input buffer of a node is filled up by messages/transactions that can not be progressively processed by the cache coherence protocol. Thus, no new messages/transactions can be received and processed. Two main solutions are commonly used to solve this problem: to provide enough buffering or to provide (logically) separate networks [27].

The first solution is viable because there is an upper bound on the number of outstanding memory operations per node. Depending on the cache coherence protocol being used, the total number of network transactions may vary but is still bounded. Therefore, the amount of input buffer space per node for the maximum number of possible incoming messages/transactions is bounded. The buffer space can be located either at the network interface or in (local) main memory. The shortcoming of this solution is that it is too expensive in practice. DSM systems using this solution normally also employ time-out and retransmission mechanisms to reduce the amount of buffer space. The MIT Alewife [1] and the HAL Mercury [144] systems use this solution.

The second solution uses separate request-response networks. This creates no deadlock if the protocol is strictly of request-response type [27]. In such a protocol, a request transaction generates a response or no transaction, and a response generates no transaction. A node can start one or more request-response cycles only when it serves its own outstanding memory operations. However, such a protocol normally produces a long chain of network transactions to complete a memory operation. For better performance, practical coherence protocols use various forwarding techniques. These protocols detect potential deadlock situations and avoid the deadlocks by resorting to NACKs or reverting to a strict request-response protocol [27] temporarily. Such a solution is used in the Stanford DASH [91], FLASH [85], and SGI Origin [88].

1.3 Problem Description

The main objective of this thesis is to design efficient communication subsystems in DSM systems by focusing on reducing remote memory access latency. Overall, this thesis can be divided into two related parts: 1) understanding the bottlenecks and drawbacks with current implementations of the communication subsystem using simulation and analytical methods and 2) proposing novel designs to help eliminating these bottlenecks and drawbacks. Before we provide an overview of our results and solutions, we examine our target problems and the associated design issues in detail.

1.3.1 Understanding the Impact of Network Contention and Developing a Comprehensive Analytical Model for DSM Performance

A. Impact of Network Contention and Their Sensitivity to System Parameters

As indicated before, remote memory access latency is crucial to the performance of scalable DSM systems. This latency is affected by two factors: 1) overhead at the protocol layer and 2) the delay at the interconnection layer. Previous research in DSMs has mostly concentrated on techniques useful for designing better node architectures, node controllers, and efficient protocols. Almost all of these evaluations are based on the assumption of an interconnection layer with a fixed delay. Representative examples include various memory consistency models [91], data pre-fetching [101], data forwarding/updating [80], remote get/put operations [105], integrated or decoupled protocol controllers [1, 90, 85, 119], and explicit communication primitives [116]. Research towards reducing network latency has been largely left untouched. However, more recently, several papers [62, 61, 119] have reported that the delay at the interconnection layer is becoming a key architectural bottleneck in designing large scale DSM systems.

The delay at the interconnection layer contains two components: minimal transferring latency and various contention delays. The former is mainly determined by the state of the interconnection technology. The latter is caused by limited resources in the interconnection network and at the network interfaces. Current and future generation networks promise to exploit performance aggressively by using different kinds of adaptive routing [43, 78, 54, 114, 141] schemes, in addition to higher speed. Adaptive routing schemes are beneficial mainly because they can avoid network contention and congestion by selecting the path used in transferring a message on-the-fly based on the temporal traffic condition of the network. Therefore, obtaining insights to the network behavior and quantifying the impact of network contention are very important for exploiting the potential performance provided by such new generation networks. Equally important is the understanding of how the impact of network contention changes when other important system design parameters are varied.

B. Analytical Model for DSM Performance and Guidelines for Designing Networks

As pointed out above, most DSM research [138, 27] in recent years has focused on the design of the protocol layer. These studies fail to provide any meaningful guidelines, other than minimizing network latency, for designing efficient networks for DSM systems. On the other hand, many studies in interconnection networks [44, 104] have focused on the impact of network components and established design guidelines based on synthetic traffic generated according to some statistical distributions. These studies have not considered the communication characteristics and the *cause-effect* relationship between messages specific to DSM systems. For example, a recent study [139] has found that the well-known Omega network did not perform as expected in the Cedar multiprocessor, an earlier generation shared memory system. Thus, the effects of network components need to be studied carefully in the context of DSM systems to establish a set of concrete guidelines for designing suitable networks for such systems.

Recently, several studies [142, 84] have reported interesting findings about the effect of certain network components in DSM systems using specific configurations. Empirical studies like these can only identify the impact of isolated design parameters for certain system configurations. However, the fundamental relationships and interactions between key components (such as processor, cache, memory, protocols, and network) of a DSM system are complex and hard to grasp. As the technology for designing processors, memory, and networks advances largely independently of one another, it is very desirable to understand various options and tradeoffs, using both empirical and analytical methods, for designing DSM systems.

1.3.2 Support for Efficient Node-Network Interface

A. Pipelining Node-Network Interface

In DSM systems, memory blocks are the basic units for enforcing coherence and transferring data. The size of the memory blocks has a profound impact on the latency of remote memory operation. Small blocks were a popular choice in early systems to achieve low latency memory operations. Current generation DSM systems tend to use much larger blocks because they can 1) amortize the overhead at the protocol layer and interconnection layer, and 2) reduce the storage requirement of directories. Unfortunately, larger blocks also incur higher network latency. In the last few years, new network technologies have substantially reduced this latency to a few hundred nanoseconds in a medium-sized system. However, a high-speed network alone can not alleviate the bottleneck of high latency associated with remote memory operations.

Under a careful examination, it can be observed that a remote memory operation demands only a timely return of the critical word. Accesses to the rest of the block often occur much later. Thus, forwarding the critical word to the suspended thread promises performance gain. Cut-through delivery and partial cache-filling mechanisms, proposed by Mike Galles [49] and other researchers, are two attractive techniques for constructing a dynamic pipeline on-the-fly for efficient data transfer. However, to the best of our knowledge, a detailed design of node controller and network interface, collectively called the *node-network interface* (NNI), supporting these techniques has not been presented in the literature. Quantitatively, it is also not clear how much system performance can be improved by these two techniques independently and together. Thus, a breakdown on performance benefits and new insights into the design tradeoffs for the pipelining design are very interesting for system designers to know.

B. Incorporating Multi-path Network

Multiple paths consisting of physical channels [10] or virtual channels [41] exist between a pair of processing nodes in almost all modern DSM systems. They are useful for alleviating congestion, increasing throughput, and reducing average message latency. Using such multiple paths may allow messages to arrive at the destination in an order different from the one they are sent, i.e, called out-of-order (OoO) message arrivals. However, with a few exceptions [90, 88], efficient cache coherence protocols [134, 138, 24] mostly assume pairwise in-order arrival.

Currently, system architects have used two strategies in incorporating such multiple-path networks in DSM systems. The first strategy is to enhance the cache coherence protocol with sufficient intelligence and enable it to resolve all critical out-of-order (OoO) message arrivals. Unfortunately, the complexity of the resultant cache coherence protocol [144] and node controller grow significantly. The alternative strategy is to enhance the network interface (NI) with reordering capability to fix every out-of-order (OoO) message arrival. But, this leads to a noticeable increase in both complexity and overhead at the network interface. It is a real challenge to design a DSM system which can obtain most advantages of the multiple-path networks without considerably increasing the complexity of the node-network interface (NNI). Furthermore, it is very interesting to know whether an implementation of such a novel design using current technology is practical.

1.3.3 Network Support for the Protocol Layer

Most research on communication subsystem in DSM systems have focused on designing efficient components at either the protocol layer or the interconnection layer. Very few research has investigated the potential benefits of modest enhancements at the interconnection layer towards improving system performance significantly. Let us examine the following two interesting problems in this context.

A. Cache Invalidation Overhead

Write-invalidation directory-based cache coherency protocols are the popular choice in the existent DSM systems [90, 85, 81, 88, 144]. In such systems, when a request to obtain exclusive-write access comes from a writer node to the home node of a block, the home node sends write-invalidation messages to all other nodes having a copy of the block. Eventually, the home node receives all the acknowledgments and then provides exclusive-write access to the writer node. This last step is necessary to maintain *sequential consistency* [91]. Variations of this sequence are used to support other consistency models like release consistency.

Two fundamental message-passing operations are used in cache invalidation, namely, sending *one-to-many* messages from the home node to a set of sharing nodes and collecting *many-to-one* messages from the sharing nodes to the home node. The former communication pattern is known as *multicast*, and the latter as *gather*, in the interconnection networks literature. Both patterns belong to the class of *collective communication* [96], as defined by the MPI standard [99]. Recently, many new multicasting schemes and mechanisms have also been proposed [21, 97, 112]. These developments lead to a natural challenge whether we can take advantage of these schemes and mechanisms to reduce cache coherency overhead in DSM systems.

(i) Full-map Directory

In existing DSM systems with full-map cache coherence schemes, the home node can identify each sharing node at any given time. The home sends multiple *unicast* messages to all the sharing nodes and receives acknowledgments from them. Such unicast message passing increases network traffic and message contention in the network. It also makes the home node a *hot-spot* in the system. This has considerable impact on increasing the *occupancy* of messages at directories [61, 85]. Such overhead indirectly get translated into high-latency for write operations and the overall system performance gets degraded. This leads to a challenge whether collective message passing schemes can be applied to wormhole DSM systems.

(ii) Limited Directory

Due to the large amount of storage required by a fully-mapped directory scheme [91, 134], many cost-effective *limited directory* schemes using less amount of storage have been proposed in the literature. Examples of some limited-directory schemes include coarse-vector [56], Limitless [81], Superset [3], and Eviction [24]. These schemes use either hardware or software mechanisms to detect and manage directory overflow. Using less number of messages in case of directory overflow is critical to the performance of DSM systems using limited directory. Otherwise, both network traffic and node occupancy [61] increases dramatically, resulting in higher *write-latency* and overall performance degradation. This leads to an interesting question: can efficient and cost-effective directory schemes be designed for DSM systems by taking advantage of the support available on new generation networks to implement fast broadcast/multicast and reduction?

B. Bi-Modal Request-Reply Network Traffic

Some crucial characteristics of DSM systems including the *cause-effect* relationship between messages (remote read/write request message followed by a reply message), *bi-modal* traffic (short messages reflecting control messages in a cache-coherency protocol and long messages reflecting transfer of cache-lines), and the periodic generation of messages by processors (based on the computational granularity) have not been carefully considered in most DSM system designs. Naturally, an intuitive question is whether these crucial characteristics can be exploited in certain ways to improve DSM system performance.

In recent years, several commercial network switching products [49, 124, 144] have promoted virtual channel mechanism as a major feature for improving network performance. Out of these products, the SGI Spider interconnect [49] is directly geared towards CC-NUMA systems. However, this study [49] does not clearly outline the performance benefit of virtual channels in CC-NUMA systems.

Recently, several application-driven studies [142] have shown that only a negligible performance benefit exists in using virtual channels and adaptive routing in a DSM system. According to these studies, the enhancements due to virtual channels and adaptive routing might not be justified considering the associated increase in router complexity. However, these studies have overlooked several important factors such as bi-modal traffic, multiple virtual injection/consumption channels, etc. Thus, how to use virtual channels appropriately to adapt to bi-modal traffic and obtain maximum performance benefit remains an interesting question.

1.4 Thesis Overview

Having examined the research issues in the design of efficient communication subsystems for DSM systems, we now present an overview of our solutions and an outline of the remaining part of the thesis. However, before we present the overview, let us look at the assumptions we make about the system architecture.

1.4.1 Assumptions

Throughout this thesis, we make assumptions that reflect the trend in current day parallel systems. In particular, we assume a CC-NUMA architecture as shown in Fig. 1.5. The entire system consists of a number of processing nodes connected together using a scalable network. Each node has a processor, its private cache, a portion of global shared memory, and a sophisticated node-network interface (NNI). The NNI is the heart of a CC-NUMA system. Its main components include a node controller and separate interfaces for the processor module, memory/directory module, and network. It also contains logic for deadlock prevention and arbitration between the interfaces. The node controller observes all cache misses, synchronization events and uncached operations. It keeps track of the stable or transient information of ongoing memory operations, invalidates or retrieves cache blocks, completes incoming network transactions, and initiates outgoing network transactions. The network interface is mainly responsible for constructing, sending, receiving, and dispatching the actual messages. It also does all necessary work to provide the desired communication abstraction from the underlying network to the remaining part of a node.

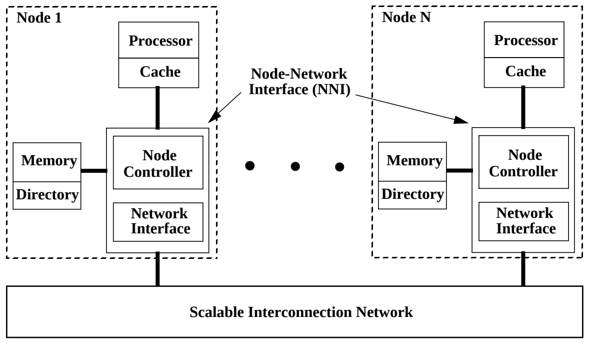


Figure 1.5: A generic CC-NUMA architecture.

We assume a wormhole routed k-ary n-cube network with fixed buffer space per channel. We assume input-buffering (using FIFO or DAMQ queuing), single (physical) injection and consumption channel pair per node for all systems (unless specifically mentioned otherwise), and the dimensional order (e-cube) or turn-model routing scheme.

For all our simulation experiments, we assume system and architectural parameters to be consistent with the best that current and future technologies have to offer. The actual values used for the system and application parameters in our studies are indicated in the respective chapters.

1.4.2 Experimental Methodology

We performed detailed simulation to evaluate the performance gains associated with our proposed designs and enhancements. A wide range of benchmark applications were used in our execution-driven simulation environment [110]. Figure 1.6 shows our experimental methodology. The FLATDSM simulates all the coherent actions of the memory system of a CC-NUMA machine on access by access basis. A flit-level wormhole-routed simulator WORMULSim was used to model a k-ary n-cube interconnection network. All simulation models were designed using CSIM [123]. The modeling of the fundamental system components such as the processor, cache, memory, node-network interface, and interconnection network is described in subsequent chapters in detail.

Throughout this thesis, we have used seven applications in our simulation evaluations. All of them are real applications or challenging computational kernels. The applications are briefly described below.

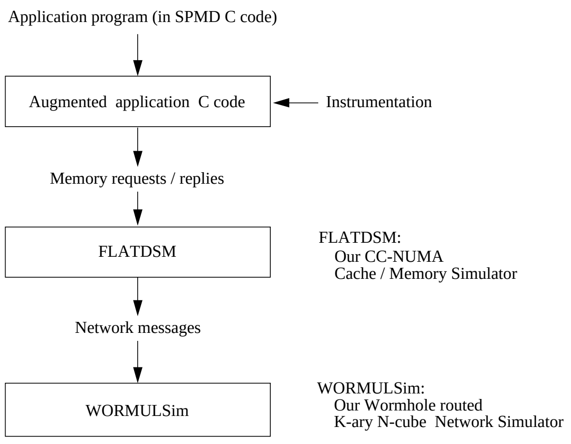

Application program (in SPMD C code)
Augmented application C code
Instrumentation
Memory requests / replies
FLATDSM
FLATDSM:
Our CC-NUMA
Cache / Memory Simulator
Network messages
WORMULSim
WORMULSim:
Our Wormhole routed
K-ary N-cube Network Simulator

Figure 1.6: Execution-driven DSM simulation framework used in this thesis.

FFT is the complex 1-D Fast Fourier Transformation computational kernel. It represents a wide variety of applications in the field of digital signal processing. The n complex input points are organized as a $\sqrt{n} \times \sqrt{n}$ matrix which is partitioned into sub-matrices and stored contiguously for each processor to improve data locality. The communication, occurring in matrix transpose steps, has an all-to-all regular pattern.

$MP3D$ is a rarefied fluid flow dynamics simulation using the Monte Carlo method. It represents a range of applications involving the force and movement of high speed objects in an extremely low density environment. Two large arrays of structures are used to store the information of molecules (the objects) and cells. The algorithm partitions the computation load on the basis of each molecule and assigns it statically to each processor. The amount of computation in MP3D may change slightly depending on the underlying machines because there are a number of harmless data race conditions in the code. The communication in MP3D is largely unstructured.

$Radix$ is the integer radix sort kernel, a major task in database applications. It iteratively creates local histograms by permuting keys and by merging them into a global histogram. The permutation phase requires, depending on the key values, an all-to-all irregular communication (in the worst case).

$Barnes$ is an application representative of the class of hierarchical N-body methods, used in astrophysics, electrostatics, and plasma physics, among others. In this application, a globally shared space-motion-mass tree (its primary data structure) is continually reconstructed and repartitioned at every time-step. Particles are assigned to each processor and accessed through pointer arrays. The communication pattern in Barnes is hierarchical and irregular.

LU is the computational kernel of lower and upper matrix decomposition. It represents a wide range of dense linear algebra applications. The 2D $n \times n$ input matrix of double precision numbers is converted into a 4D $N \times N \times B \times B$ matrix for purposes of reducing communication and load balancing. The elements assigned to the same processor are allocated contiguously to improve spatial locality. The communication pattern is one-to-many and largely regular.

$Water$ is an N-body molecular dynamics application. It evaluates forces and potentials in a system of water molecules over a period of time. The main data structure is an array of the molecules in the system. It adopts a predictor-corrector method ($O(n^2)$ algorithm) to compute the forces and potentials in each time-step. Irregular communication occurs when the cell list is updated which in turn is caused by the movement of molecules into and out of cells.

APS is the All-Pairs-Shortest-Path computational kernel which uses the Floyd-Warshall algorithm to compute all pairs shortest paths problem on an input matrix. All updates to the paths are done using the conventional in-place approach as in a sequential version.

Table 1.1 summarizes the basic communication characteristics for the applications. Except the APS application which is developed by our research group, the other applications — FFT, MP3D, Radix, Barnes, LU, and Water — are ported from the popular Stanford SPLASH/SPLASH2 [126, 146] benchmark suite. For execution-driven simulations, these applications were compiled by *dlxcc* [60] using the optimization level equivalent to O2 of *gcc* before running them on the simulator testbed.

Application	Description	Communication Characteristics
FFT	Fast Fourier Transform	all-to-all, regular
MP3D	Low density fluid flow simulation	unstructed, irregular
Radix	Integer radix sort	all-to-all, irregular
Barnes	Hierarchical N-body simulation	hierarchical, irregular
LU	Blocked LU decomposition	one-to-many, regular
Water	Molecular dynamics simulation	near neighbor, irregular
APS	All pairs shortest path solver	unstructed, irregular

Table 1.1: Communication characteristics of benchmark applications.

1.4.3 Overview of the Proposed Solutions

Having described the assumptions for this thesis, we now present an overview of our results and solutions. In Chapter 2, we focus on understanding various types of network contention and evaluating their impact on DSM performance [34, 35]. In particular, we have two main goals: 1) to estimate the impact of network link contention and network interface contention on the overall performance of DSM applications, and 2) to study the impact of critical architectural parameters on network contention. We propose a methodology of using three increasingly sophisticated network models to capture and isolate various types of contention inside network and at network interface. Based on simulation experiments using the benchmark applications and representative system parameters of current technology, for an 8×8 wormhole-routed system, our results demonstrate that network contention can degrade performance up to 59.8%. Out of this, up to 7.2% is caused by network interface contention alone. The study also indicates that network contention becomes dominant for DSM systems using smaller caches, wider cache line sizes, low degrees of associativity, high processing node speeds, high memory speeds, low network speeds, or small network widths.

Chapter 3 focuses on developing an analytical model for DSM performance and establishing concrete guidelines for designing networks [32, 36]. We systematically address this important issue in three steps. We first present a new parameterized performance model for a DSM system. This model takes into account key aspects such as the structure of an application program, advanced features of compute processor, cache/memory hierarchy and coherence protocol, and interconnection network. Then, using this model we study the effect of different network components (link speed, link width, switch routing delay, and topology) in two types of DSM systems (hardwired and customized co-processor coherence controller) and establish guidelines for designing suitable networks. Finally, we use detailed simulations driven by benchmarks to validate our model and guidelines. The results conclusively demonstrate that our parameterized model can reveal the fundamental interactions between different components in the system and correctly predict performance trends when network design parameters are varied. Regardless of the types of DSM systems, better performance is achieved by i) increasing link speed instead of link width and ii) increasing link width instead of dimensionality under constant link bandwidth. However, under constant bisection bandwidth, increasing the dimensionality of a network is not beneficial. Another important insight from detailed simulation is that network contention experienced by short messages is crucial to the system performance. Overall, this study lays a solid foundation for designing better networks for current and future generation DSM systems.

Chapter 4 focuses on designing pipelined node-network interfaces (NNI). We propose three new pipelined node-network interface designs [37]. These designs use memory sub-blocks extensively in memory hierarchy and messaging layers while still maintaining coherence at the granularity of memory blocks. They allow overlapping between message sending, receiving, and transmission by exploiting the use of micro-packets in existing high-speed interconnection networks and/or early processor restart by exploiting pipelined partial cache filling. We study important implementation issues such as interlocking between block address and sub-block data and the interleaving of sub-blocks of different memory blocks. The effectiveness of the new interface designs compared to a conventional one is evaluated based on detailed simulations. Our results show that the most sophisticated interface design can improve the performance by 5-40% across different applications compared to the baseline configuration. We have also examined the sensitivity of this design to a number of important system parameters. Overall, this study demonstrates that pipelined node-network interface designs require modest modifications to existing hardware and can improve the performance of a DSM system significantly.

In Chapter 5, we look at the problem of incorporating multiple-path network into a DSM system. We start by examining the drawbacks of the two existing strategies which involve complicated hardware logic, either at the cache coherence controller level or at the NI level. Then, we propose a new strategy that uses *block correlated FIFO channels*, which exploits the natural interplay between network interface and cache coherence protocol in scalable DSM systems with multiple-path networks [38, 39]. This strategy detects all potential coherence-sensitive (pairwise) race conditions and prevents them from occurring in the networks. It allows the use of both a simple cache coherence protocol and an inexpensive network interface. The simulation results indicate that DSM systems using our new strategy always provide either the best or very close to the best overall performance. This study provides valuable insights into the design tradeoffs in incorporating modern networks into DSM systems.

In Chapters 6 and 7, we look at the problem of cache invalidation overhead in DSM systems using full-map directory cache coherence schemes and limited directory schemes, respectively. We propose a new *multidestination* message-passing approach to implement such directory-based cache coherency with less number of messages, less network traffic, and reduced occupancy at home nodes [29, 30, 28]. A set of multidestination *reservation* and *gather* worms are used to distribute invalidation messages and collect acknowledgments. Various destination grouping schemes are proposed to generate such multidestination worms in order to implement full-map or limited directory-based cache coherence schemes on wormhole networks supporting deterministic dimensional routing and adaptive turn-model routing schemes. These mechanisms and grouping schemes are evaluated for benchmark applications. The interplay between the reduction in the number of invalidation messages, network messages, routing adaptivity, and overall reduction in execution time are also studied. The results indicate that significant reduction in overall execution time can be achieved by using multidestination messages. They also demonstrate that current and future DSM systems can take advantage of these mechanisms and schemes to deliver better performance.

Chapter 8 focuses on designing suitable networks to accommodate request and reply traffic efficiently. We present two simple techniques for exploiting the promising performance benefit of virtual channels in DSM systems [31]. First, we propose to use unequal number of virtual channels in the request and reply networks to harness performance benefit from the inherent unbalanced traffic in DSM systems. Next, we apply the virtual channel mechanism to injection and consumption channels to alleviate the performance bottleneck at the network interface. These new design strategies are evaluated through simulation using representative benchmark applications. The effect of critical system parameters (such as cache line size, routing delay, and network topology) on the new designs is also studied. Overall, our results show that the virtual channel mechanism can considerably reduce the time for shared memory accesses and the overall execution time of DSM applications. This study demonstrates that a carefully coordinated design strategy using the virtual channel mechanism can significantly improve the performance of distributed shared memory systems. Chapter 9 concludes this thesis and suggests future research directions.

CHAPTER 2

IMPACT OF NETWORK CONTENTION ON DSM PERFORMANCE

As mentioned in Chapter 1, high remote memory access latency has been a lead impediment to achieving the full performance potential of DSM systems. One of the main factors affecting the remote memory latency is the delay at the interconnection layer. This delay itself contains two components: minimal transferring latency and various contention delays. In this chapter, we study the problem of quantifying remote read/write waiting time and the contention at the interconnection layer in DSM systems. Specifically, we aim to answer the following questions: 1) how much does remote read/write waiting time affect the performance of DSM systems when network details are considered? 2) how much does network contention affect the performance of DSM systems? 3) how and to what extent do the waiting time and network contention change under different cache organizations, memory systems, processor speeds, network speeds, and link widths?

To address these questions, we examine the process of transferring a message in detail, classify the types of potential network contention, and point out the shortcomings of existing network models. To fix these shortcomings, we propose a methodology consisting of three network simulation models to evaluate the impact of network and to isolate the effect of contention occurring at the network interface and within the network, respectively. Based on this methodology, we present a comprehensive and in-depth quantitative analysis on network contention using benchmark applications. The results show that a substantial portion of overall execution time of each application is the read/write waiting time and that network contention can degrade the overall performance significantly (by up to 59.8% on an 8×8 system). Out of the overall network contention, only a small portion is caused by network interface contention (less than 7.2% out of the 59.8%), implying that the contention delay inside the network is the main one among the contention delays at the interconnection layer. We also evaluate the effect of network contention when major architectural parameters are varied. It is shown that smaller caches, larger cache lines, lower set associativity, higher processing node speeds, higher memory speeds, lower network speeds and narrower networks can significantly increase the effect of network contention on application performance.

This chapter is organized as follows. Section 2.1 characterizes components of network contention in DSM systems. Section 2.2 provides a methodology for evaluating network contention. Details of simulation environment are discussed in Section 2.3. Simulation results and discussions are presented in Section 2.4. Related work is reviewed briefly in Section 2.5. Concluding remarks are made in Section 2.6.

2.1 Message Transfer and Network Contention in DSMs

Let us consider the transmission of a message (request or reply) from one node to another in a CC-NUMA DSM system, as illustrated in Fig. 2.1. Various resources are required for this to succeed. First, space must be available for the message to be constructed in the sending buffer at the sender's network interface. Such space may not be available because of a number of reasons such as: 1) disparity between the processing rates of the node controller and the network interface, 2) link contention between messages in some part of the network. Such contention, when a message is blocked due to lack of a buffer, can be defined as *sending buffer contention.*

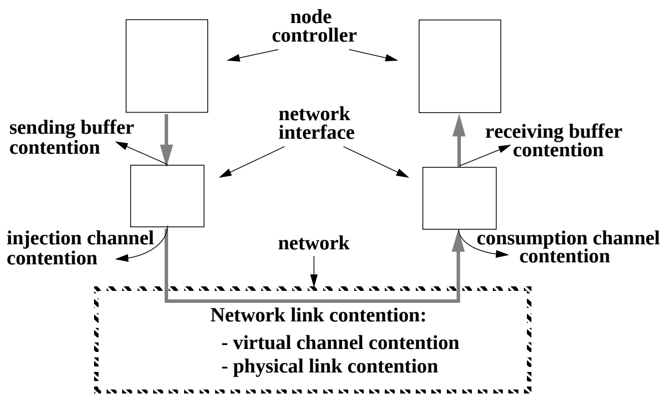

node controller
sending buffer contention
network interface
receiving buffer contention
injection channel contention
network
consumption channel contention
Network link contention:
- virtual channel contention
- physical link contention

Figure 2.1: Various types of network contention in a CC-NUMA system.

Let us consider a wormhole-routed network [104], as used in current generation CC-NUMA DSM systems like the SGI Origin. Once a message has been generated, an *injection channel* must be obtained to copy the message flit by flit into the associated router. The message may have to wait when the injection channels are being used by other messages. Such a blocking is defined as *injection channel contention.*

The entire message moves flit by flit in a pipelined fashion and holds links and the associated buffers in the network. Handshaking flow control signals are used along each link to advance the flits. As the header flit of the message passes through each router along its path, the router must make a routing decision and reserve the corresponding outgoing link for the remaining flits. If the outgoing link is not available, the message will be blocked in the network. Such a blocking is known as *link contention* (or *virtual channel contention* if multiple virtual channels share a single physical link).

In the case that multiple virtual channels share a physical link, if more than one virtual channels have flits ready for transfer, the physical link is multiplexed among the channels (*demand-multiplexing*). This effectively reduces the rate at which each message moves. This scenario is defined as *physical link contention.*

After the header flit of the message arrives at the destination, a consumption channel in the network interface of the destination node must be obtained to copy the message from the router into the network interface. Again, the message may be blocked when the consumption channels are being used by other messages. This is known as *consumption channel contention.*

Finally, a receiving buffer in the destination's network interface must be obtained to assemble the entire message before it can be delivered to the node controller. If there is no receiving buffer available, the message will again be blocked. This is known as *receiving buffer contention.*

Clearly, the communication time of a message can be significantly affected by any of the above types of contention. Due to the temporal nature, it is usually very difficult to measure the effect of the network contention on a real DSM machine. Some analytical contention models have been used before [71]. However, the accuracy lost in the analytical models, caused by ignoring the crucial dependency between computation and communication in an application, remains unclear. This leaves simulation as the most plausible approach for quantifying the effect of various types of network contention.

Due to the complexity of modeling a network, most research in DSMs has ignored network contention partially or entirely. In the WWT [117] and the Typhoon [118] simulators, a constant network latency of 100 processor cycles is assigned for every message independent of the length of the message, the distance traveled, and other traffic in the network. A DSM simulator used in the Stanford FLASH [59] research group models network interfaces. The network latency of every message is calculated based on the length of the message and half of the diameter of the network. None of these DSM simulators model the channel contention or physical link contention. Our study shows that these types of contention inside the network dominate the overall network contention for current generation network technology. In the next section, we propose some modifications to these network models which can provide us useful information about the types of network contention in DSM systems.

2.2 Methodology for Modeling Network Contention

The basic idea of our methodology is to construct a series of network models. Each of these models simulates the network at an increasing level of detail and is driven by the same DSM node and memory simulator using exactly the same input. Differences in the execution times of these models provide us information about kinds of contention occurring at the network. Specifically, we construct a series of three network models as described below. Figure 2.2 illustrates the differences and the relationships among these models.

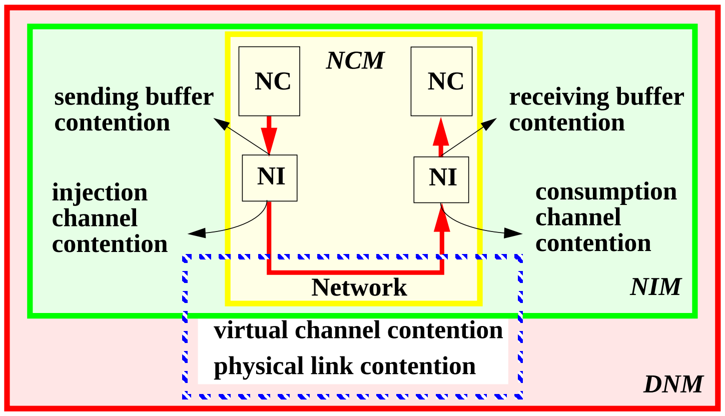

NCM
NC
NC
sending buffer contention
receiving buffer contention
NI
NI
injection channel contention
consumption channel contention
Network
NIM
virtual channel contention
physical link contention
DNM

Figure 2.2: Differences and relations among the network models.

No-Contention Network Model (NCM): This model is an enhancement of the network model used in the original WWT simulator [117]. It assumes: 1) infinite number of sending buffers, 2) infinite number of injection channels, 3) infinite number of consumption channels, 4) infinite number of receiving buffers, and 5) no traffic interference inside the network. These assumptions guarantee that no network contention can ever occur during any communication. In our NCM model, each message is delayed by the total time for construction, network propagation, and consumption. The construction time and consumption time vary depending on the length of the message. The network propagation time is a function of both the length and the distance traveled by the message. It is noted that this model captures the minimum transferring latency from sending a message to receiving it.

Interface Only Network Model (NIM): The NIM model is an enhancement of the network model used in the FLASH project [59]. The NIM model simulates detailed management of a limited number of sending buffers, receiving buffers, injection channels, and consumption channels. The NIM model still assumes no traffic interference inside the network. Such a model captures the types of contention occurring within the network interface. For every message, if there is no contention within the network interface, the message is delayed in this model by exactly the same amount of time as in the NCM model. It is clear that any difference in the predicted performance between this model and the NCM model is caused by contention occurring inside the network interfaces during application execution.

Detailed Network Model (DNM): This model simulates the detailed management of transmission links and intermediate switches, in addition to the network interface. The flow control mechanism used by the network is modeled as closely as possible. For every message that is not involved in contention inside the network, the message is delayed by exactly the same amount of time as in the NIM model. Such a model captures all types of contention within the network interfaces and every part of the network. It is clear that any difference in the predicted performance between this model and the NCM model is caused by contention occurring in the entire network (including the network interfaces). Any difference in the predicted performance between this model and the NIM model is caused by contention occurring inside the network alone (excluding the network interfaces).

2.3 Simulation Environment

To apply the above methodology for quantifying the network contention, we simulated a DSM machine similar to the FLASH [85]. In this section, we describe the specific DSM implementation used for collecting the results under different network models.

2.3.1 The Simulated Architecture

The architecture of the CC-NUMA system we simulated is illustrated in Fig. 2.3. It has 64 processing nodes. The processor in each node is assumed to be a 200 MHz single-issue microprocessor with a perfect instruction cache and a 128 KB 2-way set associative data cache with a line size of 16 bytes. The cache operates in dual-port mode using write-allocate write-back policies. The instruction latencies, issue rules, and memory interface are modeled based on the DLX design [60]. The memory bus is 8 bytes wide. On a memory block access, the first word of the block is returned in 30 processor cycles (150 ns). The successive words in the block follow in a pipelined fashion. The machine uses a full-mapped, invalidation-based, three-state directory coherence protocol [85, 90].

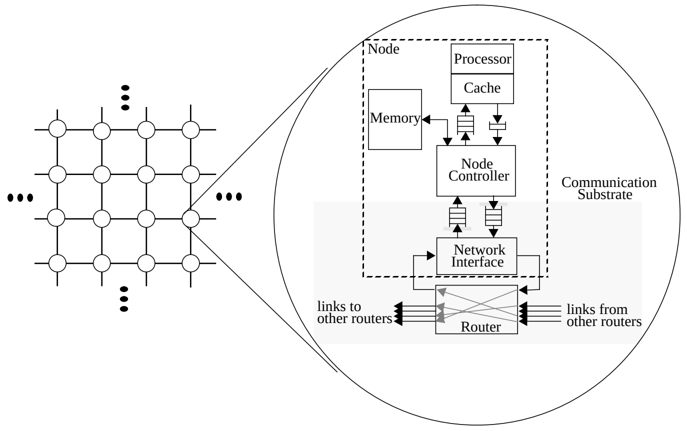


Figure 2.3: The CC-NUMA architecture simulated.

The node simulator models the internal structures, the queuing and contention at the node controller, main memory, and cache. The sizes of the queues between the internal structures and the appropriate actions taken if a queue is full are shown in Table 2.1.

Queue	Size	Action if full
Incoming network queue	8 messages	Messages blocked
Outgoing network queue	8 messages	Node controller stalls
Memory request queue	1 request	Node controller stalls
Incoming processor queue	8 messages	Processor stalls
Outgoing processor queue	1 messages	Node controller stalls

Table 2.1: Default sizes of various queues in a simulated node controller.

Following the assumptions used in [62], the node controller incurs a small fixed occupancy for purely generating/receiving a message. The node controller at the home node of a remote request/response incurs a higher fixed occupancy, because data (in most cases) and directory information must be retrieved and manipulated. In such a case, the memory access is assumed to proceed in parallel with the node controller. The node controller also incurs extra occupancy per invalidation sent. In a scenario where the processor of a node has a load/store miss to a clean block whose home node is the node itself, we assume the memory access can be pipelined with the cache access. Thus, the block retrieval, directory manipulation, and cache filling are overlapped. The above architectural features correspond to representative current generation CC-NUMA systems. Table 2.2 summarizes the system parameters used in our baseline configuration[2].

[2]All the experiments performed in this chapter assume that the target DSM system supports the sequential memory consistency [27].

Memory Hierarchy Parameters	
Processor frequency	200MHz
Cache access	1 cycle
Cache line size (L)	16 bytes
Cache set associativity	2
Cache size per node	128 Kbytes
Memory word width (W)	8 bytes/cycle
Memory response delay	30 cycles
Cache block fill time	30+L/W cycles
Memory block access time	30+L/W cycles
Node Controller Occupancy	
Directory check	7 cycles
Directory check and update	14 cycles
Each invalidation request	12 cycles
Message forward	3 cycles
Network Interface Parameters	
Outgoing message startup ($T_{outgoing}$)	15 cycles
Incoming message dispatch ($T_{incoming}$)	8 cycles
Control message size	6 bytes
Data message size	22 bytes
Injection channels per node	1
Consumption channels per node	1
Network Parameters	
Network frequency	200MHz
Channel width / Flit size	2 bytes
Link Propagation (T_{link})	1 network cycle
Router switch delay (T_{sw})	1 network cycle
Routing time (T_{rout})	4 network cycles
Physical network	1
Virtual networks	2
Virtual channels per virtual network	1

Table 2.2: Default system parameters used in the simulation.

2.3.2 Implementing Network Models

We assumed the nodes in the machine being connected with a pair of virtual networks sharing a physical 2D 8×8 wormhole network [104]. The physical network was assumed to operate at a frequency of 200 MHz. The rest of this section describes the implementation of each network model in detail.

No-Contention Network Model (NCM): The NCM model takes a message from the source node controller and informs the node controller at the destination node that a message has arrived after a time delay equal to the calculated network latency of the message. The network latency of a message is calculated as:

$$\begin{aligned} latency = \quad & T_{outgoing} + (T_{rout} + T_{link}) * distance + \\ & (T_{sw} + T_{link}) * length + T_{incoming} \end{aligned}$$

where *distance* is the number of hops from the source to the destination, and *length* is the length of the message in flits. $T_{outgoing}$ is the time delay for a message to be constructed by the network interface. T_{rout} is the time needed for a router to make a routing decision based on its knowledge about the network and the requirement of a message contained in the header flit. T_{link} is the time for any flit of a message to propagate over a single physical link. T_{sw} is the time for a non-header flit of a message to pass through a router. The parameters used in our simulation for these network delays are listed in Table 2.2. In order to guarantee FIFO delivery between a source-destination pair, end-to-end flow control is enforced in the model. It is noted that the incoming/outgoing network queues at the network interface are ignored in this model.

Interface Only Network Model (NIM): This model takes into account detailed timing and contention delays associated with structures such as the sending buffers, receiving buffers, injection channels, and consumption channels within the network interface. As discussed in Section 2.2, for a node to send a message, a space in the sending buffer must be reserved for $T_{outgoing}$ time to construct the message. Once the message is constructed, if there is an injection channel available, it is reserved for $(T_{link} + T_{sw}) * length$ amount of time to inject the entire message into the network. At exactly $(T_{link} + T_{rout}) * distance$ time after the header flit of the message is injected into the network at the network interface of the source node, a consumption channel and a space in the receiving buffer at the network interface of the destination node is reserved for $(T_{link} + T_{sw}) * length$ time. Once the message has been consumed entirely into the receiving buffer, the node controller at the receiving node is informed about its arrival after $T_{incoming}$ amount of time. In case any of the above reservations can not succeed (because of other messages in the network), the message can not move forward and continues to hold the resource(s) it has acquired. Such a network model guarantees the FIFO property of message delivery between each pair of nodes if there is only one injection and one consumption channel at each network interface.

Detailed Network Model (DNM): This model takes into account the internal structures such as data links, channel buffers, signal lines, etc., within and between the routers, in addition to the network interface. It accurately models the mechanisms for wormhole switching [104] such as: distributed routing, book-keeping on channel status, and flit-level asynchronous ready/empty handshaking. In this model, when a message is first injected into the network, the header flit of the message reserves the channels in the routers for the remaining flits of the message, while moving forward along its path from the source to the destination. When the header moves into a router, a delay of T_{rout} is incurred for the routing decision. If the (desired) outgoing channel (decided by the routing scheme) is available, the header flit reserves that channel and reaches the next router after a time elapse of T_{link}. Each non-header flit of the message can move forward only when its immediate predecessor has already moved. Such movement is modeled by using asynchronous handshaking signals between adjacent routers. Every non-header flit incurs a latency of T_{sw} at every router and T_{link} at every link on the path to the message destination. The tail flit of the message releases the previous channel when it leaves each router. When two virtual channels over the same physical link are attempt to transmit messages, the physical link is demand-multiplexed, resulting in slowdown of both messages. This model guarantees the FIFO property of message delivery between each pair of nodes under the dimension order routing scheme.

Table 2.3 shows the contention-free latencies (in processor cycles) of memory accesses under a variety of conditions. These times are deduced from our simulation models and the parameters shown in Table 2.2. To help the reader understand these values and the associated process being modeled, as an example, Table 2.3 shows the operation breakdown of the remote clean read-miss (the second row in Table 2.2). These memory latencies are comparable to the latencies presented in [92, 59], reflecting the accuracy of our simulations.

Memory Access	Home Location	Directory State	# Inv or CpyBack	Latency
Load	local	not cached	0	33
	remote	not cached	0	151
	remote (2-party)	home dirty	1	166
	remote (3-party)	remote dirty	1	335
Store	local	not cached	0	33
	local	remote shared	1	140
	remote	remote shared	0	161
	remote (2-party)	home shared	1	176
	remote (3-party)	remote shared	1	304
	remote (3-party)	remote dirty	1	335

Table 2.3: Contention-free latencies of typical memory accesses. The remote nodes are assumed to be neighboring nodes. All latency values are given in processor cycles (5ns per cycle). The fourth column (with title "# Inv or CpyBack") shows the number of invalidation or copy-back messages needed to be sent, as dictated by the cache coherence protocol being used [90, 24].

We used four applications—Barnes (2K bodies, $\theta = 1.0$, 4 steps), LU (256×256, 8×8 blocks), Radix (256K keys, 512 radix, 512K max key value), and Water (512 molecules, 4 steps) in our simulation evaluation.

2.4 Simulation Results and Discussion

We now present and discuss the results of our simulation study. All results are reported using the total execution time, normalized to the time predicted by the NCM simulator using the baseline configuration (with parameters shown in Table 2.2) for the same application. Each such time is further broken down into four categories: processor computation time (Comput), read stall time (Read), write stall time (Write), and synchronization stall time (Synch). We first evaluate the

Sub-operation	Latency
Cache-miss to request in the network	19
Request transmit time (6 bytes)	14
Request at the home node to output header transmit	45
Data return in network (22 bytes)	30
Response arrival to beginning of cache fill	11
Cache fill	32
Total	151

Table 2.4: Latency break down of a clean read-miss to a neighboring node in processor cycles (5ns per cycle).

contention for the baseline system. Next we study the impact of cache organization, memory system, node speed, and network parameters on network contention. In order to isolate their impact, we vary these parameters one at a time with respect to the baseline configuration.

2.4.1 Impact of Network and Network Contention in Baseline System

Figure 2.4 shows the execution time of each application on the baseline configuration using the NCM, NIM, and DNM model, respectively. First, the results show clearly that the shared memory read/write waiting time is a significant component of the overall execution time, ranging from 16.4% to 58.8%, for various applications in our experiments. Since the number of remote memory accesses dominates that of local shared memory accesses, the results confirm that low remote memory access latency is critical to the overall performance of a DSM system. This conclusion remains valid for all experiments throughout this section regardless of the specific parameter being varied. Second, as expected, network contention slows down every application. However, the actual effect depends on the application, ranging from 5.3% in Water to 16.8% in Radix. Radix incurs high network contention because of the all-to-all irregular communication occurring during its histogram merge phase. It is clear that network contention increases the time of each category (Read, Write, and Synch) in every application. The results show that contention within the network interface alone only accounts for a small portion of the total contention, less than 0.84% in our experiments. In other words, contention inside the network is the dominant cause of higher network latency. This observation remains true for all our experiments.

2.4.2 Impact of Cache Organization

Let us now examine the impact of alternate cache designs on the network contention. Specifically, we perform experiments by changing the cache size, line size, and set associativity.

Effect of Cache Size per Processing Node: Figure 2.5 shows the normalized execution times and their breakdowns for the systems with 32KB, 64KB, 128KB, and 256KB cache per node, respectively. Two observations are noteworthy. First, as the cache size increases, the network contention in each application reduces to a constant amount (approximately). This is because when the cache is large enough, the largest working set for an application can fit into the cache completely. Our results show that even in such a case the slowdown caused by network contention

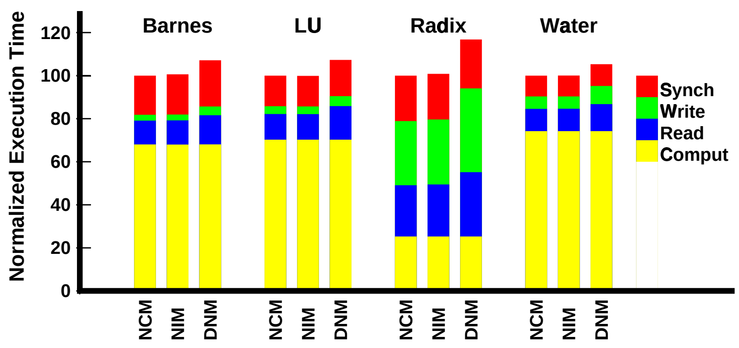


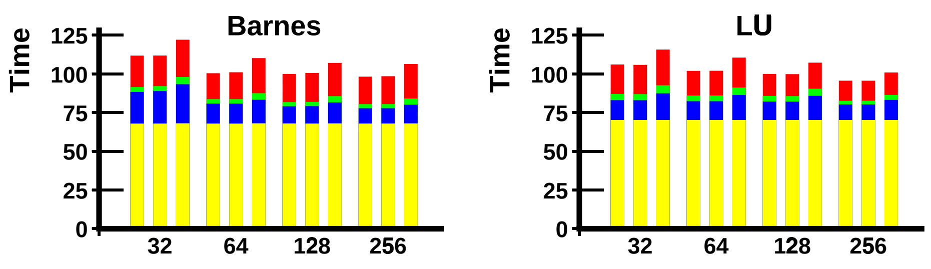
Figure 2.4: Overall execution time and its breakdown on the baseline system for three different network models.

ranges from 4.8% (in Water) to 16.8% (in Radix). This also shows that our baseline configuration gives us a rather conservative estimate of the network contention[3]. Second, when the cache size decreases, the network contention becomes more significant. The primary reason for this is the extra traffic in the network caused by cache capacity misses. This is true even when the smallest working set of an application still fits in the cache.

Effect of Cache Line Size: Figure 2.6 shows the execution times for the systems with 128 KB cache per node and cache line size of 16, 32, 64, and 128 bytes, respectively. As the cache line grows larger, the network contention in all applications shows the bath-tub behavior, i.e., first decreasing to a pollution point and then increasing significantly. The surprising discovery from these results is that the pollution point is relatively small, around 32 or 64 bytes per line, for three out of the four applications. The reason for this is as follows. When the cache line increases, the reply messages which contain data grow longer. This causes increase in the time required for transmitting them, thus longer time is spent on the links and routers, causing more network contention. Such an effect not only slows down a message of this (reply) type, but also has the potential to increase the latency of messages of other types.

Similar to a uniprocessor system, longer cache line exploits spatial locality existing in an application at the cost of reducing the benefits of temporal locality. The effect of data pre-fetching because of longer cache line can reduce the number of cache misses. However, the increase in cache line size does not reduce the number of messages by proportion beyond a certain size. In a DSM system, this leads to an increase in the overall volume of network traffic because of unnecessary data pre-fetching. This also results in more network contention and the network latency increases exponentially. Such an effect can be observed by examining the difference between the results of the NCM and DNM models for Radix: the proportion of the network latency in the overall execution time grows from 19.6% for a 32 byte cache line, to 34.7% for a 64 byte cache line, and to 59.8% for a 128 byte cache line. The reason that Radix shows this effect with smaller cache lines compared to other applications is because the communication to computation ratio is higher in this application.

[3]This is because the baseline configuration assumes a cache size large enough to hold the largest working set of any of the applications we used. This represents a system that generates very little cache-capacity miss traffic and therefore *underestimates* network contention.



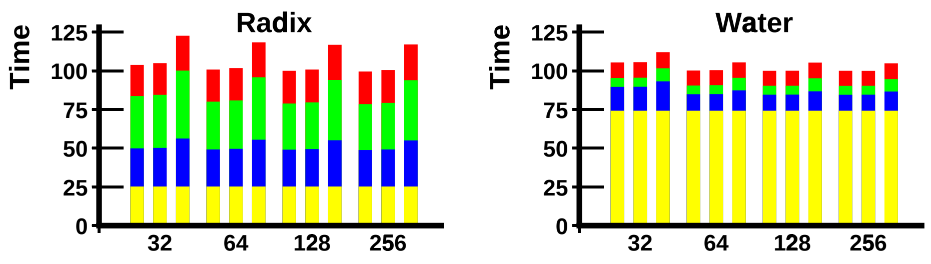
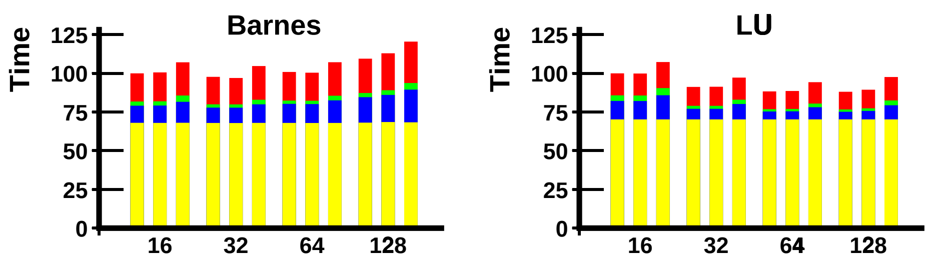


Figure 2.5: Effect of cache size per node (32/64/128/256 KB) on network contention. Every system configuration has three normalized execution times (from left to right) corresponding to the NCM, NIM, and DNM model, respectively. The breakdown of the execution time is similar to the ones used in Fig. 2.4.



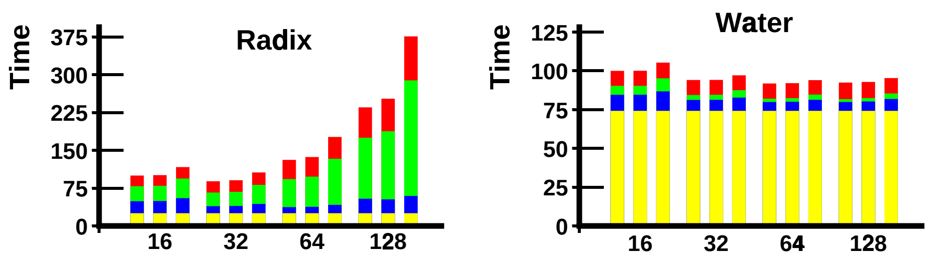


Figure 2.6: Effect of cache line size (16/32/64/128 bytes) on network contention.

Effect of Cache Associativity: The execution times of the applications when the cache is organized as direct-mapped, 2-way, and 4-way associative are shown in Fig. 2.7. When the set associativity is equal to or greater than 2, the proportion of network contention remains almost constant in each application. When the cache is organized as direct-mapped, the proportion of network contention increases in all applications because the conflict misses increase the network traffic. Our experiments show that a varying degree of performance degradation (9.1% (Barnes), 12.4% (LU), 21.1% (Radix), and 6.6% (Water)) occurs for a direct-mapped cache of size 128KB.

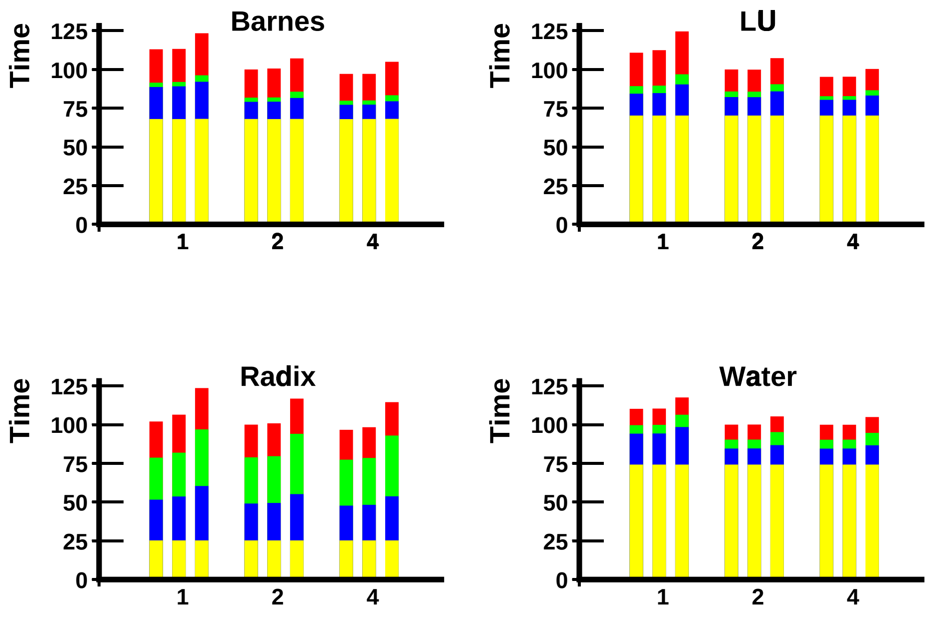


Figure 2.7: Effect of cache associativity (1/2/4-way) on network contention.

In summary, network contention is sensitive to the design choices of the cache in a node. This is especially true for selecting the cache line size. Overall, alternative cache designs other than our baseline configuration tend to increase the impact of network contention in a DSM system.

2.4.3 Impact of Memory System

We focused on two primary components in designing memory systems for DSM: memory response time and memory bus width.

Effect of Memory Response Time: Memory response time is the time interval from the issue of a memory access command to the time when the first word in a memory block is available for use. The experimental results shown in Fig. 2.8 correspond to three different response times: 20, 30, and 40 processor cycles. As much as 7.4% performance degradation occurs in Barnes, 7.9% in LU, 22.2% in Radix, and 3.1% in Water for a memory response time of 20 processor clock cycles. As the memory responds faster, the network contention increases slightly in all applications because the request and reply messages are generated in shorter time intervals, resulting in greater traffic congestion in the network.

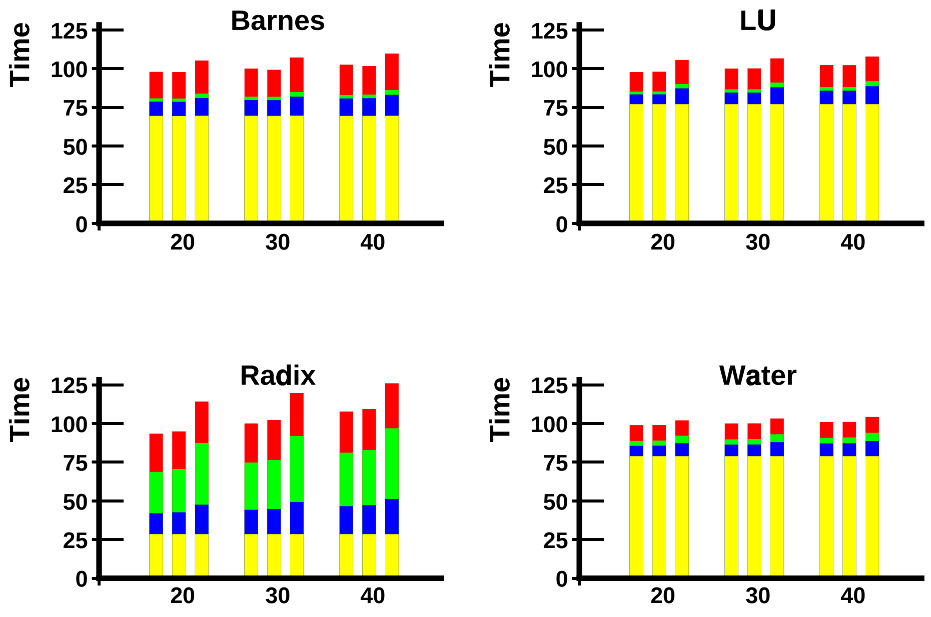


Figure 2.8: Effect of memory response time (20/30/40 processor clock cycles) on network contention.

Effect of Memory Bus Width: Memory bus width dictates the amount of time needed for finishing an access to a memory block after the first word is available. The experimental results shown in Fig. 2.9 are for memory bus widths of 4, 8, and 16 bytes, respectively. In these experiments, we used a block (line) size of 32 bytes. As the memory bus becomes wider, the network contention barely changes because of the availability of efficient memory pipelining.

Overall, our study indicates that when the memory module becomes faster in a node, the network contention increases. Between memory response time and memory bus width, the former has a stronger impact on network contention.

2.4.4 Effect of Node Speed

In this section, we examine the impact of node speed on the network contention. Our CC-NUMA system is assumed to use an integrated node controller. Therefore, when the node speed increases, all parts in a node (processor, cache, memory, node controller, and network interface) are assumed to become faster proportionately. The results in Fig. 2.10 show that the network contention increases considerably in most application as the node speed increases. Specifically, the performance degradation because of network contention worsens from 5.3% to 11.3% in Barnes, 5.3% to 7.9% in LU, 16.9% to 22.2% in Radix, and 3.1% to 3.2% in Water, as the node speed

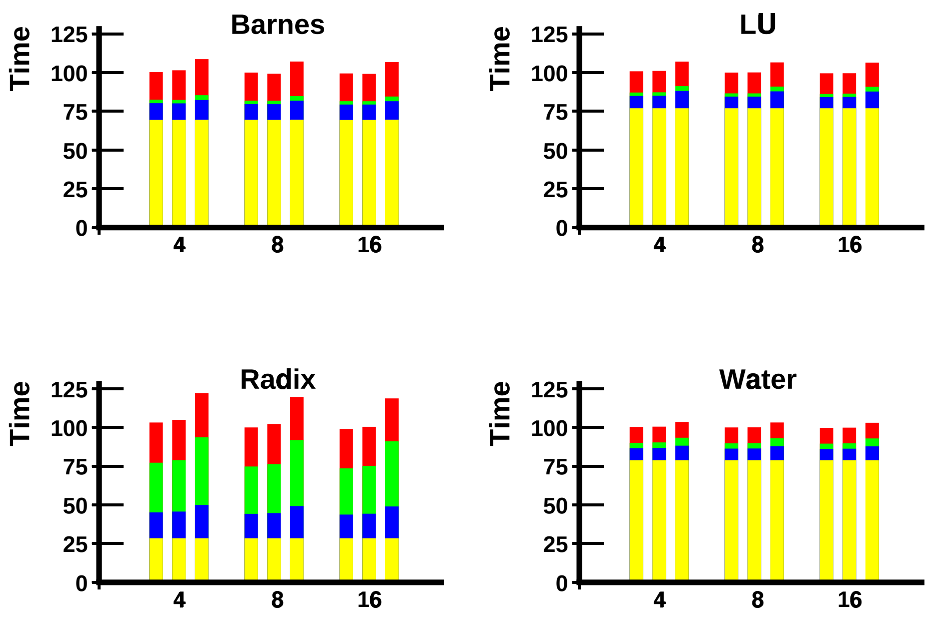


Figure 2.9: Effect of memory bus width (4/8/16 bytes) on network contention.

increases from 100MHz to 400MHz. The reason for this is, again, that the network is stressed in trying to cope with more traffic generated by the faster processor in a given amount of time.

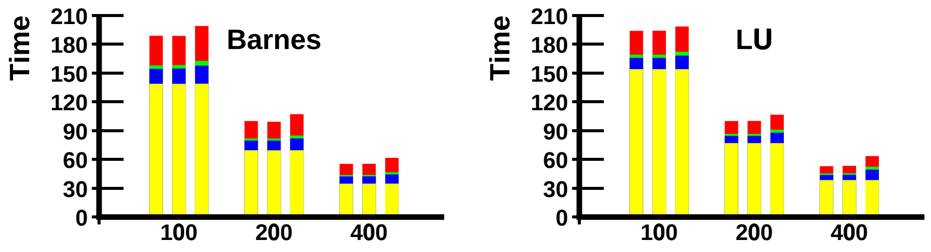
2.4.5 Impact of Network Parameters

In this section, we examine the impact of network design (network speed and network link width) on the network contention. Intuitively, a higher link speed and/or wider link width increases the bandwidth of the network, leading to less network contention.

Effect of Network Speed: Not surprisingly, as can be observed from the results shown in Fig. 2.11, the network contention is significantly worse in a slower network than that in a faster network. Our results show that the performance degradation changes from 4.8% to 11.4% in Barnes, 2.1% to 19.9% in LU, 23.7% to 32.2% in Radix, and 2.7% to 5.6% in Water, as the network speed changes from 400MHz to 100MHz.

Effect of Network Link Width: As expected, from the results shown in Fig. 2.12, the network contention is significantly worse in a network with narrower links than in a network with wider links. For example, the performance degradation increases from 7.2% to 9.5% in Barnes, 2.8% to 20.1% in LU, 12.9% to 29.0% in Radix, and 4.4% to 6.9% in Water, as the network width reduces from 32 to 8 bits.

In general, a higher bandwidth network does reduce the network contention. However, considering the commonly used narrower links and slower networks in DSM systems, network contention remains an important factor for designing high-performance systems.



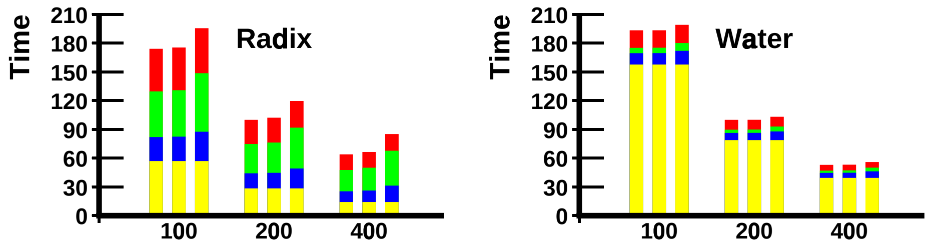
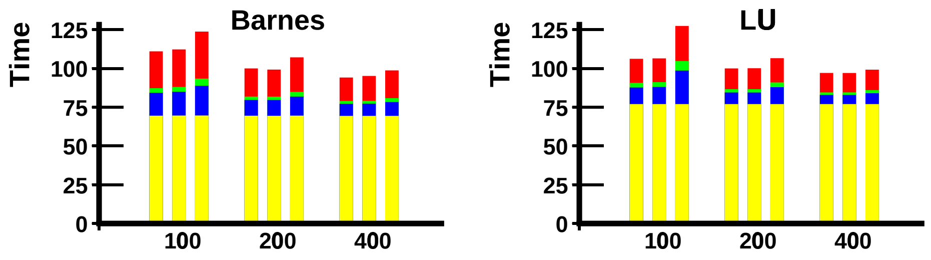


Figure 2.10: Effect of node speed (100/200/400 MHz) on network contention.



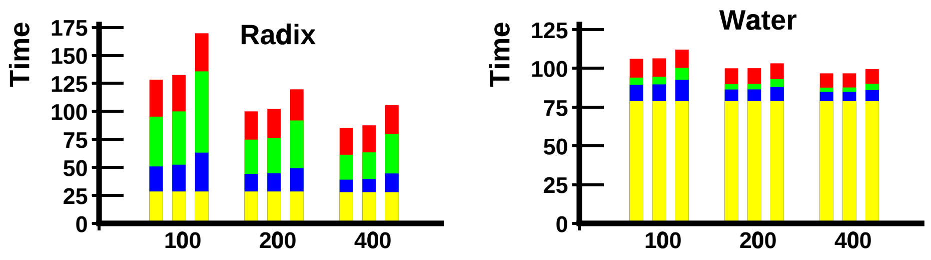


Figure 2.11: Effect of network speed (100/200/400 MHz) on network contention.

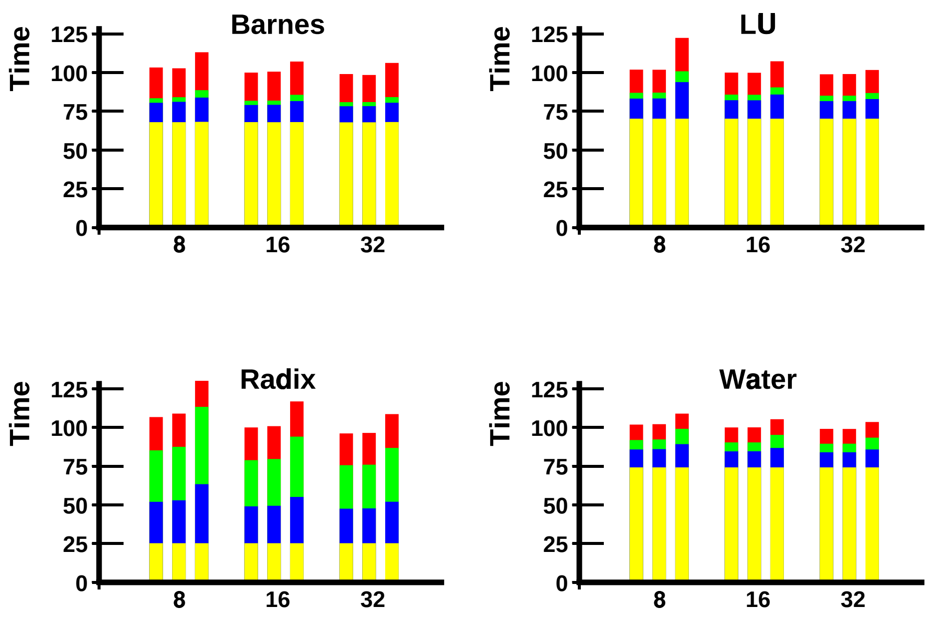


Figure 2.12: Effect of flit width (8/16/32 bits) on network contention.

2.5 Related Work

Two most popular network models in DSM research are the constant latency model and the average latency model, as used in the WWT [117] and the FLASH [85, 59] projects. As mentioned before, these models do not provide useful insights into the effect of network contention on DSM system performance. A set of network simulation models for DSM systems have been proposed in [23] to show the tradeoff between accuracy and efficiency of network simulation. However, in this chapter, our focus has been to isolate and quantify various types of network contention and study their impact on the overall DSM system performance under a set of design choices. Since network contention remains an important factor in designing DSM systems, in the next chapter, we develop a comprehensive analytical model for estimating the performance of DSM systems and derive a set of useful guidelines for designing better networks for DSM systems.

2.6 Summary

In this chapter, we have studied the impact of network and network contention on the performance of representative applications on a CC-NUMA system. Three network models have been proposed to isolate and evaluate the impact of network, contention at the network interface, contention inside the network alone, and the overall network contention. We have also studied such impact when varying a variety of architectural parameters: cache size, cache line size, cache set associativity, processing node speed, memory speed, memory bus width, network speed, and network link width.

Besides confirming that the shared memory read/write waiting time is a significant component of the overall execution time, our results show that the impact of network contention on the overall application performance is significant. Furthermore, the major component of this network contention is shown to occur inside the network alone. These results also demonstrate the importance of modeling main types of network contention in general, and contention within the network in particular, while evaluating designs for DSM systems. If network contention is taken into account, application performance can differ by as much as 60% when compared to the corresponding performance estimated by models that do not take any sort of network contention into account. When compared with models that take only network interface contention into account (and assume contention free transmission within the network), application performance can differ by as much as 50%.

Finally, our study shows that various architectural parameters can have considerable impact on the effect of network contention on the overall application performance. Smaller caches, larger cache lines, lower set associativity, higher processing node speeds, higher memory speeds, lower network speeds and narrower networks can significantly increase this effect of network contention on the application performance. These results show that changes to any of these parameters of a DSM system can have a much greater impact on the overall DSM performance if studied in conjunction with the methods for reducing the amount of network contention.

CHAPTER 3

COMPREHENSIVE ANALYTICAL MODEL FOR DSM PERFORMANCE

In Chapter 2, as we studied the impact of network and network contention, we have described a sophisticated simulation testbed for DSM systems. However, one main shortcoming of empirical methods based on either detailed simulations or measurements is that the method can only identify the impact of isolated design parameters for certain system configurations. For computer system designers, more important questions are to find the bottlenecks in existing systems and to decide which design parameters need to be improved for higher performance in future systems. To answer these questions, empirical methods typically require experiments to be done in an exhaustive manner. In practice, some kind of analytical models are used to provide a set of candidate parameters. Based on these candidate parameters, different improvements and modifications are proposed. The results are evaluated using simulations and measurements. This is also the approach we take in this thesis. In this chapter, we develop a comprehensive analytical model to estimate the performance of DSM systems. This model attempts to reveal the fundamental relationships between key components of such systems.

We achieve our goal in three steps. First, we derive a parameterized analytical performance model for a CC-NUMA architecture with a general network. This model integrates application characteristics (number of threads, number of synchronization points, granularity of computation, and ratio of read/write operation), processor characteristics (*CPI*), cache/memory hierarchy (cache miss rate, access time for cache and memory, protocol overhead, and average number of outstanding memory requests), and network characteristics (message send/receive overhead and bi-modal traffic). To the best of our knowledge, no such comprehensive model exists in the literature before. Next, as an example of using this model, we consider wormhole routed networks and analyze the impact of different network design choices such as link speed, link width, routing delay, and topology on the overall performance of DSM systems. Finally, we use an application-driven simulation approach to validate our model and obtain additional insights such as the impact of network contention. The evaluations are based on running benchmark applications on a simulated 64 processor system.

3.1 Analytical Model for a Typical DSM System

In this section, we derive a parameterized analytical performance model for a typical DSM system. This model captures three important factors: application behavior on a processor, cache/memory hierarchy, and network.

3.1.1 Characterizing Application Behavior on a Processor

Conceptually, any shared-memory application can be decomposed into a set of concurrent and coordinated threads as shown in Fig. 3.1. The set size can grow or shrink dynamically during the execution of the application as threads are created or terminated.

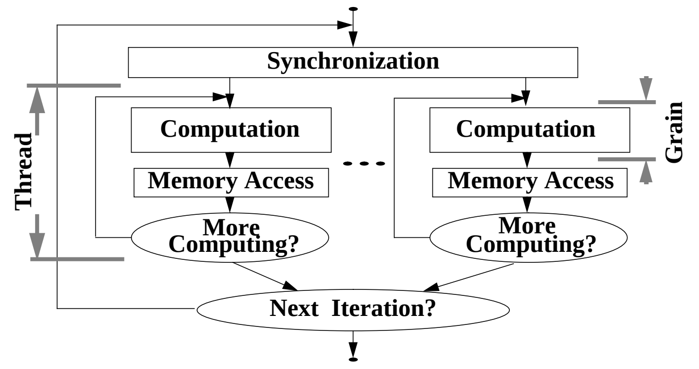


Figure 3.1: A programmer's point of view to the execution behavior of a typical DSM application.

Each of these threads carries out a certain amount of computation and communicates with one another by reading from or writing to common memory locations. In order to guarantee the correctness of the application, a subset of these reads/writes must be carried out in a specific order via inserting explicit synchronization primitives, such as barriers and locks. As a general practice, to save the high overhead of thread creation and termination, the number of threads are typically kept constant and the same threads are used before and after the synchronization points. For the purpose of performance modeling, such a thread exhibits a repeated pattern as follows: executing several register-only instructions successively for computation and then followed by a memory access (load/store) instruction. The short run of the successive register-only instructions is commonly referred to as a computation *grain* of the application. We can call the above description as the *programmer's point of view* of the application. It helps us to understand the essential behavior of an application on a given DSM system. As far as the analysis is concerned, a grain can be extended to contain a sequence of instructions which do not involve any shared memory reference or synchronization operation. Let us denote the average execution duration of a grain as T_{grn}. This can be characterized as follows:

$$T_{grn} = g * CPI * T_{pclk} \tag{3.1}$$

where g is the number of instructions in a grain, known as *grain size*; CPI is the average number of cycles per instruction of the processor, and T_{pclk} is the processor's clock cycle time. In essence, the value of g represents the inherent computation to communication ratio of an application.

As an alternative to the programmer's point of view, a *processor's point of view* can be established to provide us additional insights into the execution time of an entire application. This alternate view describes the behavior of an individual processor when running its portion of the application. Let us assume that the processor in our modeled system supports multithreading and multiple outstanding requests. Such a processor model is powerful enough to capture almost all the characteristics introduced by existing latency reduction or latency tolerance techniques, such as lockup-free caches [82], relaxed memory consistency [53], hardware and software data prefetching, speculative load and execution [120], and multithreading.

As an example, let us consider the behavior of a typical application on a processor for a short duration, as depicted in Fig. 3.2. Assume two threads, $thr1$ and $thr2$ are mapped onto this processor. At the beginning of the observed duration, which is also assumed as the beginning of one iteration, thread $thr1$ executes a grain $grn11$ and issues a nonblocking read from location $x1$. The thread continues to execute its next grain $grn12$, then issues another nonblocking read from $x2$ at the end of $grn12$ and is suspended because of data dependency. Instead of waiting for either x's data to be available, a thread context switch takes place. The processor starts running a grain $grn21$ from the second thread $thr2$. At the end of $grn21$, a blocking read from $y1$ is issued. At the moment, the processor becomes idle until one of the data items is available. As soon as the earliest item, say $x1$, is available, thread $thr1$ resumes to execute the next grain $grn13$. Assume the data items $y1$ and $x2$ become available during the execution of $grn13$. This allows thread $thr1$ to continue to execute another grain $grn14$, which waits for $x2$. Finally, thread $thr1$ issues a synchronization acquire signal and is blocked. Immediately, thread $thr2$ starts running its next grain $grn22$ and is suspended after finishing $grn22$ and issuing its synchronization acquire signal. Again, the processor is left idle until a synchronization release signal comes back. Once the expected synchronization release signal arrives, the next iteration starts.

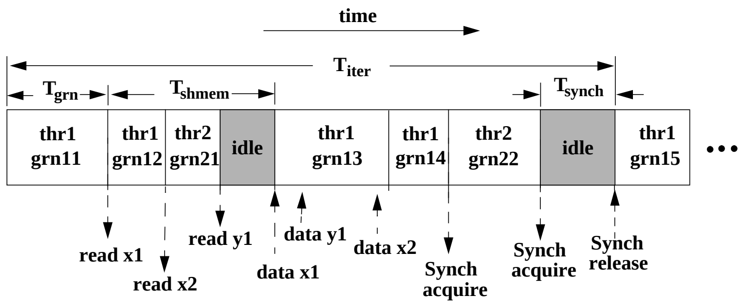


Figure 3.2: A processor's point of view to the execution behavior of a typical DSM application.

Based on the above observation, in general, the overall execution time of an application can be expressed as below assuming a perfect load balancing algorithm. For simplicity and clarity, we ignore the overheads for context switching and scheduling.

$$T_{exec} = (((T_{grn} + T_{shmem}) * N_{ref}/N_{pend}) * N_{thr} + T_{sync}) * N_{iter} \tag{3.2}$$

where T_{grn} is the average execution time per grain, T_{shmem} is the average latency per shared memory reference, T_{sync} is the average synchronization waiting time per iteration per processor, N_{ref} is the average number of shared memory references per iteration per thread, N_{pend} is the average number of simultaneous outstanding memory requests per processor, N_{thr} is the average number of threads per processor, and N_{iter} is the number of iterations of the program. Except T_{sync} and T_{shmem}, all other parameters in Eqn. 3.2 largely depend on the program and the compiler, assuming N_{pend} is less than the maximum number of outstanding requests allowed by the hardware.

Conceptually, the synchronization waiting time contains the time processors wait for one another to reach the same synchronization point and the pure time of a synchronization operation which the last participating processor executes. A synchronization operation is usually a special variant of shared memory reference, such as Fetch&Inc instruction, memory fence instruction, or QOLB scheme [91] supported in typical DSM systems. Thus, we can rewrite T_{sync} as follows:

$$T_{sync} = N_{bal} * T_{grn} + T_{shmem} \tag{3.3}$$

where N_{bal} is a load balance parameter. The value of N_{bal} depends on both the application and load balancing algorithm. Since the number of processors in a DSM system is often not a divisor of the total number of grains in an iteration, the difference of workload per processor is typically one grain unit ($N_{bal} = 1$) in the optimal scenario. In next section, we estimate the value of T_{shmem}.

3.1.2 Characterizing Cache and Memory Hierarchy

It is not very difficult to characterize the effect of caching on T_{shmem}. For simplicity, we focus on a single level cache. However, it can be very easily extended to multi-level cache hierarchy. The parameter T_{shmem} can be expressed as follows:

$$\begin{aligned} T_{shmem} &= R_{read} * T_{read} + R_{write} * T_{write} && (3.4)\\ T_{read} &= R_{rhit} * T_{hit} + R_{rmiss} * T_{r,mem} && (3.5)\\ T_{write} &= R_{whit} * T_{hit} + R_{wmiss} * T_{w,mem} && (3.6) \end{aligned}$$

where R_{read} (or R_{write}) is the rate of reads (or writes) out of total shared memory references, T_{read} (or T_{write}) is the average latency per shared read (or write), R_{rhit} (or R_{whit}) and R_{rmiss} (or R_{wmiss}) are the hit and miss ratios for shared reads (or writes), $T_{r,mem}$ (or $T_{w,mem}$) is the latency of a shared read (or write) miss reference, and T_{hit} is the latency for a cache hit reference.

The values of $T_{r,mem}$ and $T_{w,mem}$ depend on the design of memory subsystem in the DSM system. More specifically, they depend on two factors: a) cache coherence protocol being used and b) memory access latency. The overhead of a cache coherence protocol can be further broken down into two components: delays incurred at the node controller and the average number of messages generated for each read or write operation.

In this chapter, we assume directory-based protocols. Let us assume that the node controller incurs a fixed delay T_{prot} for handling each message and manipulating the directory. In a situation where data must be retrieved from the main memory, the access is assumed to proceed in parallel with the node controller operation. In a scenario where the processor of a node has a load/store miss to a clean block whose home node is the node itself, we assume the access to the memory module can be pipelined with the cache. In other words, the retrieval of the block, the filling of the cache line, and the manipulation of the directory information are all overlapped. Such overlapping is common in modern DSM systems. Under these assumptions, $T_{r,mem}$ and $T_{w,mem}$ can be estimated as follows:

$$\begin{aligned} T_{r,mem} &= T_{prot} + T_{mmod} + N_{r,s} * T_{smsg} + N_{r,l} * T_{lmsg} && (3.7)\\ T_{w,mem} &= T_{prot} + T_{mmod} + N_{w,s} * T_{smsg} + N_{w,l} * T_{lmsg} && (3.8) \end{aligned}$$

where T_{mmod} denotes the memory module access time, $N_{r,s}$ and $N_{r,l}$ denote the average number of short and long messages generated by the coherence protocol which fall on the critical path of the program's execution per read operation. Similarly, $N_{w,s}$ and $N_{w,l}$ are those numbers per write. It is worth noting that such a bimodal distribution of the message length has very important implications on the overall performance of a DSM system and especially on the design of better networks for a DSM system. We study these implications in Section 3.2 in detail.

3.1.3 Characterizing Interconnection Network

In this section, we examine the components of communication latencies for short (T_{smsg}) and long (T_{lmsg}) messages in a general network. Both latencies contain certain amount of header processing overhead at sender (T_{snd}) and receiver sides (T_{rcv}). Each of them also contains the network delay, T_{snet} or T_{lnet}, respectively. Thus, we have:

$$T_{smsg} = T_{snd} + T_{snet} + T_{rcv} \quad (3.9)$$

$$T_{lmsg} = T_{snd} + T_{lnet} + T_{rcv} \quad (3.10)$$

3.1.4 Putting All Components Together

Now, let us put all components together by substituting and merging Equations 3.2-3.10 and renaming some of the coefficients as given below:

$$R_{hit} = R_{read} * R_{rhit} + R_{write} * R_{whit} \quad (3.11)$$

$$R_{miss} = R_{read} * R_{rmiss} + R_{write} * R_{wmiss} \quad (3.12)$$

$$N_{short} = R_{read} * R_{rmiss} * N_{r,s} + R_{write} * R_{wmiss} * N_{w,s} \quad (3.13)$$

$$N_{long} = R_{read} * R_{rmiss} * N_{r,l} + R_{write} * R_{wmiss} * N_{w,l} \quad (3.14)$$

where R_{hit} and R_{miss} reflect the overall cache hit/miss ratio of all shared memory references, N_{short} and N_{long} mean the average number of short and long messages generated by the cache coherence protocol for each shared memory reference and which fall on the critical path of the program's execution. For convenience in the following discussion, let us denote $T_{iter} = T_{exec}/N_{iter}$ as the average execution time of an application per iteration per grain. Table. 3.1 provides a summary of the important parameters and notations used in this model. Using these notations, the execution time of an application per iteration per grain can be re-expressed as follows:

$$\begin{aligned} T_{iter} = {} & (N_{ref}/N_{pend} * N_{thr} + N_{bal}) * T_{grn} + \\ & (N_{ref}/N_{pend} * N_{thr} + 1) * R_{hit} * T_{hit} + \\ & (N_{ref}/N_{pend} * N_{thr} + 1) * R_{miss} * \\ & \overbrace{[(N_{short} + N_{long}) * (T_{prot} + T_{mmod} + T_{snd} + T_{rcv})}^{nni\ overhead} + \\ & \overbrace{N_{short} * T_{snet} + N_{long} * T_{lnet}}^{network\ delays}] \end{aligned} \quad (3.15)$$

The above equation is quite general. It incorporates the key factors related to the performance of a DSM system with no restrictions to the memory consistency model, cache coherence protocol,

T_{iter}	Application's execution time per iteration
N_{thr}	Number of threads per processor
T_{grn}	Average execution time of a computation granule
N_{ref}	Number of shared memory operations per iteration per thread
N_{pend}	Average number of outstanding memory operations
N_{bal}	Difference of processor's load in granules per iteration
R_{hit}	Cache hit ratio per shared memory access
R_{miss}	Cache miss ratio per shared memory access
T_{hit}	Processor waiting time on a cache hit
T_{prot}	Node controller occupancy for processing a network message at one end
T_{mmod}	Memory module access time
N_{short}	Average number of short messages on the critical path of servicing a cache miss caused by a shared memory operation
N_{long}	Average number of long messages on the critical path of servicing a cache miss caused by a shared memory operation
T_{snd}	Network interface overhead for sending a message
T_{rcv}	Network interface overhead for receiving a message
T_{snet}	Average latency per short message
T_{lnet}	Average latency per long message

Table 3.1: Summary of the main parameters and notations used in the analytical performance model for DSM systems.

or type of network being used. Once a set of values are given to the above parameters/terms, this equation can predict the execution time of any given application. It can also identify the bottlenecks associated with different components of the system. Another important observation can be drawn from Eqn 3.15 is that the third term (i.e., the last term) shows that the remote memory access latency is an important component in the overall execution time of an application. Furthermore, the remote memory access latency is shown to contain two basic components: the NNI overhead and network delays. This is in consistency with the discussions in Chapter 2.

From the recent literature [25, 146], the current generation DSM systems and applications exhibit the following range of values for the parameters used in our model: T_{grn} from 2 to 500 processor cycles, R_{hit} from 0.90 to 0.99, R_{miss} from 0.01 to 0.10, T_{hit} from 0 to 2 processor cycles, T_{mmod} from 10 to 50 processor cycles, $T_{prot} + T_{mmod}$ from 20 to 55 processor cycles (with partial overlapping), N_{short} from 0.10 to 0.30, N_{long} from 0.02 to 0.10, $T_{snd} + T_{rcv}$ from 10 to 30 processor cycles, N_{ref} from 2 to 100, N_{pend} from 1 to 8, N_{thr} from 1 to 4, and N_{bal} being either 0 or 1.

In the following section, in order to focus on the impact of different components of a network, we use the following representative set of values: $R_{hit} = 0.97$, $T_{hit} = 0$, $R_{miss} = 0.03$, $T_{mmod} = 24$, $T_{prot} = 2$, $T_{snd} = 15$, $T_{rcv} = 5$, $N_{short} = 0.10$, $N_{long} = 0.02$, $N_{ref} = 20$, $N_{pend} = 1$, $N_{thr} = 1$, and $N_{bal} = 1$. The timing values are expressed in terms of numbers of processor clock cycles.

We consider three different computational granularities to represent three different application types: $Type1$ ($T_{grn} = 80$), $Type2$ ($T_{grn} = 40$), and $Type3$ ($T_{grn} = 5$). These represent medium coarse-grain to fine-grain applications. After substituting the parameters in Eqn. 3.15 with all these values, the value of T_{iter} for each type of applications can be written as follows:

$$T_{iter,type1} = 1756.86 + 2.10 * T_{snet} + 0.42 * T_{lnet} \tag{3.16}$$

$$T_{iter,type2} = 916.86 + 2.10 * T_{snet} + 0.42 * T_{lnet} \tag{3.17}$$

$$T_{iter,type3} = 181.86 + 2.10 * T_{snet} + 0.42 * T_{lnet} \tag{3.18}$$

3.2 Impact of Network Components

In this section we study the impact of different network components on the overall performance of DSM systems based on the general model developed in the last section. We consider the k-ary n-cube wormhole interconnection, a popular choice among the current generation DSM systems, and focus on four major components of a network design: link speed, link width, routing delay, and topology.

It is well known that the average message latency in such a network contains two components: no-contention latency and contention delays. Since it is difficult to model network contention analytically [44], we consider no-contention latencies only in this section. In the simulation section, we measure message latencies with and without contention and compare them with one another.

Now let us consider the average message latencies for short and long messages, as indicated in Eqns. 3.9 and 3.10, respectively. Based on the wormhole-routing principles [40], the no-contention network latencies (i.e., T_{snet} and T_{lnet}) can be further estimated as follows:

$$T_{snet} = D * (T_{rout} + T_{phy}) + (\lceil L_{short}/W \rceil - 1) * (T_{sw} + T_{phy}) \tag{3.19}$$

$$T_{lnet} = D * (T_{rout} + T_{phy}) + (\lceil L_{long}/W \rceil - 1) * (T_{sw} + T_{phy}) \tag{3.20}$$

where D is the average distance (number of hops) traveled by a message, L_{short} is the length of a short message, L_{long} is the length of a long message, W is the width of a link (flit) in the network,

T_{rout} and T_{sw} are the routing and switching delays at each router/switch for a message, T_{phy} is the propagation delay of a flit along a single link.

Network Design Choices			T_s (pc)	T_l (pc)	Type 1		Type 2		Type 3	
					Estim.	Rel(%)	Estim.	Rel(%)	Estim.	Rel(%)
Link Speed (MHz)	T_{rout}: 1 nc	100	40	168	1911.42	100.0	1071.42	100.0	336.42	100.0
		200	20	84	1834.14	96.0	994.14	92.8	259.14	77.0
		400	10	42	1795.50	93.9	955.50	89.2	220.50	65.5
	T_{rout}: 4 nc	100	88	216	2032.38	100.0	1192.38	100.0	457.38	100.0
		200	44	108	1894.62	93.2	1054.62	88.4	319.62	69.9
		400	22	54	1825.74	89.9	985.74	82.7	250.74	54.8
Link Width (bytes)	T_{rout}: 1 nc	1	26	154	1876.14	100.0	1036.14	100.0	301.14	100.0
		2	20	84	1834.14	97.8	994.14	95.9	259.14	86.1
		4	18	50	1815.66	96.8	975.66	94.2	240.66	79.9
	T_{rout}: 4 nc	1	50	178	1936.62	100.0	1096.62	100.0	361.62	100.0
		2	44	108	1894.62	97.8	1054.62	96.2	319.62	88.4
		4	42	74	1876.14	96.9	1036.14	94.5	301.14	83.3
Topol. (dim, width)	const. link width	2, 2	20	84	1834.14	100.0	994.14	100.0	259.14	100.0
		3, 2	16	80	1824.06	99.5	984.06	99.0	249.06	96.1
		6, 2	16	80	1824.06	99.5	984.06	99.0	249.06	96.1
	const. bisec. width	2, 4	18	50	1815.66	100.0	975.66	100.0	240.66	100.0
		3, 2	16	80	1824.06	100.5	984.06	100.9	249.06	103.6
		6, 1	22	150	1866.06	102.8	1026.06	105.2	291.06	120.9

Table 3.2: Predicted performance of three different application types under different network design choices using Eqns. 3.16-3.20. Note $T_s = T_{snet}$, $T_l = T_{lnet}$, pc = processor cycles, nc = network cycles.

The short messages in a DSM system are due to control messages resulted from cache coherency. The lengths of such messages are determined by the number of message types used in a cache coherency protocol, the total number of cache blocks in the system, and data protection information in the system. The length of a long message depends on the above factors as well as the cache line size. In this chapter, we assume a directory-based cache coherency protocol with 32 different message types, 64 processing nodes, 512 MBytes of globally shared memory, and cache line size of 64 bytes. These lead to the following message lengths: $L_{short} = 6$ bytes and $L_{long} = 70$ bytes. These values are also used in the simulation experiments for easy comparison. The sizes of memory, message header and cache are small by the current standard in real DSM systems. However, these carefully selected values are large enough to run the benchmark applications using the recommended input sizes from [146] and to keep the simulation time reasonable.

3.2.1 Impact of Link Speed

Let us study the impact of link/router speed on the overall performance of DSM systems. Consider an 8×8 mesh with 16-bit wide links as our network. This leads to $D = 8$ and $W = 2$. Let us examine three values of link speed: 100 MHz, 200 MHz, and 400 MHz, corresponding to link bandwidths of 200, 400, and 800 MBytes, respectively. In this chapter, we have assumed default processor speed to be 200 MHz. For a 100 MHz link, we have $T_{phy} = T_{sw} = 1$ *ncycle* (network cycle), which is equal to 2 *pcycles* (processor cycles). Depending on the current trends in router design, let us consider two different representative values for routing delays (T_{rout}): 1 and 4 ncycles.

By using the above values in Eqns. 3.16-3.20, the overall execution times for different application classes can be computed. These execution times are shown in the top section of Table 3.2. It can be observed that as the granularity of an application decreases, increasing link speed has a substantial benefit on the overall execution time. These benefits can vary from 6.1% for a medium coarse-grain application to 35.5% for a fine-grain application. These benefits are more for networks with higher routing delays, ranging from 10.2% to 45.2%.

3.2.2 Impact of Link Width

Next, let us consider the impact of increasing network link width. Assume the same network with 200 MHz link speed and consider link widths to be 1 byte, 2 bytes, and 4 bytes, respectively. The estimated execution times are shown in the middle section of Table 3.2. Clearly, the message latencies do not reduce linearly as observed with link speed. This is because only a portion of the message latency is affected by the link width, as shown in Eqns. 3.19-3.20. Increasing link width can benefit in reduction of execution time by 3.2% to 21.1% for lower routing delay and 3.1% to 16.7% for higher routing delay. Comparing these benefits with those of increasing link speed, it can be observed that increasing link speed is more beneficial than increasing link width.

3.2.3 Impact of Topology

Finally, we consider the impact of changes in topology. For a fair comparison, we consider two different strategies for varying topologies: *under constant link width constraint* and *under constant bisection bandwidth constraint* [40]. Under the former constraint, link width is kept constant as the dimension of the network changes. Under the second constraint, link width is changed so as to maintain constant bisection bandwidth.

Let us consider three different topologies: 2D (8^2), 3D (4^3), and 6D (2^6) mesh. Under constant link width constraint, let us assume the link width to be kept constant at 2 bytes. The link speed is kept constant at 200 MHz (400 MBytes/sec bandwidth). Changes in topology lead to a change in the average distance traveled by the messages ($D = 8$, 6, and 6 for the above topologies, respectively). The corresponding execution times are shown in the bottom section of Table 3.2. The latencies for 4D and 6D systems are identical because the average distances traveled in both (given) topologies happen to be identical. With respect to the overall execution time, increasing the degree of network dimension provides very little benefit (0.5% to 3.9% only).

Under constant bisection bandwidth constraint, the link width for hypercube topology is assumed as 1 byte. This indicates that the link width of 4D and 2D systems needs to be 2 and 4 bytes, respectively. The latencies for short and long messages under this constraint show interesting trends. As the degree of dimension changes from 2 to 4 to 6, the average latency for small messages first reduces and then increases. The reduction is due to lowering the average distance traveled. However, it is quickly overpowered by reduction in link width. The average latency for long messages continue to rise with increase in dimension because of reduction in link width. These changes show increasing effect on the overall execution time for the three representative applications. The increase is quite substantial for fine-grain applications, indicating that higher dimensional networks under constant bisection bandwidth constraint are not suitable for designing DSM systems.

The guidelines derived in this section are based on only no-contention message latency. In the following section, we present detailed execution-driven simulation results to validate these guidelines and study the possible changes caused by network contention.

3.3 Simulation Results

In order to validate the analytical model and the guidelines derived in the last section, we simulated a DSM system similar to the FLASH machine [85]. Before presenting and discussing the simulation results, we first describe the basic architectural features assumed, the applications and their input sizes, and the performance metrics used in our experiments.

3.3.1 Simulation Environment

We simulated a 64-node DSM system with a default interconnection as an 8×8 mesh. The processor in each processing node was assumed to be a 200 MHz single-issue microprocessor with a perfect instruction cache and a 128 KBytes 2-way set associative data cache with a line size of 64 bytes. The cache was assumed to operate in dual-port mode using write-back and write-allocate policies. The instruction latencies, issue rules, and memory interface were modeled based on the DLX design [60]. The memory bus was assumed to be 8 bytes wide. On a memory block access, the first word of the block was returned in 30 pcycles (150 ns). The successive words in the block followed in a pipelined fashion. The machine was assumed to be using a full-mapped, invalidation-based, six-state directory coherence protocol [29, 35, 90, 85]. The node controller took 3 pcycles to forward a message and 14 pcycles to manipulate the directory. The network interface took 15 pcycles to construct a message and 8 pcycles to dispatch it. The synchronization protocol assumed in the system was the QOLB protocol similar to the one used in DASH [91], supporting the conventional sequential memory consistency semantics. The node simulator modeled the internal structures, as well as the queuing and contention at the node controller, main memory, and cache. Table 3.3 summarizes the memory hierarchy parameters and node controller occupancy delays used in our simulated architecture.

We used four applications—Barnes, LU, Radix, and Water in our simulation experiments. Problem sizes for these applications were as follows: Barnes ($8K$ bodies, $\theta = 1.0$, 4 time steps), LU (512×512 doubles, 8×8 blocks), Radix ($1M$ keys, $1K$ radix, max $1M$), and Water (512 molecules, 4 time steps). These sizes are recommended in [146] to keep the important behaviors of the applications to be the same as in a full fledged system.

3.3.2 Performance Metrics

We present all our simulation results in two sets: execution time and network latency. This helps us to correlate the behavior of underlying network with the overall DSM system performance and provide important insights into the network design issues. The overall execution time is broken down into four components: computation (C), memory read waiting (R), memory write waiting (W), and synchronization waiting (S) from bottom to top shown in Fig. 3.3-3.5. All times are normalized to that of the left most system configuration in the figures for each application.

The network latencies of two types (short and long) of messages are presented separately in absolute time (microseconds). Furthermore, we present the average latencies of short and long messages in three different ways: *estimated, ideal, and real.* The *estimated latency* was calculated according to Eqn. 3.19-3.20, where D is taken as half of the network diameter, i.e., the average

Memory Hierarchy Parameters		Network Parameters	
Processor frequency	200MHz	Network frequency	200MHz
Cache access	1 cycle	Channel width / Flit size	2 bytes
Cache line size (L)	64 bytes	Link Propagation (T_{link})	1 net cycle
Cache set associativity	2	Router switch delay (T_{sw})	1 net cycle
Cache size per node	128 Kbytes	Routing time (T_{rout})	1 net cycle
Memory bus width (W)	8 bytes/cycle	Physical network	1
Memory response delay	30 cycles	Virtual networks	2
Cache block fill time	30+L/W cycles	Channels per virtual network	1
Memory access time	30+L/W cycles		
Node Controller Occupancy		**Network Interface Parameters**	
Directory check	7 cycles	Message startup ($T_{outgoing}$)	15 cycles
Directory update	14 cycles	Message dispatch ($T_{incoming}$)	8 cycles
Each invalidation	12 cycles	Control message size	6 bytes
Message forward	3 cycles	Data message size	70 bytes
		Inject. chan. per node	1
		Cons. chan. per node	1

Table 3.3: Default system parameters used in the simulation.

distance traveled per message under uniform traffic. The *ideal latency* presented in this section (shown with a legend 'w/o contention') is the average per message among all the messages of the same type. The ideal latency for short and long messages were computed dynamically during the simulation execution by accumulating the essential delays (such as T_{rout}, T_{phy}, and T_{sw}) along the actual traversal path of the messages. In other words, this should be the measured value of average latency per message if there exists no contention in the network. Lastly, the *real latency* (shown with a legend '/w contention') is the average of network latencies across all messages of the same type. The network latency for a message was measured from the start of its injection at the sender until the completion of its consumption at the receiver. This latency includes contention at injection channel, consumption channel, and communication links. It is to be noted that the difference between the values of ideal latency and estimated latency of a message type tells us how close is the traffic of that message type to the 'uniform' traffic. Similarly, the difference between the values of ideal latency and real latency of a type tells us the effect of network contention.

3.3.3 Impact of Network Speed

Figures 3.3(a) and 3.3(b) show the impact of network speed on the overall execution time. The network speed was changed from 100 MHz to 400 MHz (200 MBytes/sec to 800 MBytes/sec bandwidth). These results refer to two different routing delays ($T_{rout} = 1$ and 4 ncycles), as discussed in Section 3.2. As predicted in section 3.2.1, the overall execution time reduces as the speed of the network increases. The reduction in execution time varies across applications, ranging from 7.3% in Water to 54.6% in Radix. From the timing breakdowns, Water appears to be the most compute intensive whereas Radix is communication intensive. In other words, the average grain size in Water is quite large compare to that in Radix. The reduction in execution time is higher for networks with higher routing delay, as predicted in section 3.2.1.

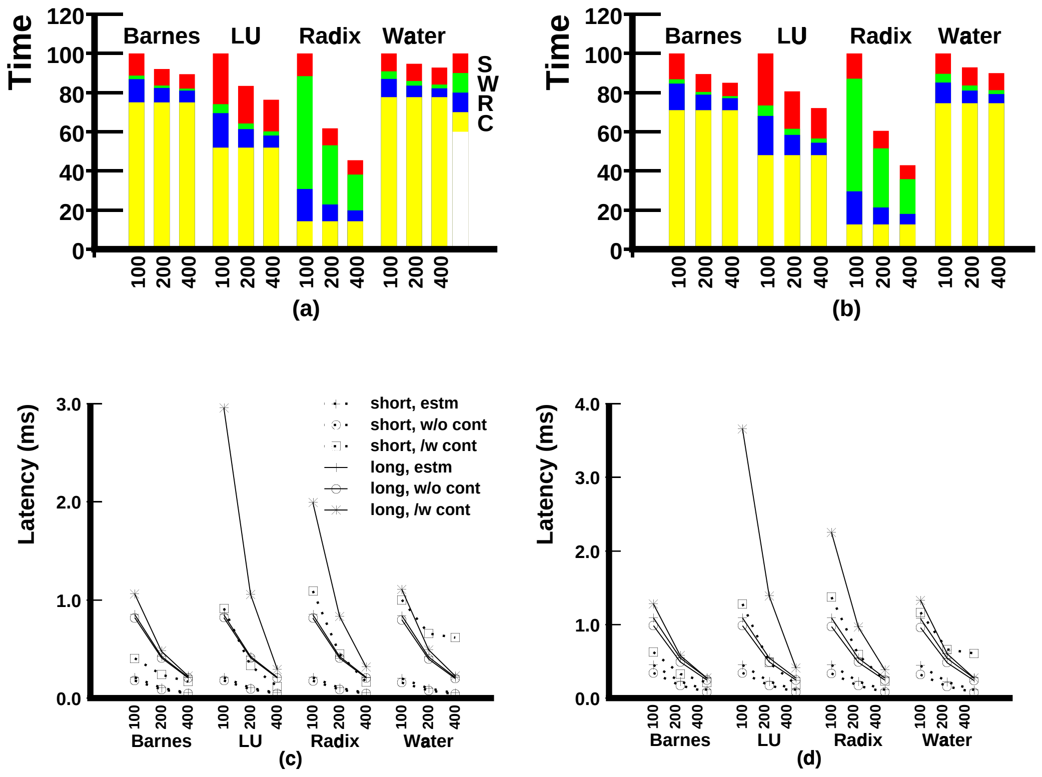


Figure 3.3: Impact of network speed on the overall execution times of benchmark applications and on the message latencies.

Figures 3.3(c) and 3.3(d) show the average message latencies for the same set of simulation experiments. Several important observations can be drawn from these data. First of all, the curves of the estimated latency are slightly above to the curves of ideal latency (no-contention) for both short and long messages. The two set of curves follow exactly the same trend and their difference margin is very small. This reveals the fact that the traffic of both message types (short and long) in a typical DSM system is close to 'uniform' traffic with a somewhat smaller average distance per message because of locality. Similar trends can also be observed from the remaining subsections.

Second, the real (with contention) latency might be much higher than the ideal latency under certain network designs. In such scenarios, contention inside the network becomes the dominant factor for network latency. Since both types of messages share the same network, the link contention between short and long messages causes the delay for short messages to be disproportionately greater. Increasing network speed causes contention to be reduced substantially for short messages, leading to overall reduction in execution time (up to 40%). Similar trends can also be observed from the remaining subsections.

3.3.4 Impact of Link Width

Figures 3.4(a) and 3.4(c) show the execution time and average latencies for the applications running under the default system configuration with varying link widths, (8, 16, or 32 bits each). These results are for routing delay of 1 ncycle. Figures 3.4(b) and 3.4(d) show the trend for routing delay of 4 ncycles. As expected from the analysis in section 3.2.2, it can be observed that wider links help reducing the overall execution time. The reduction varies from 5.3% in Water to 51.7% in Radix. The reduction is smaller for higher routing delay. The performance improvements are also smaller compared to the respective improvements obtained by increasing link speed.

3.3.5 Impact of Topology

We examine the impact of different topologies by comparing the results for 2D (8×8), 3D (4^3), and 6D (2^6) meshes under two different constraints: constant link width and constant bisection bandwidth. The data presented here are for routing delay of 1 ncycle. Other results of different topologies can be found in [33].

Under Constant Link Width Constraint:

Figures 3.5(a) and 3.5(c) show the execution time and average message latencies of each application running under various topologies. The link width was kept at 16 bits for all experiments. As predicted in Section 3.2, increasing the dimension of the network helps in reducing the execution time. However, the relative improvement is very small.

Under Constant Bisection Bandwidth Constraint:

Figures 3.5(b) and 3.5(d) show the execution time and average message latencies of each application running under various topologies. The link width was varied to maintain constant bisection bandwidth. As predicted in Section 3.2, increasing the dimension of the network has negative impact on the overall execution time. With increase in network dimension, both short and long messages undergo increasing contention, leading to increase in overall execution time. Thus, 2D systems are shown to be the best for a 64 processor system under this constraint. More generally,

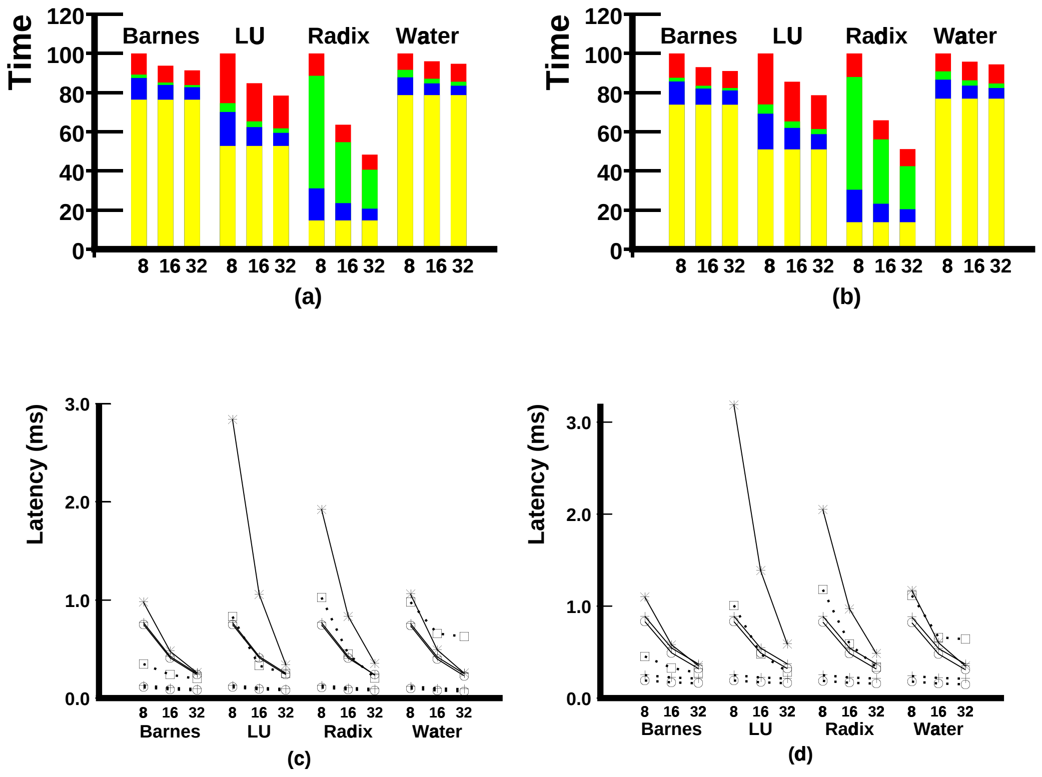


Figure 3.4: Impact of link width on the overall execution times of benchmark applications and on the message latencies.

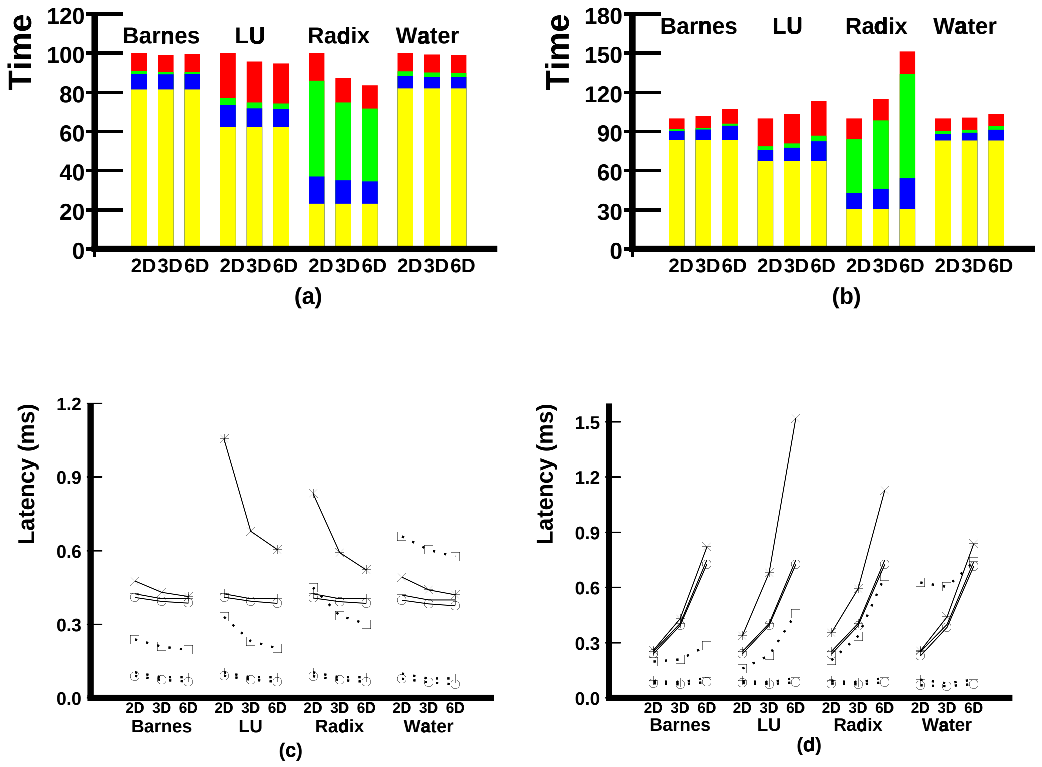


Figure 3.5: Impact of network topology on the overall execution times of benchmark applications and on the message latencies.

a low dimensional k-ary n-cube is a more preferable topology to a high dimensional k-ary n-cube for medium or large scale DSM systems under this constraint.

3.4 Discrepancy Between Analytical and Simulation Models

As mentioned before, the results estimated from our analytical model (in Section 3.2) and those obtained from our simulation testbed (in Section 3.3) show the same increasing or decreasing *trend* when each of the parameters is varied. However, the actual impacts of the parameters are different. There are two main reasons for such a discrepancy. First, the analytical model abstracts away many more system details than the simulation testbed to keep the model simple. In particular, all types of contention at a variety of system resources such as the cache, the write buffers, the memory, the model controller, the network interface, and various network components are ignored in the analytical model. Second, most parameters in the analytical model are average values, which are normally difficult to be accurate. However, such a discrepancy does not hinder us from finding out the bottlenecks in existing systems. In fact, the analytical model helps to select the right system parameters to improve system designs. In particular, this chapter, especially Eqn. 3.15, has identified the overhead at the NNI and the network delays as the two main components of the remote memory access latency. This finding corresponds closely to the simulation results in Chapter 2. In the rest of this thesis, we focus on various methods to improve these components. It is noted that simulations are used as the main tool in the in-depth studies on the effectiveness of the proposed methods because more system details can be considered in the simulation testbed than the analytical model.

3.5 Summary

Advances in processor technology is making network latency an increasingly important architectural bottleneck in DSM systems. In addition to the continued research efforts in developing latency reduction and latency tolerance techniques, a better understanding of the basic communication characteristics of DSM systems/applications and a set of concrete guidelines for designing more efficient networks targeted specifically for DSM systems are very desirable. In this chapter, we have developed a comprehensive parameterized model to estimate the performance of a CC-NUMA multiprocessor system using k-ary n-cube wormhole interconnection. This model integrates all the key aspects of a DSM system: application characteristics (number of threads, number of synchronization points, and granularity of computation), components of the node architecture (access time for cache and memory, protocol overhead, and number of outstanding memory requests allowed), and network components (link speed, link width, router delay, and topology). Based on this model we have studied the impact of different network components on the overall performance of DSM systems/applications. Finally, we have validated the impact of network components and obtained additional insights such as the impact of network contention through execution-driven simulation of benchmark applications.

Our study establishes several guidelines for designing better networks for DSM systems. Some of the important guidelines obtained from this study are: a) better performance is achieved by increasing link speed instead of link width, b) increasing dimension of a network under constant link width constraint is somewhat beneficial, c) increasing dimension of a network under constant bisection bandwidth constraint is not at all beneficial, and d) routing delay plays an important factor in the design choices of other network components.

CHAPTER 4

PIPELINED NODE-NETWORK INTERFACE DESIGNS

From the discussions in Chapters 2 and 3, especially Eqn. 3.15, it is clear that, in order to reduce the average latency of remote memory operations, we should focus on improving the minimum overhead and reducing various contention delays at the NNI and the network. In the rest of this thesis, we address the remote memory latency bottleneck problem by examining the implementations of current generation DSM systems and proposing novel solutions to improve the implementations from different aspects. In this chapter and the next one, we study two orthogonal methods to reduce the NNI overhead.

In DSM systems, memory blocks are the essential units for enforcing coherence and transferring data. The size of the memory blocks has a profound impact on the latency of remote memory operations. Early DSM systems such as Stanford DASH [91] and MIT Alewife [1] research prototypes used small blocks (16 bytes per block) for obtaining low latency. Current generation DSM systems, however, use much larger memory blocks. For example, a block size of 64 bytes is used in Sequent NUMA-Q [94]; and 128 bytes in SGI Origin [88], HAL Mercury [144], and Stanford FLASH [85]. This trend is mainly due to two reasons. First, the time overhead per remote memory operation is quite high. Thus, larger blocks are needed to amortize this overhead. Second, the storage overhead of memory directories increases linearly with the number of blocks in the system. Using larger blocks can reduce the total number of memory blocks, leading to smaller directory storage overhead. Unfortunately, larger blocks also lead to higher latency in remote memory operations.

From the discussions in Chapter 3, it is clear that the latency of remote memory operations contains five important components: cache hierarchy propagation delay, coherence management overhead, the time for sending/receiving messages, the time for message transmission, and physical memory access time. In the past few years, new network technologies have substantially reduced the time for message transmission. Examples of new high-speed networks include the SGI Origin network (a fat hypercube based on the SGI Spider switches [49]), the HAL Mercury interconnects (a flexible, irregular topology based on the PRC routers [102]), and the Cray T3E network [124] (a 3D torus). Using these technologies, a message can be transmitted across a medium-sized network in a few hundred nanoseconds. However, a high-speed network alone can not alleviate the bottleneck of high latency associated with the remaining components of the remote memory operations.

The idea of supporting cut-through delivery and partial cache-filling at the node-network interface was initially proposed in [49] to address this problem. In fact, at least two independent design teams [49, 144] have cited the potential use of these mechanisms as a main reason for doing error checking at the micro-packet level inside the network. However, to the best of our knowledge, a detailed design following this idea has not been presented in studies [49, 144] or in any other publications. In essence, cut-through delivery and partial cache-filling mechanisms, coupled with proper network technologies, can automatically construct a dynamic pipeline for efficient data transfer between any pair of nodes in the system. Such a pipeline can exploit the fact that in a DSM system, for the completion of any remote memory operation currently being resolved, only the critical word of the incoming memory block is needed. Accesses to the rest of the block often occur later, caused by instructions following the memory operation. Therefore, forwarding the critical word and resuming the execution of a suspended thread sooner promises some performance gain. However, quantitatively, it is not clear how much system performance can be improved by these two mechanisms independently and together. Thus, a breakdown of performance tradeoffs for the pipelining design is very useful. For example, consider a designer using a network which does error checking based on an entire cache block. Such a constraint would preclude partial cache-filling but not cut-through delivery. Under these circumstances, a question worth of asking is whether a pipelining design is still beneficial or not.

In this chapter, we study how to apply the pipelining idea to a DSM system similar to the Stanford FLASH. We propose a new class of pipelined node-network interface designs. Under this class, we investigate existing designs and propose three new designs. These designs use memory sub-blocks in data transferring while maintaining coherence at a block level. The first design allows early processor restart by exploiting pipelined sub-block cache filling. The second one supports overlapped message sending, receiving, and transmission by exploiting the existence of micro-packets in the high-speed interconnection network. The third design combines the advantages of the previous two. Important design issues such as interlock signaling for each design are studied. The implementation of the most efficient (the third) design is presented in detail as a working example, focusing on the organization and operation of block transfer unit—the crucial circuitry in a node-network interface. In-depth simulations have been performed to evaluate the effectiveness of the three node-network interface designs compared to a conventional design using six benchmark applications. The results show that our most efficient interface design can improve the performance up to 40% for applications. We have also examined the sensitivity of this design to a number of important system parameters such as the size of sub-block, the size of block, occupancy of coherence controller, and the size of cluster. This study demonstrates that supporting cut-through delivery and sub-block cache-filling requires only modest hardware modifications to the node-network interfaces currently being used in DSM systems and can improve the overall system performance significantly.

This chapter is organized as follows. Section 4.1 describes the system model we assumed in this study. Section 4.2 examines the basic process of remote memory operations in a typical DSM system. Section 4.3 briefly reviews the conventional block-based node-network interface design and then discusses its main performance drawbacks. Section 4.4 studies three enhanced node-network interface designs and the key design issues. Section 4.5 proposes an implementation of efficient interfaces. The results of our detailed simulation experiments are presented and discussed in Section 4.6. Section 4.7 briefly reviews some related work and in Section 4.8 we draw the conclusions.

4.1 System Model

Figure 4.1 shows the basic structure of a node controller. Overall, this structure is similar to the one used in the Stanford FLASH [85]. Separate data and control paths are provided in the node controller because data connections between the network, the memory module, and the compute processor are largely protocol independent. When a message arrives from the network or compute processor, it is divided by the preprocessor into two parts: the control part and the data part. The control part, which contains the message type (e.g., read-request) and the global address, is sent to the coherence controller (CC) while the data part is sent to the block transfer unit (BTU), if present. The coherence controller first initiates data transfer into or out of the BTU (speculatively sometimes) and then carries out the coherence book-keeping operations dictated by the protocol. The operations of the coherence controller depend on the combination of the message type and the content of the directory entry associated with the target address. Subsequently, a new control part, if a new message is dictated by the protocol, is sent to the postprocessor. In parallel to these operations in the coherence controller, the BTU can transfer the data part of the original message or that of a new message from its source to the destination. The source or destination of this local data transfer can be the compute processor module, main memory module, or network interface. Finally, the postprocessor takes the control part from the coherence controller, merges it with the corresponding data part from the BTU, and sends the resulting message out to the appropriate queue.

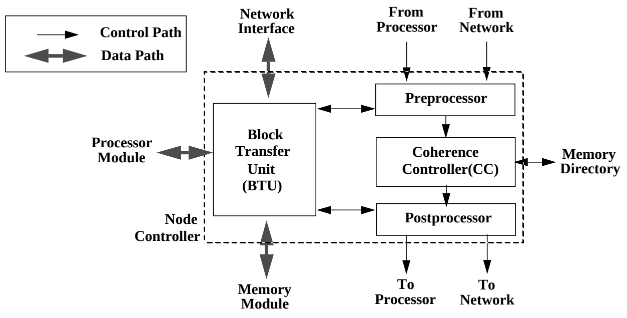


Figure 4.1: The basic structure of a node controller.

It is noted that in our model we assume independent connections from the node-network interface (NNI) to the compute processor, the main memory module, and the network. However, the same design ideas can be extended to other type of intra-nodal interconnect, such as a bus, with comparable performance benefits.

4.2 Role of NNIs in Remote Memory Operations

A DSM system provides a globally shared memory abstraction over all physical memory modules in the system. When a memory operation is issued to a location and the issuing node (which is not the home node[4]) does not have a valid copy in its private cache, a request message is sent to the home node. Eventually, a response message comes back to the issuing node with a copy of the memory block containing the content of the desired location. In such a system, a consistent view of memory is ensured at the granularity of each memory block. At any time, (directory) information is maintained at the home about the operating states of cached copies of each memory block. A cache coherence protocol is used to guarantee that a *read* to any memory location always gets a copy of the content from the latest *write* which the reading processor is aware of.

Many basic operations must be carried out for achieving various types of remote memory operations. These operations accomplish the functions of caching, directory manipulation, and message

[4]The home node of a memory location is a node in whose main memory module the location is allocated.

exchange. The exact sequence of these basic operations varies from one remote memory operation to another depending on the temporal content of a directory entry. Consequently, understanding the sequence of typical remote memory operations is often challenging.

As a working example of remote memory operations, let us consider the process of resolving a read cache miss to a remote memory location whose directory entry indicates the "uncached" state. When a compute processor issues the read, a cache miss is first detected and propagated to the NNI on the requesting node. This NNI locates the home node, constructs a network message containing the read request, and sends the message to the home node via the network. Once the message reaches the destination, the NNI on the home node retrieves the target block from the memory module, modifies the state and sharer list in the associated directory entry, constructs a reply message containing the retrieved target block, and sends the message back to the requesting node. When the reply message arrives, the NNI on the requesting node fills the cache and restarts the compute processor. The entire process can be divided into two phases: the request phase and the response phase. The request phase starts from the time when the read is issued by the compute processor until the time when the request message reaches the NNI at the home node. The response phase starts from the time when the target data is being retrieved until the time when the retrieved data reaches the processor.

The latency of the above remote memory read operation can be divided into five basic components, as described below:

1. *Req-Cache:* Time required by the requesting compute processor module to a) detect a cache miss, replace a dirty cache line if needed, and change the line to a wait state in the request phase; and b) fill a cache line, merge dirty data in relevant write buffer, provide the data to the processor, and change state to the "shared" state in the response phase.

2. *Req-NNI:* Time required by the NNI at the requesting node to a) locate the home node, construct the request message, and inject the message into the network; and b) consume the reply message from the network, extract the address and data from the message, and forward the data to the requesting cache.

3. *Network:* Time required by the network switches and links to transfer the request and reply messages from the source to the destination nodes.

4. *Home-NNI:* Time required by the NNI at the home node to a) consume the request message from the network, check the state of the associated directory entry, and issue the main memory read operation; and b) construct the reply message with data and modify the state and sharer list in the directory entry.

5. *Home-Memory:* Time required by the physical memory module at the home node for reading an entire memory block.

Clearly, the NNIs are traversed many times in the above process. For more complicated remote memory operations such as write misses to remote locations shared by multiple nodes, similar steps may be needed on the NNIs and caches at one or more third-party nodes depending on the state and sharer list in the directory entry. Therefore, the NNI's performance is crucial to reducing the overall latency of various types of remote memory operations.

4.3 Conventional Block-Based NNI Design

In most existing DSM systems, memory blocks are the basic operational unit for the NNI. In such a conventional block-based NNI, denoted as CBB-NNI design, any operation on the data must be atomic and sequential at the granularity of a block. For example, when a block of data is read from the main memory module into the NNI over the memory connection, the NNI does not start to forward the data to the processor module or to the network until the entire block has reached the NNI. Many systems such as the Stanford DASH [91], MIT Alewife [1], HP/Convex SPP [22], Sequent NUMA-Q [94], Wisconsin Typhoon [119], and MIT StarT-Voyager [7] use different variants of the CBB-NNI design. The wide use of the CBB-NNI design can be attributed to the fact that the design is a natural extension of the block-based memory access scheme used in bus-based symmetric multiprocessor (SMP) systems. The CBB-NNI design conforms to many aspects of current industrial standards and keeps the interactions among the NNI, the main memory module, the processor module, and the network to the minimum, therefore resulting in low complexity/cost in design and implementation.

In the CBB-NNI design, however, the latency of copying a full block is incurred each time a memory module is accessed, a cache is filled or retrieved, or a message containing data is moved in or out of a processing node. As an example, let us examine the components of the remote memory read operation outlined in Section 4.2. Figure 4.2 illustrates a time-line of the five components. The shading pattern of each horizontal bar indicates the type of work causing the particular time component. For example, the four bars with line pattern at the left side of this figure show the time in the request phase for handling the cache miss at both the requesting node and the home node and for transmitting the request message. The remaining bars (the blank boxes) show the time in the response phase for resolving the miss at both nodes and for transmitting the reply message. The time period of the entire process is commonly denoted as the back-to-back memory latency. It is noted that the rightmost bar in the figure shows the cache hierarchy busy time. The compute processor can resume its execution shortly after the beginning of this bar using the critical word forwarding path. The time period of the entire process excluding the last cache-filling time (last bar) is commonly denoted as the restart memory latency.

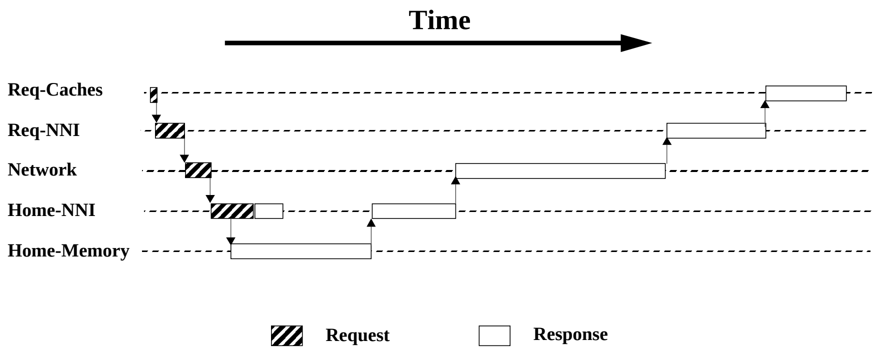


Figure 4.2: The timeline of a read miss to a clean remote memory block using the CBB-NNI design of node-network interface.

It can be observed from Fig. 4.2 that block copying is incurred five times in the response phase for transferring data: a) the target data is copied from the main memory module into the block transfer unit (BTU) at the home node; b) the data, as part of the outgoing message, is copied from the BTU into the network interface; c) the data, as part of the incoming message, is "store-and-forwarded" (a network copying variation) at the network interface on the requesting node; d) the data, as part of the incoming message, is copied from the network interface to the BTU on the requesting node; and e) the data is copied from the BTU into the cache hierarchy at the requesting node. With a block size as large as 128 or 256 bytes, the cost of block copy operations dominates the latency of remote memory operations[5].

4.4 Pipelined NNI Designs

In this section, we propose three pipelined NNI designs based on the cut-through delivery and partial cache-filling techniques. For all of them, memory block remains the granularity for maintaining memory coherence, while sub-block is the granularity for transferring data. This means that the state in the directory entry or cache line must be changed after the last sub-block data of a block is written and before the first one is read.

4.4.1 NNI Supporting Partial Cache-Filling

The concept of partial cache-filling and cache sub-blocks has been used in uniprocessor and SMP systems for quite some time. A variant was proposed in the original FLASH design [59, 85] for DSM systems. This enhancement works for retrieving/filling a cache line; for reading/writing data from/into the memory module or the network. It eliminates multiple copies by placing data in the block transfer unit (BTU) as they are received from the network or processor module. It makes the latency of a data transfer dependent of sub-block size, instead of block size. We denote the NNI enhanced with partial cache-filling support as the PCF-NNI design.

In the PCF-NNI design, the temporary storage in the BTU can be arranged as a number of buffer entries. Each entry can store data up to a memory block. Each data item (e.g., a sub-block) in every buffer entry is associated with a valid bit. The valid bit is set each time the data item is written and cleared when it is read. The BTU keeps all data items from a single block in the same buffer entry until the entire block is delivered to the destination. Via the entry index, control information can be found for delivering the data properly. The BTU typically operates in dual-ported mode with one port for read and the other for write. A destination unit monitors the valid bits for reading data from the BTU. Therefore, it does not have to wait for the entire buffer entry to be filled before starting data transfer. Figure 4.3(a) illustrates the timeline of the remote memory read operation using the PCF-NNI design. Compared to the CBB-NNI design, this design substantially reduces the store-and-forward delay incurred at the BTUs for the reply message.

4.4.2 NNI Supporting Cut-Through Delivery

Cut-through and packetization have been used extensively in modern high-speed networks [40, 124]. These techniques avoid the store-and-forward latency at each hop, leading to lower network latency. As a natural extension, an NNI can use its network interface as a cut-through device by transferring a message as successive micro-packets. Using such NNIs, a pipeline exists from a

[5]It is noted that if the critical word forwarding is used in the processor module, the cost of the last block copy is not a part of the restart memory latency. But the cost is still a part of the back-to-back memory latency.

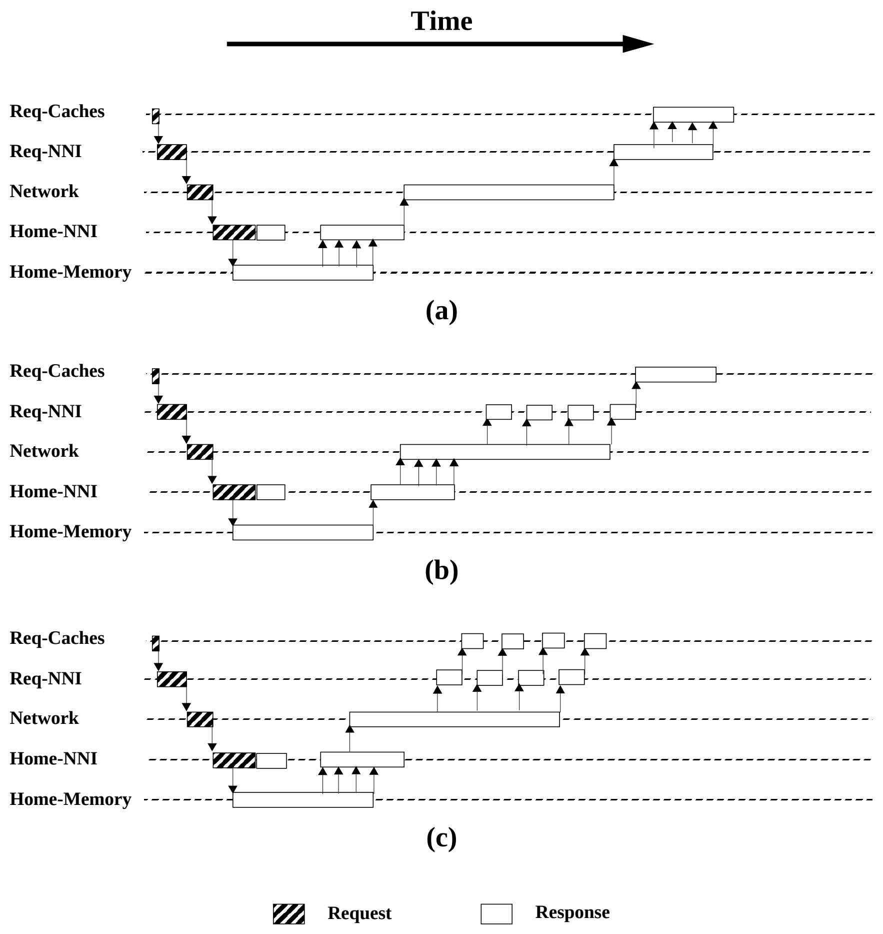


Figure 4.3: The timeline of a read miss to a clean remote memory block using pipelined node-network interfaces (NNIs): (a) PCF-NNI, (b) CTD-NNI, and (c) ICP-NNI.

sender's network interface (NI) to the receiver's NI across the network. On the sending side, the node controller moves outgoing micro-packets from the postprocessor or BTU to the NI as soon as a set of micro-packets are ready[6]. The NI then immediately puts these micro-packets onto the network link (if it is available) one after another. On the receiving side, the node controller moves the incoming micro-packets into the preprocessor or BTU as soon as a certain number of micro-packets have been deposited in the NI from the network link. We can denote the NNI enhanced with cut-through delivery as the CTD-NNI design. Most existing networks calculate the CRC based on a full message (entire cache line) instead of each micro-packet. Cache line based error checking would preclude pipelining into the cache before the CRC is complete.

The key issue in the CTD-NNI design is the handshake signaling between the node controller and the NI. A simple solution is to manipulate the transmitting and receiving FIFO queues at the NI in the unit of micro-packets, rather than full messages. The head micro-packet and the last tail micro-packet of a message can be handled differently to do the setup and tear-down work at the sending and receiving sides. Micro-packets belonging to a single message are pipelined through a queue in transferring order. A pair of head and tail queue pointers can be used to keep the operation synchronized promptly[7]. Figure 4.3(b) illustrates the timeline of the remote memory read operation using the CTD-NNI design. Compared to the CBB-NNI design, this design substantially reduces the store-and-forward delay incurred at the NIs.

4.4.3 Integrating CTD and PCF

From the previous two sections, it is clear that the cut-through delivery and partial cache-filling mechanisms largely avoid block copying latency at different stages of remote memory operations in the CBB-NNI design. Therefore, there is a potential to combine the benefits of both by integrating them. Let's denote it as the ICP-NNI design. Compared to the PCF-NNI and CTD-NNI designs, this design establishes a complete pipeline from the sender's main memory module or cache to the receiver's processor module across the sender's NNI, the network, and the receiver's NNI. It allows the application to restart (if stalled) much earlier before the entire block transfer completes, despite the fact that the same number of bytes are transferred. Figure 4.3(c) illustrates the timeline of the remote memory read operation using the ICP-NNI design. As mentioned earlier, even though the ICP-NNI design has been used in the SGI Origin [88], no design details are available in the literature. Therefore, we discuss several key issues for its design below.

Tag and data. For transferring a sub-block between the BTU and the main memory/processor module, the associated address tag must be explicitly provided. However, for transferring a sub-block between the BTU and the network interface, no memory address is needed because the address is implicitly contained in the message header. One solution is to maintain the block address and to store all the sub-blocks of a block contiguously in the BTU. This solution is similar to the one used in the FLASH [85, 59] design. This solution requires explicit synchronization at the sub-block level. The complexity for synchronization may become too high when sub-blocks which belong to different memory blocks are allowed to be stored and forwarded in an interleaved manner. A simple alternative is to store the address of each sub-block together with the data in the BTU. It eliminates the requirement of contiguousness. However, a large portion of the address storage

[6]The number can be a design parameter.

[7]For clarity, we assume that each virtual channel, if supported, has its own transmitting and receiving FIFO queues.

is wasted because the block address is embedded in the address of sub-block and stored multiple times. A solution better than both of these schemes will be presented in the next section.
Induced cache miss. Consider two successive remote operations targeted at the same memory block. Let's assume that the first operation results in a cache miss and starts the resolving process. After the critical word is forwarded, the processor issues the second operation shortly. At the time of cache checking for the second memory operation, the cache line corresponding to the block is still in a transit state because the first memory operation has not been completely resolved yet. A simple solution to this problem is to stall the second operation until the state of the cache line becomes a stable one, which will happen after the first memory operation is completely resolved.
Unrelated cache misses. Consider two successive remote memory operations targeted at different blocks. Sub-blocks belonging to different blocks may compete for the BTU. This situation occurs more often in the ICP-NNI design. The simple solution of stalling the second operation until the first one is completely resolved will work. But it prevents two independent home nodes from resolving these operations in an overlapped fashion. A slightly better solution is to buffer the reply message of the second operation at the NI until all sub-blocks of the first memory operation enter the BTU. This solution allows more overlapping than the last one, but it still prolongs the second memory operation. In the next section, we will present a solution which allows sub-blocks belonging to different blocks to use the BTU in an interleaved manner to obtain the full performance benefit of pipelining.

4.5 Implementation of BTU

This section proposes a detailed implementation of the BTU[8]. It demonstrates how the ICP-NNI design can be realized and provides a common basis for evaluating the performance of various pipelined NNI designs.

Fig. 4.4 shows our proposed organization for the block transfer unit (BTU). Internally, it contains two sub-structures: a descriptor table and a data FIFO. The descriptor table contains a number of entries held by blocks currently being processed. Each descriptor entry contains the following fields: valid bit, block tag, offset, in-count, out-count, source, and destination. Most of the fields need no explanation. The "in-count" (or "out-count") field indicates the number of sub-blocks of a block having being copied into (or out of) the BTU. The "offset" indicates the offset within the block of the next sub-block being transferred. Together with the "block tag" field, this field is used each time a sub-block is transferred to the memory module/processor. The data FIFO also contains a number of entries. Each entry contains a "data" and "descriptor index" fields. The "data" field stores the copy of a sub-block. The "descriptor index" field points to the descriptor entry containing other information related to this sub-block.

With the above organization, the basic operation of the BTU can be described as follows. When a message arrives from the network interface/processor, the preprocessor extracts the control information such as the source ID, message type, and target address from the header of the message. The information is predecoded and an entry in the descriptor table in the BTU must be reserved successfully. Once the reservation succeeds, the predecoded information is passed onto the CC for coherence protocol processing. The CC initiates any data retrieval dictated by the protocol resulting in a series of sub-blocks being copied into the BTU. At the same time, the other fields of the descriptor entry are initialized appropriately. The "block tag" and "offset" fields are filled by

[8]This is a typical implementation we consider in this chapter. Other implementations are possible based on the technological trend.

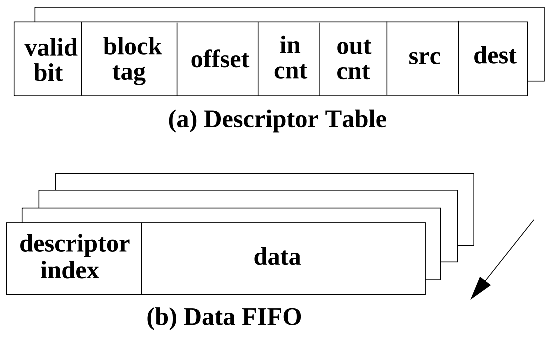


Figure 4.4: The organization of the block transfer unit (BTU).

the decomposition of the target address which is the address of the first (critical) sub-block. The "source" field is set to the source for data transfer: individual network channel (e.g., request or reply channel), processor module, or memory module. The "destination" field, decided by the coherence protocol, is set similarly when it is known and ready for forwarding. Either the preprocessor, the CC, or the postprocessor can set the destination. Occasionally, more than one destinations are desirable, e.g., the main memory module and the processor module.

When a sub-block reaches the BTU, it is stored into the data FIFO. The descriptor index is found from the descriptor table by an associative search on the source of data transfer. The "in-count" is incremented. As soon as the FIFO contains data, the first item in the FIFO can be forwarded to its destination using the information from the descriptor entry found via the "descriptor index". As each sub-block is forwarded to its destination, the associated "out-count" is incremented. So is the "offset" field (in modulo arithmetic). The entry in the data FIFO is released immediately after being forwarded. When both "in-count" and "out-count" reach the maximum (i.e., the number of sub-blocks in a block), the descriptor entry can be released. All the sub-blocks belonging to the same block pass through the data FIFO in increasing order starting from the critical one with wraparound. But sub-blocks belong to different blocks are allowed to be interleaved, as shown in Fig. 4.5.

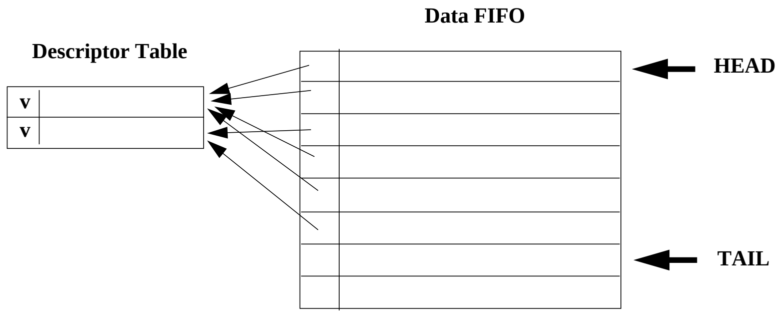


Figure 4.5: The association between the data FIFO and the descriptor table in the BTU.

When a new message (the header) reaches the postprocessor from the CC and if the data part (the tail) is needed, the destination in the descriptor entry can be set. The header is forwarded to the network interface or processor module immediately. The tail of the message is forwarded to the destination when each sub-block is available in the BTU.

It is important to note that operations involving the processor cache interface are usually dictated by the processor implementation. For example, most processors today would not allow the system to interrupt a read response transaction in order to issue a new read request. However, many modern processors will issue multiple speculative reads in order to hide memory latency. Our BTU implementation is applicable for both types of processors.

4.6 Performance Evaluation

In this section, we use a detailed simulation to evaluate the performance of our proposed node-network interface design. First, we describe the simulation parameters and methodology, then present and discuss the results for the latency of a simple remote clean read miss. Finally, we present simulation results using a suite of representative application benchmarks.

4.6.1 Simulation Environment

The system we simulated had the generic DSM architecture discussed in Section 4.1. The entire system consisted of 32 processing nodes with 2 processor modules in each node (like the SGI Origin [88]). A full-mapped, invalidation-based, three stable state directory coherence protocol [90, 85] was used. The compute processor was supported by a 16 KBytes direct-mapped write-through L1 data cache with a line size of 32 bytes and a perfect instruction cache. The instruction latencies and issue rules were modeled based on the generic superscalar RISC processor design [60]. The processor clock frequency was assumed to be 500 MHz (i.e., 500 MIPS). A coalescing write buffer with 8 entries, each of the size of the L1 cache lines, was provided between L1 and L2 caches to reduce processor stalls caused by writes. A read miss in both the L1 cache and the L1 write buffer stall the processor until the desired data was returned from the lower levels of the memory hierarchy. The processor and L1 cache were connected to a 256 KBytes 2-way set associative write-back and write-allocate L2 cache with a line size of 256 bytes via a 8-byte width data-path. The interleaved memory and directory module was assumed to support multiple read/write ports. On a memory access, the critical word of the block was modeled to be available in 50 ns. The successive words in the block followed in a pipelined fashion at the rate of 8 bytes per memory bus cycle. A directory cache was assumed to eliminate the stalls in the node controller caused by directory accesses.

In the node-network interface (NNI), the connections to the processor module and memory/directory module were assumed to be 8 bytes wide and were assumed to operate at 100 MHz. A coalescing transaction buffer of 8 entries was provided for resolving outstanding shared memory operations. Each entry was able to hold one L2 cache line and merge multiple L1 cache writes targeting the same L2 line. The machine supported the release consistency memory model [85, 88]. The synchronization protocol assumed in the system was the queue-based spin-on-local-copy protocol, similar to the one used in DASH [91]. The BTU was assumed to be organized as discussed in Section 4.5.

The four types of NNI designs were modeled as follows. The CBB-NNI design always transfers data at the memory block level. The PCF-NNI design transfers data at the sub-block level between the BTU and the processor module, main memory module, and the network interface. Any message must be completely constructed (at sender) or accumulated (at receiver) at the network interface before entering the network or the node controller. The CTD-NNI design transfers data at the block level between the BTU and the processor module, main memory module. The network interface initiates message injection as soon as the first sub-block is ready and can copy a sub-block data into the BTU as soon as it is received. The ICP-NNI design transfers data at sub-block level between all connections in the BTU and at the network interface. The default size for sub-block was 32 bytes. The network interface was assumed to take 4 memory bus cycles (40 ns) for preparing an outgoing message (header) and 8 cycles (80 ns) for error-checking and dispatching on an incoming message. The occupancies for different operations were assumed as follows. One bus cycle was spent in preprocessor and postprocessor stages each. In a system using a coherence controller, 6 bus cycles (60 ns) were taken to process a request/response, manipulate directory, and generate a new request/response; each invalidation (being sent after the first one) incurred 4 additional bus cycles (40 ns); and the bypassing took 2 bus cycles (20 ns).

The network switches were interconnected in a hypercube like the SGI Origin [88] (default was a 5D). The network switches used the dimensional order wormhole routing design and supported multiple virtual channels with a flit size of 64 bits (8 bytes). Each virtual channel had an input buffer of 32 flits (256 bytes) and an output buffer of 4 flits (32 bytes). Internally, each switch used the DAMQ input buffering scheme to eliminate the head-of-line (HOL) blocking problem [136, 49]. The switching pipeline was modeled based on SGI SPIDER switch [49], i.e., 2 switch cycles for link synchronization, 1 cycle for routing, and 1 cycle for moving a flit across the crossbar. The switch was assumed to operate at 100 MHz. Thus the link propagation time was assumed to be 1 switch cycle (10 ns).

4.6.2 Latency of Remote Memory Operations

To gain insight into the performance impact of alternative NNI designs on the latency of remote memory operations, let us examine the example remote read clean miss operation (as shown in Figs. 4.2 and 4.3) and the break down of its overall latency. For simplicity, for the time being, let us ignore all types of contention at the cache, memory, NNI, and network in this subsection. Table 4.1 shows the breakdown of the latency. It can be observed that the restart latencies of the CBB-NNI, PCF-NNI, CTD-NNI, and ICP-NNI designs are 916 processor cycles (1.83 μs), 611 processor cycles (1.22 μs), 591 processor cycles (1.18 μs), and 286 processor cycles (0.57 μs), respectively. Considering the block size difference, as we expected, the latency in the CBB-NNI design is compatible to that reported in the Wisconsin Typhoon [119] (1.5 μs); the PCF-NNI design latency is compatible to that reported in FLASH [85, 59] (1.11 μs); and the ICP-NNI design latency is compatible to that reported in the SGI Origin [88, 64] (0.57 μs). The table also shows that the back-to-back latency of the four designs are 1081 processor cycles (2.16 μs), 776 processor cycles (1.55 μs), 756 processor cycles (1.51 μs), and 451 processor cycles (0.90 μs), respectively. In other words, compared to the CBB-NNI design, the PCF-NNI, CTD-NNI, and ICP-NNI designs reduce the restart latency of the example remote memory operation by 33%, 35%, and 68%, respectively. The PCF-NNI, CTD-NNI, and ICP-NNI designs reduce the back-to-back latency of the example remote memory operation by 28%, 30%, and 58%, respectively. When contention is considered, it is reasonable to believe that the performance improvement will decrease. (In fact, we show and discuss some more complete results with contention in the next subsection.) Nevertheless, Table 4.1 shows that various NNI designs using sub-block pipelining techniques are quite effective in reducing the latency of remote memory operations.

[9]Not included in the best total which assumes data forwarding and early restart.

Steps	CBB Latency	PCF Latency	CTD Latency	ICP Latency
Requesting Processor Module				
Detect L1 & L2 cache miss	6	6	6	6
L2 bus transit	5	5	5	5
Requesting NNI				
Preprocessor	5	5	5	5
Coherence Controller	10	10	10	10
Postprocessor	5	5	5	5
Send request msg	20	20	20	20
Network				
Request msg transmission (16 bytes)	30	30	30	30
Home NNI				
Extract msg header	40	40	40	40
Preprocessor	5	5	5	5
Fetech data, modify directory	180	30	180	30
Postprocessor	5	5	5	5
Send reply msg	180	180	20	20
Network				
Reply msg transmission (272 bytes)	195	195	30	30
Requesting NNI				
Extract msg header	40	40	40	40
Preprocessor	5	5	5	5
Read msg data	180	25	180	25
Postprocessor[9]	5	5	5	5
Requesting Processor Module				
L2 bus transit	5	5	5	5
L1 & L2 Cache filling[9]	160	160	160	160
Total				
Restart	916	611	591	286
Back-to-back	1081	776	756	451

Table 4.1: Breakdown of the contentionless latency of a clean read-miss to a neighboring home node. Values are counts in 500 MHz processor cycles. CBB = CBB-NNI, PCF = PCF-NNI, CTD = CDT-NNI, and ICP = ICP-NNI.

4.6.3 Application Performance

In this section we describe the application benchmarks used in our evaluation and discuss the simulation results. We used six applications — FFT (256K points), MP3D (50K particles), Radix (1M keys, 1K radix, 1M max), Barnes (8K particles, 4 steps), LU (512×256 doubles, 8×8 blocks), and Water (512 molecules, 4 steps).

Performance Comparison of Pipelined NNI Designs

Figure 4.6 shows the overall execution times of the six benchmark applications in DSM systems using the four alternative NNI designs. The number on top of each bar indicates its height. These results were measured in simulations using the default system configuration as described in Section 4.6.1. For each application, the result of the CBB-NNI is used as the normalization basis. Results of the PCF-NNI, CTD-NNI, and ICP-NNI designs are normalized and shown in that order. The block size used in all experiments was 256 bytes. The size of sub-block used in all the three pipelined NNI designs was 32 bytes.

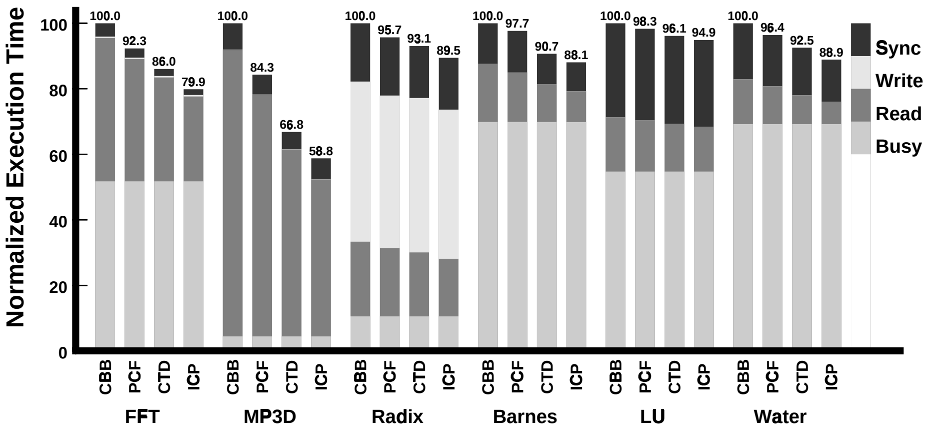


Figure 4.6: The performance of the four alternative node-network interface (NNI) designs for benchmark applications. CBB = CBB-NNI, PCF = PCF-NNI, CTD = CDT-NNI, and ICP = ICP-NNI.

The overall execution times are also broken down into four components. From bottom to top, these components are compute processor busy (Busy) time, memory read waiting (Read) time, memory write waiting (Write) time, and synchronization waiting (Sync) time. It is clear from the timing breakdowns shown in Fig. 4.6 that for the application and baseline system parameters we selected, Barnes, LU, and Water appear to be computation intensive. MP3D and Radix are communication intensive. FFT is evenly divided between computation and communication time. The communication time in MP3D is dominated by read waiting caused by true data flow dependence. The communication time in Radix is dominated by write waiting caused by false data sharing. LU has a significant amount of time spent in synchronization because of load imbalance at later execution phases. Except radix, the write waiting in each application is negligible because release consistency is used.

From Fig. 4.6, the following two important observations can be made. First, all the three alternative enhanced NNI designs improve the performance uniformly for all applications. In our experiments, the cut-though delivery (CTD) enhancement outperforms the partial cache-filling (PCF) enhancement. Integrating the two enhancements (ICP) outperforms either one alone. These trends remain consistent across all the applications even though the amount of reduction in the overall execution time varies. Overall, the PCF-NNI design improves the application performance ranging from 1.7% to 15.7%, with a typical value of about 3%. The CTD-NNI design improves the performance ranging from 3.1% to 33.3%, with a typical value of about 8%. The ICP-NNI design improves performance ranging from 5.1% in LU to 41.2% in MP3D in our experiments. It can also be observed from the breakdown that even with various resource contention, the reductions in memory waiting time by the three sub-block NNI designs are still quite significant.

Second, not surprisingly, the breakdown also shows that the enhancements studied are most effective for reducing read waiting time and less effective for reducing write waiting time. This can be explained as follows. In a DSM system supporting the release consistency model, coalescing write buffers are extensively used. This allows a processor to proceed regardless of write hit or miss under most circumstances. A write is stalled (i.e., write waiting) when all write buffers are busy. As in most systems with a write-back write-allocate cache, a write buffer is freed only after the entire requested block is written into the cache. Therefore, the important property of the sub-block technique which allows critical data forwarding and early restart becomes less effective.

In the next few subsections, we study the sensitivity of the performance benefit of the integrated NNI with respect to several important system parameters by varying these parameters one at a time.

Sensitivity to the Size of Sub-block

From earlier discussions, it is clear that the size of the sub-block is critical in the cost vs. performance tradeoff. Since sub-blocks are used in both the CTD and PCF schemes throughout the pipeline, the size of the sub-block should be a multiple of the size of micro-packet and of the size of L1 cache line[10]. Figure 4.7 shows the normalized execution time for the applications running on systems using the ICP-NNI design under the default system configuration with different sized sub-blocks. The sub-block size changes from 32 to 64 and to 128 bytes. The results are normalized to the execution times of the individual applications on the baseline configuration using the CBB-NNI design. It can be observed that as the size of the sub-block decreases, the performance benefit of the ICP-NNI design increases with a diminishing return. The reason for such a trend is the following. As the size of sub-block decreases, the granularity of the pipeline for resolving remote operations becomes finer. At the same time, the control overhead of each pipeline stage remains largely unchanged. Thus, more operation overlaps are exploited by smaller sub-blocks, leading to higher performance.

Sensitivity to the Size of Block

Figure 4.8 shows the normalized execution time for the applications running on systems using the ICP-NNI design under the default system configuration with different sized blocks. The sub-block size is fixed to 32 bytes. The block size changes from 64 to 128 and to 256 bytes. It is noted

[10]We assume that L2 cache uses sub-block technology.

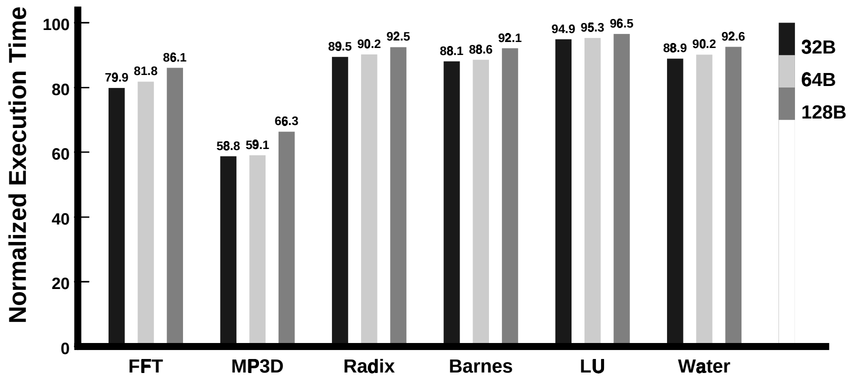


Figure 4.7: Sensitivity to the size of sub-block. The block size is fixed to 256 bytes. The sub-block size changes from 32 to 64 and to 128 bytes.

that each bar is normalized to the execution time of the same application on the corresponding CBB-NNI design using the same sized blocks. It can be observed that as the size of block increases, the performance benefit of the ICP-NNI design increases. For applications except Radix, the improvement margin also increases. This trend can be explained as follows. As the size of block increases, the amount of data transferred (and sub-operations) in a memory operation increases. The operational pipeline is kept full for a longer time period, leading to more operation overlaps and thus, higher performance. The exception in Radix is due to the significance of write waiting time induced by false sharing.

Sensitivity to the Occupancy of Coherence Controller

Figure 4.9 shows the normalized (against corresponding CBB-NNI design) execution time for the applications running on systems using the ICP-NNI design under the default system configuration with differing occupancies of coherence controller. The occupancy changes from 3 to 6 and to 12 bus cycles. It can be observed that for all applications except MP3D, the performance benefit of the ICP-NNI design is not very sensitive to the occupancy of coherence controller. This is largely caused by the separation of the data and control paths in the node controller. Using current technology, the occupancy of the CC is dominated by the data transfer latency of the BTU. The exception in MP3D at occupancy of 12 bus cycles may be caused by a program behavior change due to the nondeterministic algorithm used by this application.

Sensitivity to Cluster Size

Figure 4.10 shows the normalized (against corresponding CBB-NNI design) execution time for the applications running on systems using the ICP-NNI design under the default system configuration with differing small cluster sizes. By cluster size, we mean the number of processor modules connected to a single NNI. The cluster size is varied between 1, 2, and 4 in the experiments. At the same time, the network topology changes from 4D to 5D to 6D hypercube because the total number

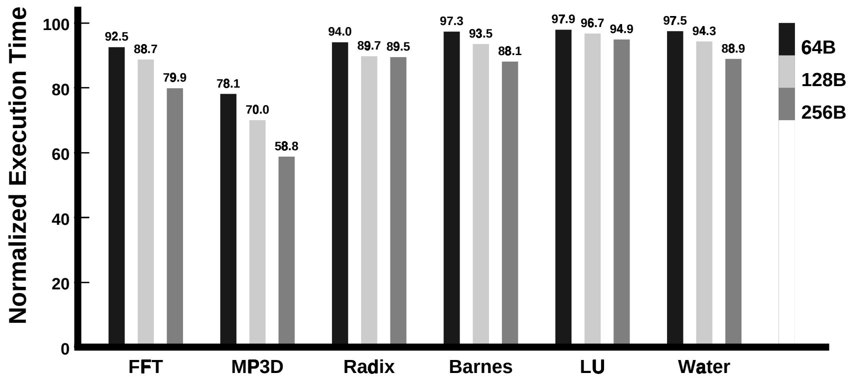


Figure 4.8: Sensitivity to the size of block. The sub-block size is fixed to 32 bytes. The block size changes from 64 to 128 and to 256 bytes.

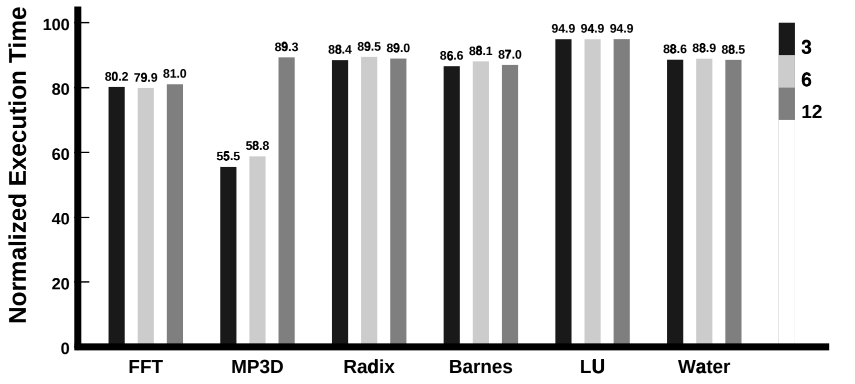


Figure 4.9: Sensitivity to the occupancy on the coherence controller. The occupancy changes from 3 to 6 and to 12 bus cycles.

of processors in the system is fixed at 64. It can be observed that for applications except Radix, the performance benefit of the ICP-NNI design is not sensitive to the size of the small cluster. This can be explained as follows. When the cluster size increases, the total number of clusters decreases. From the application viewpoint, the amount of intra-cluster communication increases while the amount of inter-cluster communication decreases. From the system viewpoint, the intra-cluster communication bandwidth increases while the inter-cluster communication bandwidth decreases. Such changes in program behavior and the system often match each other for most applications, especially with small cluster sizes. The exception in Radix occurs because false sharing introduces an unbalanced amount of inter-cluster communication and intra-cluster communication when the cluster size changes.

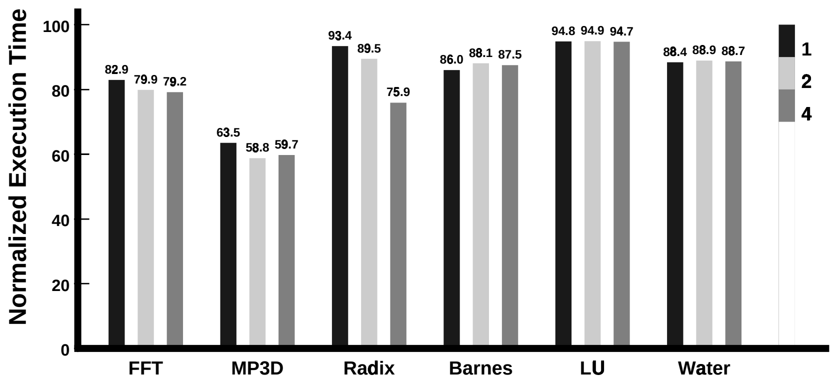


Figure 4.10: Sensitivity to cluster size. The number of compute processors connected to a single switch changes from 1 to 2 and to 4.

4.7 Related Work

Recently, in the context of network of workstations (NOW), two software/firmware techniques, similar to the PCF and CTD, have been shown to be quite successful in reducing communication latency. The first technique [69] uses two reply messages to resolve one page fault in network-based disk-caching. The first message contains a small sub-page with critical data in order to allow early processor restart. The second message contains the rest of the target page for keeping the semantics of page fault unchanged to the rest of the system. The other technique [149] applies cut-through delivery to eliminate the store-and-forward latency by pipelining packets through the adapter. Both software techniques focus on transferring large, page-sized data (8K bytes) between main memories of workstations. A somewhat similar mechanism called bulk transfer [57, 147] has also been proposed for DSM systems to pipeline and overlap the transfer of multiple cache blocks. However, its performance advantages seem to be limited to applications with many data/task migrations [58, 57]. The extra complexity of introducing bulk transfer mechanism into existing systems is also considerably high. Our work is different from the above two studies because: a) our targeting systems are hardware DSMs which possess totally different characteristics from NOWs and b) our proposed designs are completely hardware based.

4.8 Summary

The remote memory access latency, a bottleneck in DSM systems, can be divided into a number of components such as the time being spent for cache hierarchy propagation, memory directory manipulation, message sending and receiving, and message transmission inside the network, in addition to the response time of memory modules. These components are strongly affected by the node-network interface (NNI) design because different NNI designs may allow various sub-operations in cache propagation, memory directory manipulation, message sending and receiving, and message transmission to be pipelined and overlapped with one another.

In this chapter, we have proposed three new pipelined NNI designs, namely, the partial cache-filling NNI (PCF-NNI) design, the cut-through delivery NNI (CTD-NNI) design, and the integrated cut-through delivery and partial cache-filling NNI (ICP-NNI) design. These designs apply the pipelining idea to caching and messaging layers while maintaining coherence at the block level. We have studied the important design issues such as interlock signaling at various pipelining stages at the sub-block level. We have also presented in detail an implementation of the core circuitry, i.e., the block transfer unit (BTU), for our most sophisticated, efficient design.

Simulations have been performed to evaluate the effectiveness of the interface designs compared to a conventional one using six benchmark applications. Our results show that in our baseline configuration, compared to the CBB-NNI design, the PCF-NNI, CTD-NNI, and ICP-NNI designs can improve application performance by up to 15%, 33%, and 41%, respectively. For most applications, the typical performance improvement is about 3%, 8%, and 11%, respectively. The sensitivity of such improvements to several important system parameters such as the size of sub-block, the size of block, the occupancy of coherence controller, and the size of cluster are also examined through simulation experiments. The results show that the performance benefit increases as the size of sub-block decreases or as the size of block increases. The performance benefit is also shown to be not so sensitive to the occupancy of the coherence controller or the size of cluster for most applications.

Overall, this study demonstrates that appropriate pipelined designs require modest hardware modifications to existing node-network interfaces currently being used in DSM systems and can improve the overall system performance significantly.

CHAPTER 5

INCORPORATING MULTIPLE-PATH NETWORKS BY USING EFFICIENT NODE-NETWORK INTERFACE

A pronounced component of the remote memory access latency is the network delay, as highlighted by the results in Chapters 2 and 3. One effective means to alleviate network congestion and increase throughput, thus leading to lower network delay, is to use multiple paths in transferring messages between a pair of nodes in such systems. Modern switches/routers like SGI SPIDER [49] and HAL PRC [144] use performance-enhancing mechanisms such as cut-through switching, multiple virtual channels, and efficient buffering extensively. As a result, multiple paths exist between any given pair of nodes, either implemented physically [10] or supported logically via virtual channels [144, 49], in most existing CC-NUMA systems. One side effect of using multiple paths in transferring messages is that messages from a source may arrive at a destination in different orders. This is known as the pairwise *out-of-order (OoO) message arrival* problem. Although exceptions exist [88], many directory-based cache coherence protocols [138] designed for DSM systems require pairwise in-order arrival.

To exploit the advantages of a multiple-path network in a DSM system, architects currently use two alternative strategies. The first one, used in the SGI Origin, is to enhance the cache coherence protocol with more intelligence so that it can detect and resolve all critical out-of-order (OoO) message arrivals. The main drawback of this strategy is the high complexity in the design, verification, and implementation of the resulting coherence protocol [144]. The second strategy, used in the HAL Mercury, is to enhance the network interface with reordering capability to ensure that all pairwise messages seen by the coherence protocol are in-order (total FIFO channel). The drawback of this strategy is a noticeable increase in both complexity and overhead (delay) at the network interface.

In this chapter, we propose a new strategy for exploiting the benefits of multiple-path networks in a DSM system using *block correlated FIFO channels*. This new strategy detects all potential coherence sensitive (pairwise) race conditions and prevents them from occurring. It allows the use of both an in-order (FIFO) cache coherence protocol and a simple network interface. We also present an efficient implementation of this strategy based on current technology. To quantitatively evaluate the performance of our proposed strategy, we have performed simulation experiments using practical system configurations and benchmark applications. For most applications, the results show that the new strategy is less than 1% slower than the SGI Origin approach and up to 40% *faster* than the HAL Mercury approach. This demonstrates that DSM systems using our proposed strategy can provide very competitive performance at a much lower cost.

This chapter is organized as follows. Section 5.1 reviews issues relevant to communication in DSM systems. Sections 5.2 and 5.3 describe the intelligent cache coherence protocol strategy and the total FIFO channel strategy, respectively. In Section 5.4, we propose the block correlated FIFO channel strategy and present an efficient implementation. Section 5.5 describes our methodology for performance evaluation and discusses the results, and finally in Section 5.6 we draw the conclusions.

5.1 Relevant Design Issues

In this section, we discuss several key issues relevant to incorporating multiple-path networks in CC-NUMA systems.

Complexity of Cache Coherence Protocol: The cache coherence protocol of a CC-NUMA system ensures that a *read* to any memory location always gets the content from the latest *write* which the reading processor is aware of. Various cache coherence protocols differ on the types of states and messages being used. The complexity of a protocol increases with the number of state types and message types. Although fast high coverage techniques exist [47], complete coverage verification methods rely on exhausting the reachability of finite state machines or their equivalents. Until now, verification of a highly complex coherence protocol remains a monumental task for computer architects [115]. Complex protocols also introduce larger overhead on the critical path, leading to longer occupancy for each state transition in the protocols.

OoO Arrival and OoO Event: As mentioned earlier, modern high-performance networks support multiple virtual channels [144, 49] or multiple physical routes [10] between a pair of nodes to increase bandwidth and/or reduce congestion. When virtual channels are supported, a path can be viewed as a chain of virtual channels from the source node to the destination node. The latency of a message varies depending on the length of the message, the length of the used path, and congestion along the path being used. Therefore, it is possible that between a source-destination pair, a message using one path reaches its destination sooner than a previously sent message using another path, as shown in Fig. 5.1. Such a bypassing scenario is commonly known as an *out-of-order (OoO) arrival* of messages [27]. A message may bypass more than one messages. For each of the bypassed messages, exactly one *out-of-order (OoO) event* occurs.

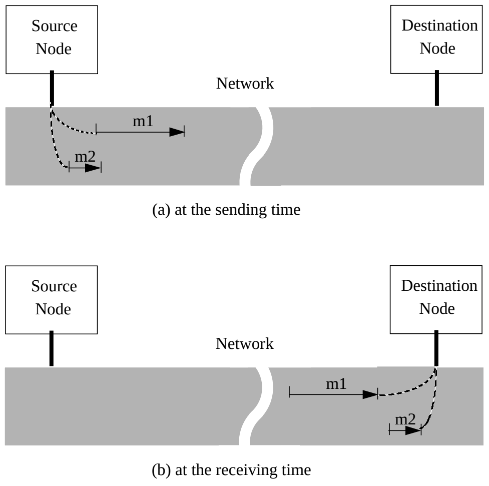


Figure 5.1: An example of out-of-order (OoO) arrivals of messages between a source and destination pair.

Example Race Condition: In a CC-NUMA system, many types of race conditions may be caused by OoO events. Figure 5.2 shows one example. Node N_1 sends a write-miss request ($Rxq(1)$) to block B_1 whose home is node N_2 and whose current owner is node N_3. Once the request Rxq arrives at N_2, a flush request ($Fsh(2)$) is sent by N_2 to N_3. Assume right before the arrival of Fsh at N_3, the (dirty) copy of B_1 is displaced and sent back ($Wbk(3a)$) to N_2. Once Fsh reaches N_3, a reply ($Frp(3b)$) is sent by N_3 back to N_2 as the acknowledgment. Either message Wbk or Frp may reach N_2 first (i.e., a race condition) in a network allowing OoO arrivals. This uncertainty becomes a problem for many cache coherence protocols because after the Frp arrives, reply $Rxp(4)$ normally takes a copy from the main memory at N_2 which may or may not have been updated by the Wbk. Such a (coherence-sensitive) race condition must be detected and resolved properly.

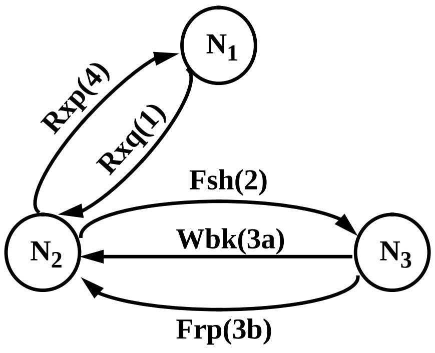


Figure 5.2: An example race condition caused by an OoO event.

For resolving race conditions, in the following three sections, we examine two existing strategies and their drawbacks, and propose a new efficient strategy.

5.2 Intelligent Cache Coherence Protocol Strategy

The basic idea behind the intelligent cache coherence protocol (I-CC) strategy is to build enough intelligence into the cache coherence controller so that it can detect and revolve all race conditions. Typically, the network in the target system is logically separated into request and response networks. Within each logical network, complete freedom is granted in routing a message from its source to destination. Figure 5.3(a) illustrates the network connecting two arbitrary nodes in such a system. Four parallel paths are shown, with two paths in each of the two logical networks. This strategy can use simple and efficient network interfaces with a selection function which decides whether the request or the response network should be used for transferring a particular message.

In this strategy, the coherence protocol must detect a race condition on-the-fly. The popular approach is to denote the relevant history of operations on each memory block using different directory/cache states. By checking the current state and the type of message, all race conditions can be detected successfully. Once a 'spoiling' message arrives before the expected 'authoritative' message (e.g., message 3b bypasses 3a in Fig. 5.2), the spoiler (message 3b) can be buffered or NAKed. Other actions like reverting to a simpler protocol, or combining transactions can also be taken to resolve it depending on different performance optimization goals [27]. For a more detailed

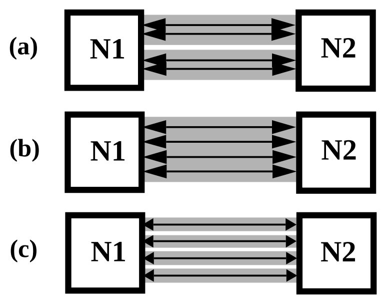


Figure 5.3: The networks between two nodes in a system using: (a) the I-CC, (b) the T-FIFO, and (c) the C-FIFO strategies.

description on such cache coherence protocols, readers can refer to [85, 88, 27]. It is to be noted that OoO events across memory blocks do not cause any race condition by virtue of synchronization mechanisms (e.g., release consistency [85, 27]) used in the CC-NUMA systems.

In order to get a rough idea on the complexity of an intelligent cache coherence protocol, let us examine the protocol used in the SGI Origin [88] as an example. As reported in [27], there are 15 requests (including invalidations and interventions) and 39 responses, 7 directory states[11], and more than 7 cache states used in this protocol[12]. Compared to cache coherence protocols that are used in the next two strategies, this complexity is significantly higher. This is the main drawback of the I-CC strategy.

Overall, CC-NUMA systems using the I-CC strategy are expected to deliver high performance because the benefits of the multiple-path network are aggressively exploited. Unless race conditions have actually occurred, forward progress in applications can always be made.

5.3 Total FIFO Channel Strategy

The philosophy of the total FIFO channel (T-FIFO) strategy [144] is to build a powerful network interface (NI) to shield the effect of OoO arrivals from the rest of the node. The network can use any path for transferring a message from its source to destination. The in-order message arrival property, ensured by the NI, effectively eliminates all race conditions for cache coherence protocols, allowing many simple and efficient (in-order based) cache coherence protocols (such as those discussed in [138, 27, 115]) to be used. Figure 5.3(b) illustrates an example network with four parallel paths connecting two arbitrary nodes in a system using the T-FIFO strategy. For a race condition as shown in Fig. 5.2, this strategy ensures that the cache coherence protocol always sees message 3a before message 3b regardless of which one has arrived at the NI of node N_2 first.

The T-FIFO strategy demands two key functions from the NI: a) to detect all occurrences of OoO arrivals and b) to remedy every such occurrence into a set of in-order arrivals. A representative design is to use a sliding window protocol enhanced with reordering capability. Figure 5.4 shows the pseudo-code description of such an NI. The code is self-explaining. Detailed description on

[11]Including one state for efficient page migration.

[12]Many optimizations targeted towards better performance have been incorporated into the SGI Origin coherence protocol. Otherwise, these numbers may be slightly smaller.

the NI design can be found in [38]. The main enhancement contains (pairwise) sequence number manipulation for OoO arrival detection and buffer management for restoring message order.

```
Sending:                          Receiving:
  gen-seqno(msg) /* r */            receive(msg)
  construct(msg)                    if (out-of-seq(msg)) /* r */
  send(msg)                           reorder-defer(msg) /* 2r */
                                    else { dispatch(msg)
                                      reorder-dispatch()} /* f(r) */
```

Figure 5.4: Pseudo-code description of the NI used in the total FIFO channel strategy. 'Dispatch' means delivering 'msg' to the cache coherence controller. The comments show the time delays for reordering operations.

Typically, OoO arrivals occur rarely in a system. Not every OoO arrival causes a race condition. However, the overhead for detecting OoO arrivals slows down every network transaction. This is the main drawback of the T-FIFO strategy.

5.4 Block Correlated FIFO Channel Strategy

In this section, we propose a new strategy which uses simpler cache coherence controllers and inexpensive NIs. We first develop several key concepts used in this strategy and then present an efficient implementation.

5.4.1 Eliminating Race Conditions

A careful examination can reveal the fact that in a DSM system, the services of memory operations targeting to the same memory block must be serialized. For convenience of discussion, let us define two memory operations to be *block correlated memory operations* if their target memory blocks are the same. We can similarly define *block correlated messages* and *block correlated network transactions.* A race condition is a scenario in which the serialization order of two block correlated memory operations may be observed differently by the involved nodes because of OoO events. Such an OoO event can be defined as a *block correlated OoO event.* All race conditions are caused by block correlated OoO events. Because each memory operation is serviced by a chain of network transactions, to eliminate all race conditions, a necessary and sufficient condition is that all block correlated network transactions must maintain the in-order property while non-correlated network transactions can proceed arbitrarily[13]. The important performance implications of this condition will become clear in Section 5.5.

[13]When the order between non-correlated network transactions is critical, it will be ensured by the synchronization mechanism of the DSM system.

5.4.2 Block Correlated FIFO Channel

A *block correlated FIFO channel (BCFC)* can be defined as the abstraction of a FIFO channel used for transferring messages relating to a particular memory block. Every memory block has a distinct block correlated FIFO channel (BCFC) associated with it between each pair of nodes. The number of BCFCs between two given nodes is equal to the total number of blocks in the shared address space. Such BCFCs put no restriction on the arrival order of non-block correlated messages because they travel in separate BCFCs. Therefore, BCFCs provide a perfect mechanism for enforcing the necessary and sufficient condition for eliminating race conditions discussed in the previous subsection.

5.4.3 The Strategy and Implementation

We propose to build DSM systems using simple (in-order based) cache coherence protocols and block correlated FIFO channels (BCFCs). We call such a strategy the *block correlated FIFO channel (C-FIFO) strategy.* This strategy prevents race conditions, such as the one shown in Fig. 5.2, from ever occurring because messages 3a and 3b are block correlated and thus message 3b will never bypass message 3a.

At a first glance, it seems impractical to implement BCFCs in a system because of the large number of BCFCs required. Most current generation networks, such as those used in IBM SP [10], HAL Mercury [144], and SGI Origin [88], support only a small number of (physical or logical) parallel paths (FIFO channels). However, if we map (or collapse) multiple BCFCs onto a single path, the path can be viewed as *block correlated* to a particular *set* of memory blocks. Using this mapping idea, a simple implementation of the C-FIFO strategy becomes realizable. For efficiency purposes, the mapping from BCFCs to parallel paths should incur a minimum overhead. An ideal mapping function is the 'modulo' operation on the block address associated with each BCFC. Since the number of parallel paths is typically small in a system, selecting a path for transferring a message based on the few least significant bits of the block address associated with the message can be done easily. Figure 5.3(c) illustrates four block correlated paths (i.e., four logical networks) connecting two nodes in a system using the C-FIFO strategy.

It is clear that the C-FIFO strategy can use cache coherence protocols as simple as those used in the T-FIFO strategy and NIs as efficient as those used in the I-CC strategy. Figure 5.5 shows a qualitative comparison of these strategies. In the next section, we compare and discuss the performance of these three strategies, quantitatively.

5.5 Performance Evaluation

This section presents our simulation-based performance evaluation methodology, simulation results, and discussions.

5.5.1 Methodology

The hardware cache coherent multiprocessor we simulated had a generic DSM architecture similar to the FLASH system [85], supporting release consistency. Each processor was modeled as a 300 MHz single-issue superscaler, supported by a 8 KB direct-mapped write-through L1 D-cache (32 bytes per line), a perfect I-cache, and a 128 KB 2-way associative write-back L2 cache (256 bytes per line). Coalescing write buffers were provided for both L1 and L2 caches (8 entries each). A read miss in L1 cache and L1 write buffer stalled the processor. The memory module

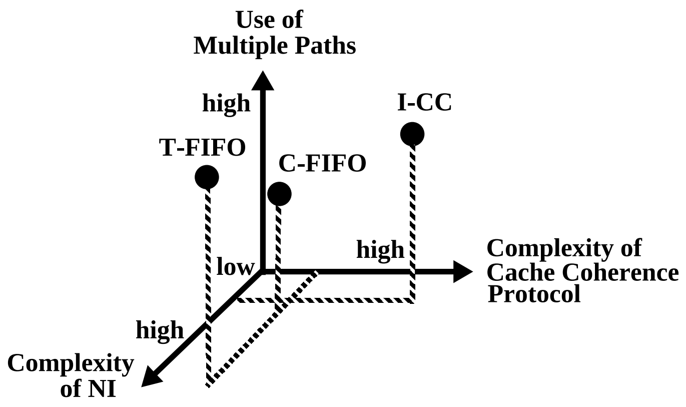


Figure 5.5: A qualitative comparison of alternative strategies for incorporating multiple-path networks in DSM systems along three fundamental design axes. The 'Use of Multiple Paths' axis indicates the flexibility for transferring a given message.

was assumed to support multiple read/write ports, with a 66 ns response time per memory access. A directory cache was assumed to eliminate any directory access stall. The bandwidth between the node controller and each module (e.g., processor and memory modules) was assumed to be 1.6 GB/sec.

The system had 64 processing nodes connected via a 5D hypercube with 2 nodes per switch (like the SGI Origin [88]). The network used the dimensional order (from low to high) wormhole routing scheme [35, 32, 27] and supported 2 virtual channels. Each virtual channel was assumed to be 64 bits wide and have an input buffer of 256 bytes and an output buffer of 32 bytes. The network switch was assumed to operate at 100 MHz. The link synchronization time, routing time, and time to cross the crossbar was assumed to be 20 ns, 10 ns, and 10 ns, respectively. The link propagation time was assumed to be 10 ns.

Two compatible cache coherence protocols originally presented and verified in [115] were used, with extensions for release consistency. The non-FIFO coherence protocol was used for the I-CC strategy, while the FIFO coherence protocol was used for the T-FIFO and the C-FIFO strategies. The occupancy on cache coherence controller for each network transaction was assumed to be mostly 46.2 ns (14 processor cycles), with additional 16.5 ns (5 processor cycles) for each invalidation message. The equal occupancy assumption might favor the I-CC strategy because of the higher complexity of its coherence protocol. However, based on our simulation results, this bias does not affect the overall conclusions of this study.

The basic network interface (NI), as used in the I-CC strategy, was assumed to take 40 ns for sending or receiving a message. For the C-FIFO strategy, extra 10 ns was assumed for selecting a path. For the T-FIFO strategy, the reordering capability was modeled as shown in Fig. 5.4. For simplicity, a delay of r was assumed for performing a load-modify-store operation on a sequence number and $2r$ for depositing or removing an OoO message at the NI. Table 5.1 summarizes the default memory system parameters used in our simulations.

In the simulation experiments, we considered five configurations: one using the I-CC strategy, one using the C-FIFO strategy, and three using the T-FIFO strategy (denoted as T-FIFO-1, T-FIFO-2, and T-FIFO-3). The T-FIFO-1, T-FIFO-2, and T-FIFO-3 configurations differed only in

Memory Hierarchy Parameters	
Processor frequency	300MHz
L1 cache access	1 proc cycle
L1 cache line size	32 bytes
L1 cache set associativity	1
L1 cache size per node	8 Kbytes
L1 coalescing write buffers	8
Data-path width between L1 and L2	8 bytes
L1 cache line fill time	6 proc cycles
L2 cache access	2 proc cycle
L2 cache line size (L)	256 bytes
L2 cache set associativity	2
L2 cache size per node	128 Kbytes
L2 coalescing write buffers	8
Memory bus frequency	100MHz
Memory response delay	20 proc cycle
Memory bus width (W)	16 bytes per mem cycle
L2 cache line fill time	20 proc cycles + L/W mem cycles
Memory block access time	20 proc cycles + L/W mem cycles
Coherence Controller Occupancy	
Directory check	7 proc cycles
Directory check and update	14 proc cycles
Invalidation request	5 proc cycles each
Cache intervention	3 proc cycles
Network Interface Parameters	
Outgoing constructing	4 mem cycles
Incoming dispatching	4 mem cycles
Reordering overhead parameter	r mem cycles
Control message size	16 bytes
Data message size	16+L bytes
Network Parameters	
Network frequency	100MHz
Channel width / Flit size	8 bytes
Link Propagation	1 net cycle
Router switch delay	1 net cycle
Routing time	1 net cycles
Link synchronization time	2 net cycles
Physical network	1
Virtual channels	2

Table 5.1: Default system parameters used in the simulation.

the value of r, which was assumed to be 100 ns, 200 ns, and 300 ns, respectively, corresponding to aggressive, intermediate, and conservative implementations. In the I-CC configuration, one virtual channel was used for transferring requests and the other for responses. In the C-FIFO configuration, one virtual channel was used for transferring messages related to even addressed blocks and the other for odd ones. In the T-FIFO-1, T-FIFO-2, and T-FIFO-3 configurations, both virtual channels were used with no distinction.

We used six applications — FFT (64K points), MP3D (50K particles), Radix (1M keys, 1K radix), Barnes (8K particles, 4 steps), LU (512 by 512 matrix), and Water (512 molecules, 4 steps) — in our simulation evaluations.

5.5.2 Results and Discussions

In this section, we evaluate the performance of DSM systems using the I-CC, T-FIFO, and C-FIFO strategies. We study the overall execution times of applications, the characteristics of network transactions, the characteristics of block correlated OoO arrivals and OoO events, and the impact of several key system parameters.

Overall Results

The overall execution times of benchmark applications on different system configurations are shown in Fig. 5.6. All times are normalized to that of the I-CC configuration. The times are further broken down into four components: the CPU computation busy time (Busy), the memory read waiting time (Read), the memory write waiting time (Write), and the synchronization waiting time (Sync). It can be observed that the C-FIFO configuration always delivers either the best or very close to the best performance among the five evaluated configurations. The actual performance difference between the C-FIFO configuration and the best one varies across applications from 0% (in MP3D) to 2.7% (in Radix). For most applications, this difference is less than 1%. With the simplicity at the NI level and at the cache coherence controller level, such a performance makes the C-FIFO strategy very attractive.

For all applications except Radix, the performance trend of the five configurations is the same. Namely, the configurations using either the I-CC strategy or the C-FIFO strategy outperform those using the T-FIFO strategy. The performance of the I-CC configuration (strategy) and that of the C-FIFO configuration (strategy) are very comparable. Among the configurations using total FIFO channels, the performance decreases for all applications as the reordering overhead (r) increases, as expected.

From the timing breakdowns, it can be observed that the CPU computation busy time remains almost constant across all configurations in every application. This is expected because the configurations (thus the strategies) target to reduce various waiting times due to communication, not the computation busy time. Two facts can be easily observed from the breakdowns. First, the computation to communication ratios are within typical ranges for each individual application, consistent with results reported by other research [85, 1]. Second, for all applications except Radix, the write waiting time is negligible. This correlates to earlier research on release consistency [27].

Characteristics of Network Transactions

Figure 5.7 shows the average latency of network transaction on the five evaluated configurations during the execution of applications. For most applications, the latency is significantly higher in the T-FIFO strategy than those in the I-CC or C-FIFO strategy. This shows that a noticeable

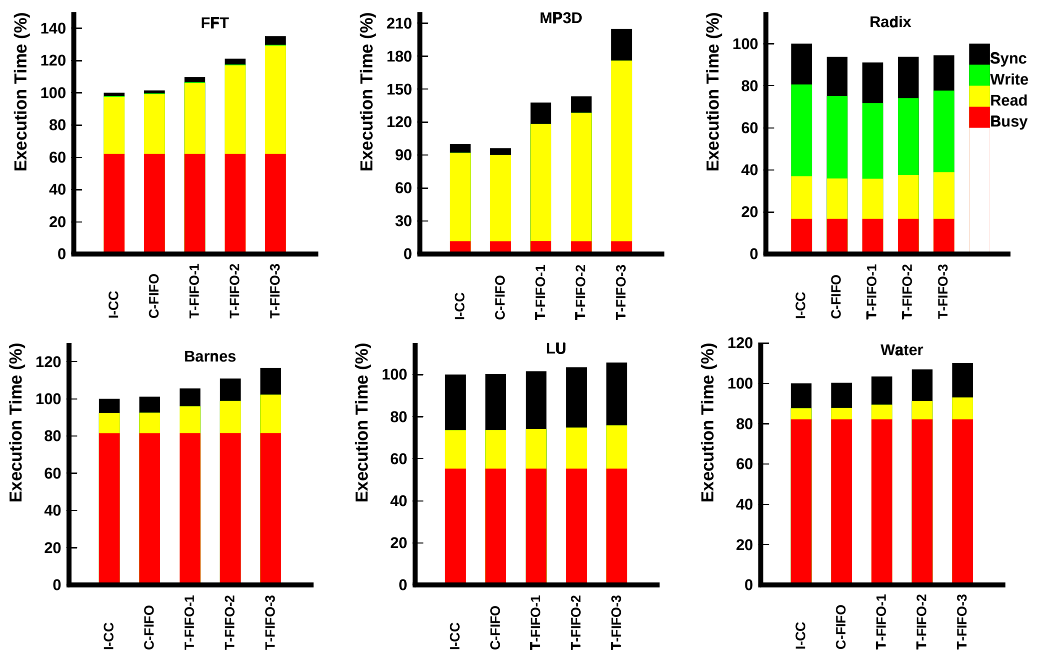


Figure 5.6: The overall execution times of benchmark applications on different configurations.

portion of the overhead incurred by the message reordering at the network interface (NI) lies on the critical path of every network transaction, especially the uncontended network transactions.

It can be observed that a strong correlation exists between Figs. 5.6 and 5.7. This is caused by two reasons: a) the computation remains almost same across configurations; and b) the total number of network transactions changes marginally across configurations, as shown in Table 5.2. This shows that the average latency of a network transaction has a strong impact on the overall performance of a CC-NUMA system. Any new technique for reducing this metric can potentially improve the overall system performance significantly.

	FFT	MP3D	Radix	Barnes	LU	Water
I-CC	294	5,344	5,862	1,585	521	612
C-FIFO	290	5,091	5,857	1,572	511	612
T-FIFO-1	292	5,169	5,882	1,604	496	610
T-FIFO-2	292	4,823	5,890	1,560	484	610
T-FIFO-3	292	4,741	5,870	1,566	477	610

Table 5.2: Total number of network transactions (in thousands).

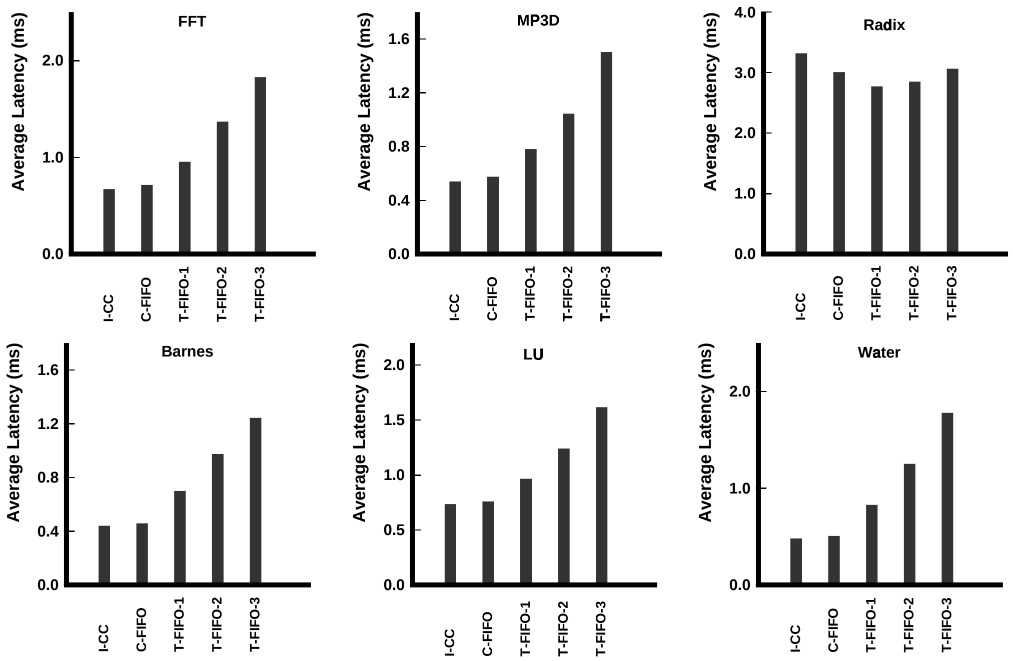


Figure 5.7: The average latency of a network transaction on different configurations.

Now, let us examine the performance results of Radix in Fig. 5.6. As shown in the figure, the C-FIFO configuration still provides the second best performance. However, the I-CC configuration is surprisingly outperformed by all other configurations. This phenomenon can be explained by the usage of the parallel virtual channels in different configurations. In Radix, especially at the permutation phase when the local histograms are merged into the global histogram in the earlier iterations, multiple writers and false sharing generates bursty heavy network traffic. This causes temporarily congestion in the network. In the T-FIFO-1, T-FIFO-2, and T-FIFO-3 configurations, the two virtual channels were used equivalently for transferring any messages. To its contrary, in the I-CC configuration, one virtual channel was dedicated to transferring request messages, the other to transferring response messages. Due to the imbalance between the request and response traffic, the virtual channels and thus network bandwidth in the latter configuration are not used as effectively as those in the former when the network is congested. However, in the C-FIFO configuration, the usage of the virtual channels and thus network bandwidth is improved to a certain extent depending on the temporal distribution of the block addresses. This result indicates that the T-FIFO and C-FIFO strategies can adjust better than the I-CC strategy when severe network congestion occurs.

Characteristics of OoO Arrivals and OoO Events

To gain more insights into the severeness of penalty on the average latency of network transaction exerted by total FIFO channels, we examined the characteristics of pairwise out-of-order (OoO) arrivals and OoO events in the most aggressive T-FIFO-1 configuration. The T-FIFO-2 and T-FIFO-3 configurations cause worse penalties. Table 5.3 shows the rate of OoO arrivals between two nodes, the average number of OoO events generated per OoO arrival, the average rate of block correlated OoO events per OoO event, and the average number of block correlated OoO events per message arrival. It can be observed that the average number of block correlated OoO events per message arrival (i.e., the erroneous outcome of a race condition) is a very small number, in the order of 10^{-2} or less. Our experimental data for the T-FIFO-2 and T-FIFO-3 configurations also showed that this number was even smaller. Such a small value indicates that the overhead incurred on every message transmission at the network interface (NI) in the T-FIFO strategy over-kills system performance.

	FFT	MP3D	Radix	Barnes	LU	Water
OoO Arrival Rate	1.91%	1.95%	2.65%	0.76%	3.49%	1.03%
OoO Ev/OoO Arr	1.0002	1.0006	1.0129	1.0000	1.0000	1.0002
OoO Haz/OoO Ev	66.65%	0.10%	5.65%	6.55%	12.55%	0.01%
OoO Hazard/Msg	1.27e-2	1.95e-5	1.50e-3	4.98e-4	4.38e-3	1.03e-6

Table 5.3: Summary of out-of-order (OoO) messages per pair of processing nodes in a total FIFO channel system (T-FIFO-1 configuration).

The above evaluations were based on a specific set of implementations. To ensure that the conclusions are not limited to certain implementations, we also studied the impact of several key design parameters relevant to communication. In the following subsection, we present results on the impact of the L2 cache designs and the network topology.

5.5.3 Impact of Cache Design Parameters

In this subsection, we study the impact of the L2 cache line size and the L2 cache size on the performance trend of the three strategies.

Impact of Smaller L2 Cache Line Size

It is well known that varying the cache line size of a given cache has the bath-tub effect on the overall execution time. A smaller L2 cache line reduces the average latency of network transaction and alleviates false sharing between nodes. On the other hand, the amortized cost per network transaction is higher. The increased misses at the L2 cache generate more network transactions. Figure 5.8 shows the execution time breakdowns of the I-CC, the C-FIFO, and the T-FIFO-2 configurations with a L2 cache line of 128 bytes. Compared to the corresponding results with a line size of 256 bytes, the overall performance improves in all three configurations for each application. Indirectly, this trend can be observed from Figs. 5.6 and 5.8 based on two facts: a) the absolute CPU computation busy times barely changed for each application in all our experiments; and b) the relative percentages of the CPU computation busy times increase from Fig. 5.6 to Fig. 5.8. It is to be noted that both the SGI Origin [88] and the HAL Mercury [144] systems use a L2 cache line of 128 bytes. Interestingly, with this cache line size, the performance gap among different configurations, especially the improvement of the C-FIFO configuration over the T-FIFO configuration, increases. This is because the reordering overhead becomes more prominent under reduced overall execution time.

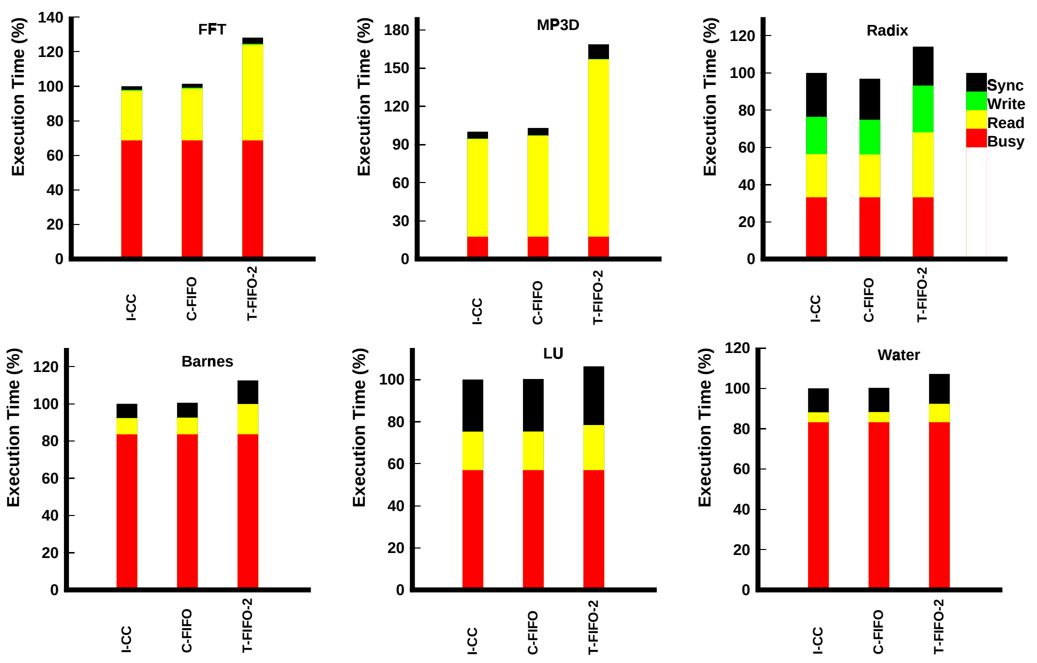


Figure 5.8: Impact of smaller cache line (128 bytes) on the overall execution times of benchmark applications on three configurations.

Impact of Larger L2 Cache Size

Let us now examine the case where each processing node has a larger L2 cache. A larger L2 cache helps to reduce capacity misses and thus potentially decreases the communication to computation ratio. Figure 5.9 shows the results of the three configurations with 256 Kbytes L2 caches (twice of the size used in experiments shown in Fig. 5.6). As expected, the overall performance improves in all three configurations for each application as the cache size increases. However, the performance trend remains unchanged (compared to Fig. 5.6) among all three strategies while the performance gap grows. This again attributes to the more prominent role of the communication subsystem on the overall performance.

5.5.4 Impact of Network Topology

We have also studied the impact of network topology on the performance improvement. We first examined the case where more processing nodes are connected to the same switch. Then, we examined the case where network bisection bandwidth decreases.

Impact of Larger Cluster Size

In general, a larger cluster increases the amount of intra-switch communication and reduces inter-switch communication. Figure 5.10 shows the results from systems consisting of 16 switches connected in a 4D hypercube topology with each switch connecting to 4 processing nodes (thus a total of 64 processing nodes). It is noted that each switch in the system is an 8-port bidirectional

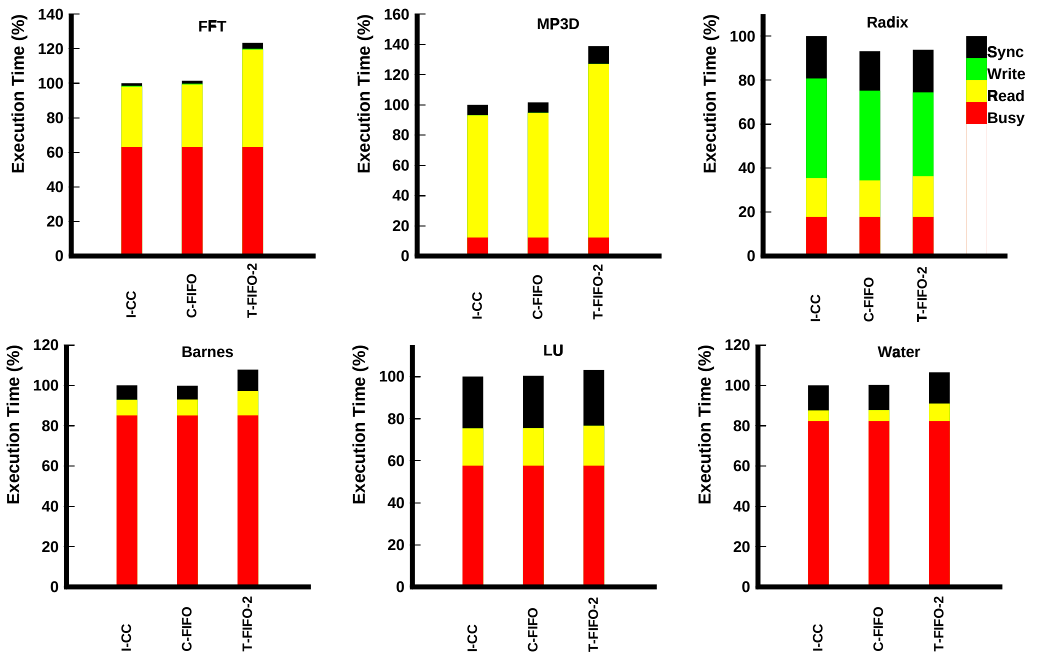


Figure 5.9: Impact of larger cache size per node (256K) on the overall execution times of benchmark applications on three configurations.

crossbar. In the experiment, the link bandwidth was maintained as the same as the default system. Due to the larger cluster size, the total number of switches decreased. As a result, the actual bisection bandwidth of the network also decreased. It can be observed that the overall performance of each application decreases (compared to Fig. 5.6) in all three configurations as cluster size increases. Interestingly, the performance gap among the configurations changes depending on the application. This gap increased in MP3D; barely changed in Radix, Barnes, LU, and Water; and decreased in FFT. However, the basic trend of performance among the three strategies remained unchanged from Fig. 5.6.

Impact of Smaller Network Bisection Bandwidth

Under the constraint of fixed number of nodes, as the dimensionality of a k-ary n-cube network decreases, the bisection bandwidth decreases, the diameter increases, and therefore the impact of network congestion increases. The main difference between the experiments in this section and the previous section is that the amount of intra-switch and inter-switch communication remains unchanged. Figures 5.11 and 5.12 show the results from systems containing 32 switches connected by a 3D and 2D mesh, respectively. In all experiments, each switch connects to 2 processing nodes, the same as in our base configuration. It is noted that the number of bidirectional ports supported by a switch changes as the connectivity decreases. It can be observed that the overall performance of each application decreases significantly in all three configurations as topology changes from 5D (Fig. 5.6) to 3D (Fig. 5.11) to 2D (Fig. 5.12). At the same time, the performance gap among the configurations increases for Radix and decreases for all other applications. Both the increase and

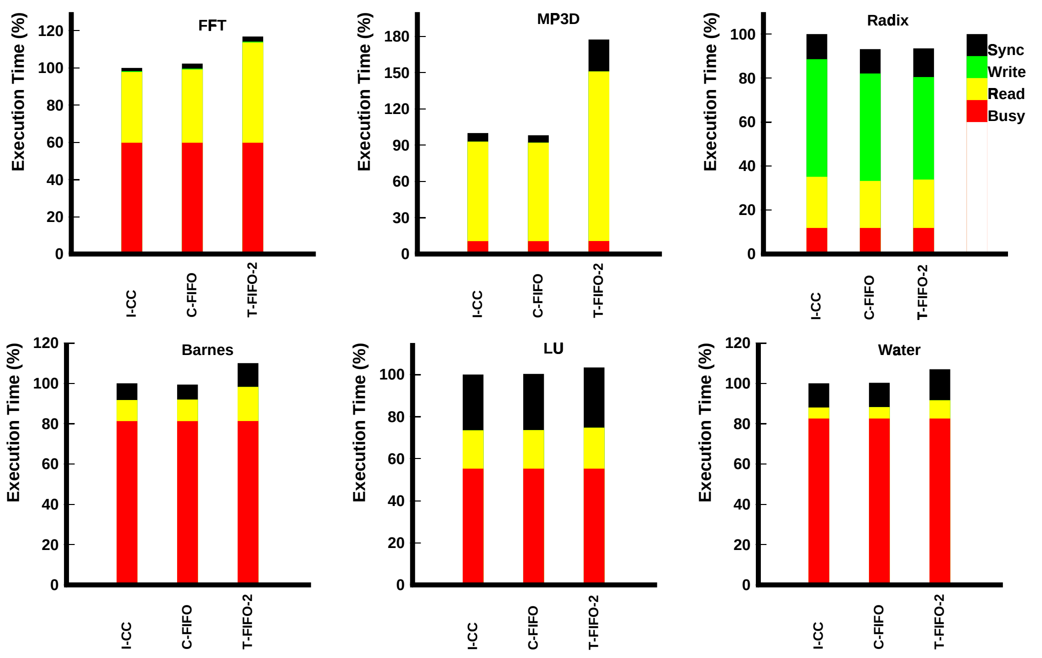


Figure 5.10: Impact of larger cluster size (4 processing nodes per switch) on the overall execution times of benchmark applications on three configurations.

decrease of the gap can be attributed to the growing prominence of congestion in the network, as explained in subsection 5.5.2. Nevertheless, our conclusion about the overall performance of the three strategies still remains unchanged.

5.6 Summary

In this chapter, we have proposed a new, block correlated FIFO channel (C-FIFO) strategy for incorporating multiple-path networks in scalable DSM systems. This new strategy combines the advantages and avoids the drawbacks of two existing strategies, i.e., the intelligent cache coherence protocol (I-CC) strategy and the total FIFO channel (T-FIFO) strategy. An efficient implementation of this new strategy using current technology has also been presented. Detailed performance evaluations demonstrate that for most applications, our proposed C-FIFO strategy outperforms the T-FIFO strategy by a factor of up to 40% and performs almost equal to the I-CC strategy at a much lower cost.

This study shows that not all network transactions in DSM systems are equally important at a given time. The effective latency of network transactions which can contribute to forward progress in applications is crucial for the overall system performance. With the simplicity at the cache coherence controller level and at the network interface level, the C-FIFO strategy offers a significant cost-performance advantage over the existing strategies. Current and future generation DSM systems can therefore benefit significantly by using this strategy.

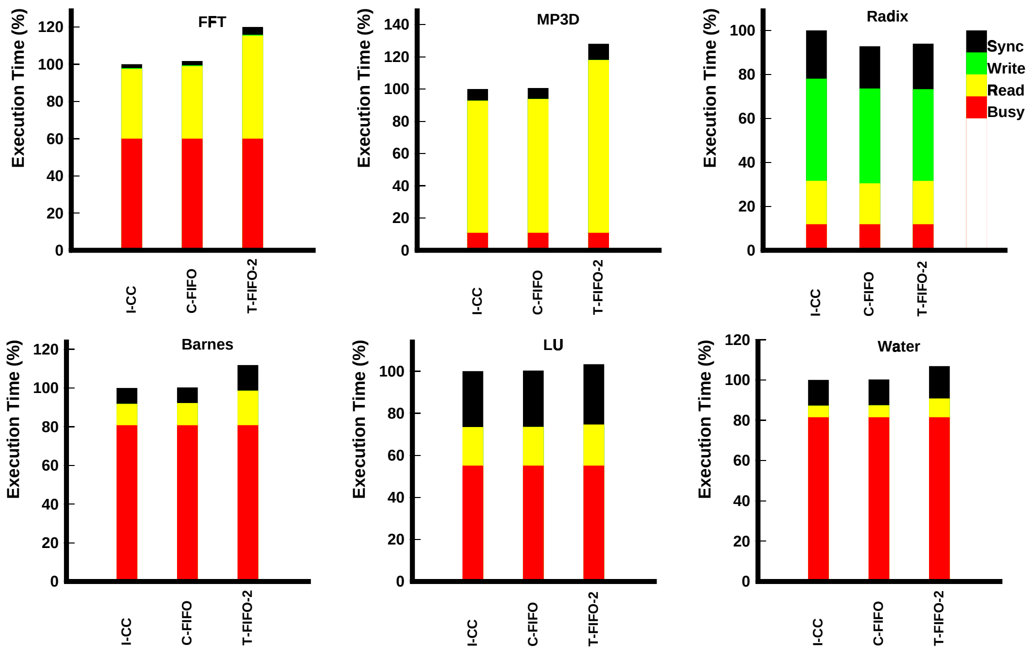


Figure 5.11: The overall execution times of benchmark applications on three configurations. Switches are connected using a 3D mesh topology.

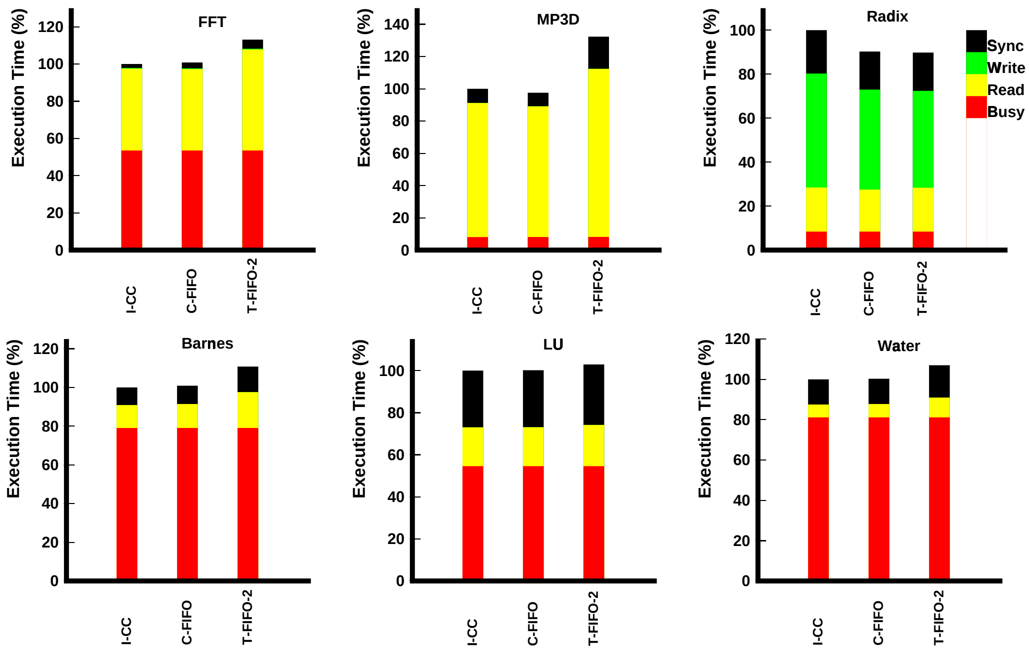


Figure 5.12: The overall execution times of benchmark applications on three configurations. Switches are connected using a 2D mesh topology.

CHAPTER 6

REDUCING INVALIDATION OVERHEAD IN FULL-MAP COHERENCE SCHEMES

In the previous two chapters, we have focused on the NNI and proposed novel NNI designs. In the next three chapters, we change our focus to the network and propose enhancements at the network for making the protocol layer more efficient. These enhancements help to reduce the remote memory access latency by addressing both the overhead at the NNI and the network delay — the two main components of the remote memory access latency according to Eqn. 3.15 in Chapter 3. Specifically, we look at the cache invalidation overhead problem in this chapter.

In current generation directory-based DSM systems, when a write operation is initiated, the home node sends multiple (unicast) invalidation requests to all sharing nodes and receives acknowledgments from them. These request/ack messages result in high traffic (thus, contention) in the network and incur considerable overhead at the NNI [61]. They also make the home nodes *hot-spots* in the network. Therefore, cache invalidation causes high-latency for write operations, leading to degradation on the overall system performance.

Under closer examination, it can be observed that cache coherence protocols use two fundamental message-passing operations in cache invalidation: sending *one-to-many* messages from one node to a set of other nodes and collecting *many-to-one* messages from a set of nodes to one node. Both patterns belong to the class of *collective communication* [96]. A few earlier studies have considered using collective communication in cache invalidation. In the WWT project, the invalidation requests are broadcasted using a dedicated broadcast network [117]. The drawback of such a design is that the broadcast network is very costly. Bhuyan et al. have proposed an embedded hierarchical ring broadcast mechanism for implementing fast invalidations [95]. The drawback is the high latency incurred.

Recently *multidestination* message-passing mechanisms have been introduced for wormhole networks to achieve low-latency multicast [112, 111] and gather [108] operations on distributed memory systems. In this chapter, we propose a set of multidestination-based *reservation* and *gather* worms to implement cache-coherence with reduced overheads. A small set of *invalidation acknowledgment (i-ack)* buffers are proposed to be used at the router interfaces to facilitate fast collection of acknowledgments. Different grouping schemes are proposed to send cache-invalidation requests and collect acknowledgments for deterministic (dimensional order) and adaptive (turn-model) routing schemes. The effect of these grouping schemes to reduce the number of invalidation request/ack messages, total network messages and overall execution time is studied through simulation experiments. Depending on the invalidation characteristics in these applications, a wide range of improvement (2-15% reduction in overall execution time) is observed. It is shown that turn-model routing with a *density-dependent column grouping* leads to the best reduction on overall execution time for these applications. In this chapter, we consider a full-map directory system with sequential consistency. However, our methodology is quite general and can be applied to other directory schemes and consistency models. In the next chapter, we show how this methodology can be applied to limited directory schemes.

6.1 Cache Invalidation Overhead in DSM systems

In this section, we analyze communication characteristics of cache invalidation transactions and the associated overhead.

6.1.1 Characteristics of Invalidations

In a system using a directory-based write-invalidation protocol, when a request to obtain exclusive-write access comes from a writer node to the home node of a cache block, the home node sends invalidation request messages to all sharers. These sharers, after receiving the invalidation requests, invalidate their respective cache blocks and send an acknowledgment each to the home node. The home node receives all these acknowledgments and then provides exclusive-write access to the writer node requesting the cache block. The entire sequence can be defined as an *invalidation transaction*, which consists of mainly two phases: *request* and *acknowledgment.*

Figure 6.1 illustrates the two phases for a sample sharing distribution in a dimensional order routed 8×8 mesh DSM supporting point-to-point (unicast) message passing. In this example, the degree of sharing for the memory block being invalidated is assumed to be 24. Thus, the home node sends 24 invalidation requests and receives 24 acknowledgments. Let us denote such a framework as *Unicast-based Invalidation and Unicast-based Acknowledgment (UI-UA).* Under this framework, the invalidation takes considerable time due to sending and receiving a large number of messages. The hot-spot effect occurs at the home node in both the request phase as well as the acknowledgment phase.

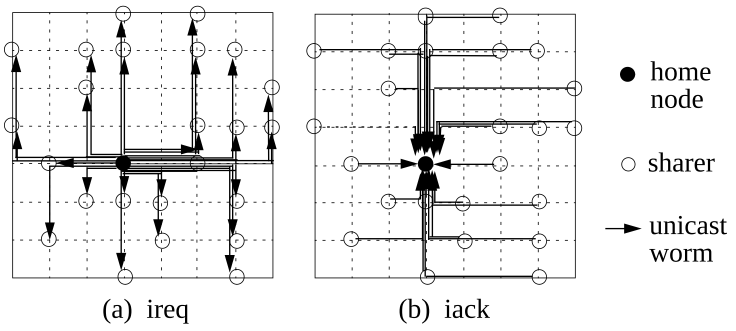


Figure 6.1: Example of an invalidation in an dimensional order routed mesh DSM supporting unicast communication: (a) the request phase and (b) the acknowledgment phase.

It is noted that a strong dependency exists among the request and reply messages for achieving cache coherence, leading to deadlocks in such systems. As a common practice, a pair of separate networks (at least logically separated) are used. Any request and reply messages that are related to each other are forced to travel in different networks to break the hold-and-wait condition that is necessary for a deadlock to occur. This means that the above two phases of an invalidation transaction usually take place in separate networks. Thus, for a major fraction of the time of an invalidation transaction, these two phases progress in an overlapped manner.

6.1.2 Latency and Traffic Estimates

In order to analyze the latency of an invalidation transaction quantitatively, let us use the following four simple measures. *Home node occupancy* [61] reflects the amount of processing time required at a home node in order to send the requests and receive the acknowledgments. It is proportional to the number of messages sent from and received by the home node. *Average distance* of a sharer from the home node reflects the network latency component of invalidation latency. Based on the underlying routing scheme, this component is related to the average number of hops traveled by a message. The *number of messages* and *total number of hops* traversed by the messages offer us valuable insight to the volume of network traffic generated by an invalidation transaction. For the example shown in Fig.6.1, the values of these four measures are 48, 4, 48, and 190, respectively.

Let us now derive an estimate for latency of an invalidation on a $k \times k$ mesh using wormhole routing [104]. Assume the average number of nodes sharing a block is d. Thus, the invalidation will require $2d$ messages. Let us assume the following timing parameters: message startup time equal to α, router delay equal to β, and one hop delay per flit equal to γ. Assume the message length is l flits. For a $k \times k$ system, the average distance traveled by a single message is k. During the invalidation transaction, the home node sends requests to the sharers one after another. If we ignore network contention, the overall time for all the requests to be received by the sharers $T_{ireq} = d\alpha + k\beta + (l-1)\gamma$. Let us assume that cache invalidation at a sharer takes δ time on average. An acknowledgment takes $T_{iack} = \alpha + k\beta + (l-1)\gamma$ time to reach the home node. By assuming the best overlapping, i.e., $(d-1)$ acknowledgments have been received and processed by the home node prior to the arrival of the last acknowledgment from a sharer, the overall invalidation latency $T_{inv} = T_{ireq} + \delta + T_{iack} = (d+1)\alpha + 2k\beta + 2(l-1)\gamma + \delta$.

Typically, network contention and hot-spot effect surrounding the home node increase this latency considerably. For the above analysis, the total number of hops traveled by $2d$ messages are $2dk$. This relates closely to the volume of communication traffic required for an invalidation. As d and k increase, it can be observed that the volume of communication traffic increases, leading to a potential increase in network contention. It also increases the no-contention invalidation latency, T_{inv}. This leads to a question whether less than d invalidation requests and d acknowledgments can be used to accomplish an invalidation with d sharers. We use a multidestination message passing approach to accomplish this.

6.2 Multidestination Message Passing

The multidestination message passing mechanism allows data to be delivered to or picked up from multiple nodes with a single message. A new Base-Routing-Confirmed-Path (BRCP) model has been recently proposed in [112] to implement multidestination mechanism on wormhole networks with different routing schemes.

6.2.1 The BRCP Model

Figure 6.2 shows feasible paths under the BRCP model for a 2D mesh supporting two kinds of routing. In a dimensional order routed (also known as e-cube routed) system — assuming messages are routed first along the row and then along the column, a multidestination worm can cover a set of destinations in row/column/row-column order as shown in Fig. 6.2. On a turn-model system, a multidestination worm can cover destinations along any west-first non-minimal path, in addition to e-cube paths, as shown in Fig. 6.2. The significant benefit of this BRCP model comes from the fact that a message can be delivered to multiple destinations with the same overhead as that of sending it to a single destination. Similarly, information can be gathered from multiple destinations with a single message.

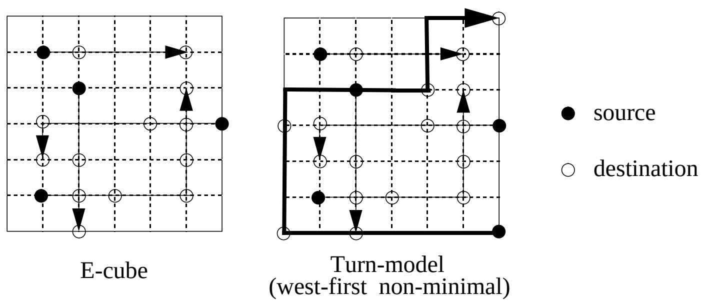


Figure 6.2: Feasible paths for multidestination worms in a 2D mesh [112].

6.2.2 Multicast and Gather Worms

A multidestination multicast worm consists of a set of destinations along a feasible path in its header. The worm uses *forward-and-absorb* capability at the router interface of each intermediate destination. Using such worms, sophisticated multicasting schemes are feasible to implement a multicast with reduced latency [112].

A multidestination gather worm collects information from multiple nodes at their respective router interfaces [108]. Each router interface can have a set of buffer entries containing special flags. An 'on' state of such a flag indicates that the associated processing node has arrived at the gather execution point. A typical gather worm, while passing through the router interface of intended destinations, checks for this flag. If the flag is 'on', the worm collects the information and proceeds ahead. If the flag is not 'on', the worm waits for this flag to be 'on'. Such a mechanism allows a gather worm to collect data/signal in a cumulative manner from all its intermediate destinations and deliver it to the final destination. In the following section, we propose augmentation to these multidestination worms for effective cache invalidations.

6.3 Frameworks for Reducing Overhead

In this section, we introduce new frameworks and mechanisms to implement efficient invalidations with multidestination messages.

6.3.1 MI-UA Framework

By using a multidestination multicast worm, a home node can send an invalidation request to a set of sharers along a single path. Let us name such a worm as *multidestination-based invalidation request (m-ireq)* worm. After receiving the request, a sharer invalidates its local cache and sends a unicast-based acknowledgment message back to the home node. The home node collects all the individual invalidation acknowledgments. We identify such a framework as *Multidestination-based Invalidation request and Unicast-based Acknowledgment (MI-UA)*.

Let us compare the MI-UA and the UI-UA frameworks through an example. Figure 6.3 highlights the distinction between these two frameworks. We only illustrate a portion, relating to the sharing nodes along column 6, of the example invalidation transaction previously shown in Fig. 6.1. The rest of the invalidation transaction can be implemented similarly and is not illustrated here for clarity. During the request phase of the invalidation transaction in the UI-UA framework, as shown in Fig. 6.3(a), the home node sends five unicast-based worms to the sharers. While in the MI-UA framework, as shown in Fig. 6.3(b), the home node sends only two (up-turn and down-turn) m-ireq worms. During the acknowledgment phase, each of these sharers sends a unicast-based invalidation acknowledgment back to the home node in both frameworks. It can be observed that for the example invalidation transaction in Fig. 6.1, the home node needs to send only 11 invalidation worms in the MI-UA framework as compared to 24 worms in the UI-UA framework. This demonstrates considerable potential to reduce occupancy at the home node as well as the invalidation latency.

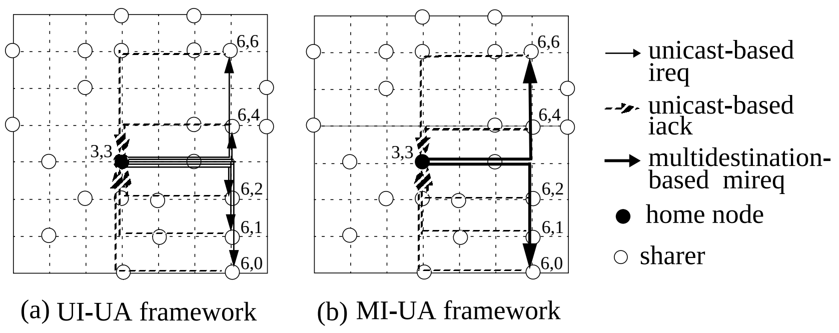


Figure 6.3: Comparing the communication traffic for an invalidation transaction in two frameworks: (a) UI-UA and (b) MI-UA.

Besides supporting *forward-and-absorb* mechanism at each router interface [112], the MI-UA framework does not require any additional modifications to the current generation DSM systems. The main thrust of this framework is to reduce occupancy of the home node and the volume of network traffic incurred in the request phase of an invalidation transaction. In the acknowledgment phase, however, this framework reduces neither the occupancy at the home node nor the occupancy at the sharers. It also does not reduce the volume of network traffic. The question is whether a better framework is feasible which can achieve the above three kinds of reduction in the acknowledgment phase. In order to have such a framework, we first introduce three novel mechanisms.

6.3.2 Three Novel Mechanisms

I-gather Worm

Let us consider using gather worms to collect the acknowledgments. Assume that after receiving the request and invalidating the local cache block, a sharer can set a buffered signal at its associated router interface instead of being forced to send a unicast-based acknowledgment. A multidestination-based gather worm can pass through each router interface associated with the sharers, collect the signal, and deliver the information to the home node. We name such a gather worm as a *multidestination-based invalidation gather (i-gather)* worm. Such i-gather worms significantly reduce: 1) the occupancy of cache invalidation for most sharers, 2) the volume of network traffic, and 3) the occupancy at the home node.

I-ack Buffer

A special buffer at the router interface is used to store the signal associated with an invalidation acknowledgment of a memory block for a short period of time. We denote such buffer as *i-ack buffer*, an entry in the i-ack buffer as *i-ack entry*, and the signal as *i-ack signal.* Ideally, every router interface could have a dedicated i-ack entry for each block of the global memory. However, it is not feasible to do so because of cost consideration. Fortunately, the average number of invalidation transactions, which a node participates during a given time interval, is quite small. Let us consider allowing sharing of the i-ack entries at a router interface among the ongoing invalidation transactions. In order to distinguish which i-ack entry is used by which invalidation transaction of a memory block, the block address of the invalidation transaction is stored in the i-ack entry and used as an identifier. When an i-gather worm arrives, it can do a fully-associative search based on the block address to find its corresponding i-ack entry and collect the i-ack signal. Hence, we propose that the structure of an i-ack entry consists of a free/used flag, an invalidation acknowledgment signal (i.e.,*i-ack signal*), a block address field, and a field to hold a copy of the invalidation acknowledgment message. We discuss the usage of the message field in next section. Figure 6.4 shows the enhanced node organization and the associated router interface.

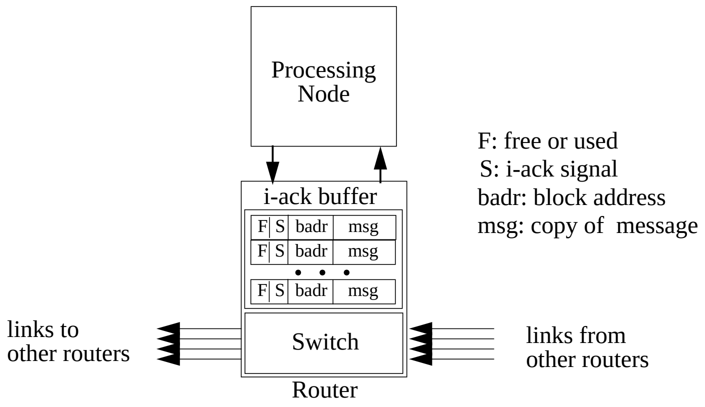


Figure 6.4: Enhanced node organization and the associated router interface with the i-ack buffer.

I-reserve Worm

With the above enhanced node organization and router interface, we introduce the concept of a *multidestination-based invalidation reservation (i-reserve)* worm. The i-reserve worm is a variation of the general multidestination-based multicast worm. Basically, an i-reserve worm is an augmented m-ireq worm. In addition to delivering the invalidation request message, it also *reserves* an i-ack entry at the router interface of each destination. When a sharer receives an invalidation request, it invalidates its cache block and sets the i-ack signal in the reserved i-ack entry. Later, when the associated i-gather worm arrives, it collects the i-ack signal and releases the i-ack entry.

6.3.3 MR-MA Framework

We glue the above three mechanisms together to make a *Multidestination-based invalidation Reservation and Multidestination-based gather Acknowledgment (MR-MA)* framework. For this framework to function smoothly, we need to investigate several important issues closely.

I-ack Hits and I-ack Misses

In our proposed design, there are only a few i-ack entries at the router interface. When an i-reserve worm reaches the router interface, it is possible that there is no available i-ack entry. What should the MR-MA framework do on such scenarios? When such a scenario occurs (denoted as *i-ack miss*), we allow the i-reserve worm to deliver the invalidation request and keep moving. Under such circumstances, after receiving and processing the invalidation request, the sharer sends a unicast-based invalidation acknowledgment message. Later when the corresponding i-gather worm reaches this sharer, it can find out that there is no i-ack entry reserved for the block and proceeds to the next sharer/home node. We define an *i-ack hit* as a success in reserving an i-ack entry by an i-reserve worm. In Section 6.5, we will show simulation results corresponding to i-ack hits/misses with varying number of i-ack entries. It is to be observed that 2-4 entries are sufficient to have a hit ratio higher than 98.7%.

I-ack Signal Counts

Once i-ack misses occur, it becomes impossible for the home node to decide when an invalidation transaction completes, based on the number of acknowledgment messages it has received. Hence, we incorporate an i-ack counter field in the i-gather worm. This field can be used to remember how many i-ack signals that an i-gather worm actually collects. For a unicast-based acknowledgment, this count is assumed to be 1. At the beginning of an invalidation, from the directory entry for a memory block, the home node knows the total number of sharers (denoted as x). On receiving the invalidation acknowledgments for the memory block, the home node decrements x by the i-ack count in the acknowledgments. When x becomes 0, the invalidation transaction is completed.

Let us apply this MR-MA framework to the example invalidation transaction shown in Fig. 6.1. Again, only the portion relating to sharers along column 6 is illustrated in Fig. 6.5 for clarity. The home node (3,3) sends two i-reserve worms. As the i-reserve worms propagate, let us assume that only one i-ack miss occurs at sharer (6,1). The i-gather worms later collects two i-ack signals each. (Special unicast-based acknowledgment messages are needed in a dimensional order routed system because the i-gather worms can not reach the home node under X-Y routing.) Independently, sharer (6,1) sends a unicast-based acknowledgment back to the home node. Overall, assuming the best-case where only i-ack hits occur, it can be observed that for the example invalidation transaction in Fig. 6.1, the home node needs to send and receive only 22 invalidation worms in the MR-MA framework as compared to 48 worms in the UI-UA framework or 35 worms in the MI-UA framework. This demonstrates further reduction in the volume of network traffic, occupancy at the home node, and most importantly the invalidation latency.

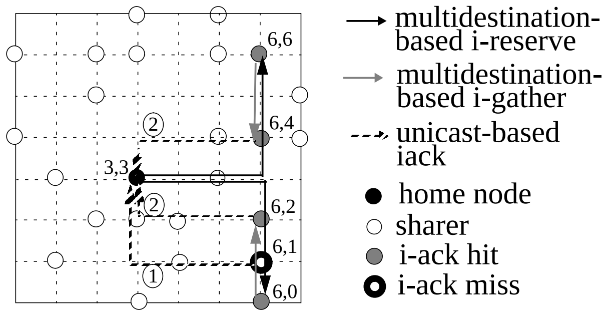


Figure 6.5: An example of a portion of invalidation using the MR-MA framework.

Figure 6.6 presents the routing algorithm for a reservation worm. Figure 6.7 presents the routing algorithm for a gather worm. Figure 6.8 presents the algorithm at the NNI to process the various invalidation requests. Figure 6.9 presents the algorithm at the NNI to process the various invalidation acknowledgments.

6.4 Grouping Schemes

It can be seen that i-reserve and i-gather worms must be generated on-the-fly by the directory controller because of the dynamic property of sharing. Let us define *grouping* as the procedure of selecting a set of i-reserve and i-gather worms to cover all sharers of a memory block. In this section, we propose and analyze some heuristic grouping schemes for dimensional order routing and turn-model routing. All grouping schemes are illustrated with the example invalidation pattern introduced in Fig. 6.1.

6.4.1 Associated Issues

Directory Organization and Grouping for I-reserve Worms: It can be very helpful if the pointer array is so organized that when an invalidation transaction occurs, the home node can send the i-reserve worms with little overhead. Assume that the full-map protocol is used to enforce coherence. Let us consider organizing the presence bits in the column major order and use the bit string address encoding for multidestination worms [108]. Hence, different portions of the presence bits can be directly taken as the routing headers of the i-reserve worms.
Grouping for I-gather Worms: In all the schemes we discuss, the final destination node of an i-reserve worm initiates an i-gather worm to collect the i-ack signals. This node needs the identifiers of the sharers that are covered by the associated i-reserve worm. To satisfy such a requirement, a copy of the routing header of the i-reserve worm can be duplicated and carried as an additional data item in the worm. This may increase the length of an i-reserve worm slightly. However, the

```
Algorithm Reservation Worm
    Input:     a reservation worm, a router
    Output:    a possible invalidation request to the router's node,
               its subtype is BUFF, NONBUF, or MULTI.
ReservationWorm(rworm, router)
{
    $d$ = next destination of $rworm$;
    if ($router \neq d$)
        Forward $rworm$;
    else if ($d \neq$ final destination of $rworm$) {
        if (check($router$.consump-channel)==FREE) {
            Forward $rworm$;
            $i$ = reserve($router$.i-ack);
            if ($i \geq 0$)
                Send a BUFF inval request to the node with value $i$;
            else
                Send a NONBUF inval request to the node;
        }
        else {
            $i$ = reserve($router$.i-ack);
            if ($i \geq 0$) {
                Buffer a copy of $rworm$ in i-ack;
                Forward $rworm$;
                wait($router$.consump-channel);
                Send a BUFF inval request to the node with value $i$;
            }
            else
                wait($router$.consump-channel);
        }
    }
    else {
        reserve($router$.consump-channel);
        Send a MULTI inval request to the node;
    }
}
```

Figure 6.6: The routing algorithm for a reservation worm.

```
Algorithm Gather Worm
    Input:     a gather worm, a router
    Output:    a MULTI inval ack to final destination
GatherWorm(gworm, router)
{
    $d$ = next destination of gworm;
    if ($router \neq d$)
        Forward gworm;
    else if ($d \neq$ final destination of gworm) {
        $i$ = search(router.i-ack, gworm.badr);
        if ($i < 0$)
            Forward gworm;
        else if (status(router.i-ack[i].signal)==OCCUR) {
            free(router.i-ack);
            Increment gworm.count;
            Forward gworm;
        }
        else {
            Store gworm in router.i-ack[i].msg;
            wait(router.i-ack[i].signal);
            free(router.i-ack[i]);
            Increment gworm.count;
            Forward gworm;
        }
    }
    else {
        reserve(router.consump-channel);
        Send a MULTI inval ack to the node;
}
```

Figure 6.7: The routing algorithm for a gather worm.

```
Algorithm Invalidation Handler in a Node's NNI
    Input:    a inval request
    Output:   i-ack node done signal, or a unicast inval ack, or a gather worm
InvHandler(ireq, router)
{
    typ = subtype of the ireq;
    if (typ == BUFF) {
        Invalidate cache;
        i = i-ackbuf(ireq);
        Set router.i-ack[i].signal;
    }
    else if (typ == NONBUF) {
        Invalidate cache;
        Send unicast UNI inval ack;
    }
    else { /* MULTI inval req */
        Invalidate cache;
        header = grouping(ireq.destlist);
        Put 1 in new gather worm's count;
        Send the gather worm using header as its path;
    }
}
```

Figure 6.8: Invalidation handler algorithm at the NNI.

```
Algorithm Invalidation acknowledgment Handler in a Node's NNI
    Input:    an inval acknowledgment, a memory block directory
    Output:   complete an invalidation transaction
IAckHandler(iack, directory)
{
    typ = subtype of the iack;
    if (typ == UNI)
        Decrement directory.iack;
    else /* MULTI inval ack */
        directory.iack = directory.iack − iack.count;
    if (directory.iack == 0)
        Complete the invalidation transaction;
    else
        Wait for more inval ack;
}
```

Figure 6.9: Invalidation acknowledgment handler algorithm at the NNI.

delay induced by such an increase in worm length is very small under wormhole routing. After the final destination of an i-reserve worm receives the message, its directory controller does a grouping with itself as the source node and the home node as the final destination in order to generate appropriate routing header for the corresponding i-gather worm. Readers are encouraged to refer to [30] for further details.

6.4.2 Dimensional Order Routing in 2D Mesh

Up-and-Down Column Grouping Scheme

In an *up-and-down column grouping* (UD) scheme, a home node needs maximum two i-reserve worms in each column for an invalidation transaction. One such worm, the *up i-reserve* worm, moves along the +Y direction after a possible turn from ±X. The complementary *down i-reserve* worm moves along the –Y direction. A sharer in a column locating on the row containing the home node can be covered by either worm. An associated i-gather worm visits the sharers in the reverse order of the i-reserve worm along the ±Y direction to collect the i-ack signals. The i-ack signal count is finally carried back by a unicast-based worm to the home node. Figures 6.10(a) and 6.10(b) illustrate the request and acknowledgment phases using the UD scheme. As discussed in Section 6.1.1, we consider two dimensional order routed virtual networks, i-reserve worms moving in one and i-gather worms in the other.

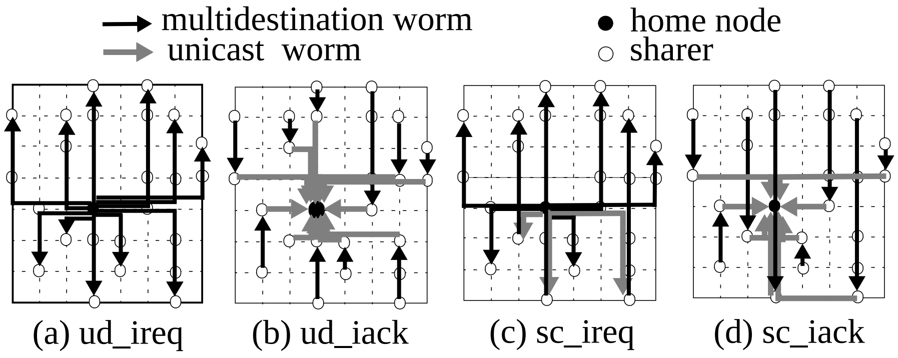


Figure 6.10: Dimensional order routing based grouping schemes.

To do a performance analysis similar to that for the UI-UA framework, we use similar assumptions as described in Section 6.1.2. In addition, let us assume the startup time for multidestination-based messages as α_m. Assume d_{ud} i-reserve worms are initiated at the home node on average. Thus, the best-case invalidation latency, $T_{inv,ud}$, can be derived as $\alpha+(d_{ud}+1)\alpha_m+2h_{ud}\beta+2(l_{ud}-1)\gamma+2\delta$ and the total number of hops traversed by the messages, $H_{inv,ud}$, as $3d_{ud}h_{ud}$.

Selective Column Leader Grouping Scheme

In a *selective column leader grouping* (SC) scheme, a home node initiates an up or down i-reserve worm as in the UD scheme if it can cover all the sharers in a column. Otherwise, the home node initiates a unicast-based worm to a selected column leader. The leader is the highest or lowest sharer in a column whichever is closer to the home node. Figures 6.10(c) and 6.10(d) illustrate the request phase and the acknowledgment phase using the SC scheme. In the SC scheme, for each column, the home node sends maximum one worm. Compared to the UD scheme, this scheme achieves a better balance between the occupancy of the home node and the propagation delay of the invalidation request than the UD scheme. Assume $d_{sc,u}$ unicast-based worms and $d_{sc,m}$ multidestination-based worms are initiated at a home node on average for an invalidation. This leads to: $T_{inv,sc} = (d_{sc,u} + 1)\alpha + (d_{sc,m} + 1)\alpha_m + 2h_{sc}\beta + 2(l_{sc} - 1)\gamma + 2\delta$ and $H_{inv,sc} = 3(d_{sc,u} + d_{sc,m})h_{sc}$.

6.4.3 Turn Model Adaptive Routing in 2D Mesh

It is to be noted that any dimensional order routing based grouping scheme can be used in networks supporting turn-model routing [104]. However, we discuss two new grouping schemes to exploit the adaptivity better. Similar to e-cube routing, we consider two virtual networks, one supporting west-first routing and the other east-first.

Dual-Path Grouping Scheme

In this scheme a home node needs maximum two i-reserve worms for an entire invalidation transaction. One such worm, the *left i-reserve* worm, covers all the sharers on the left side of the home node and travels in the network using east-first routing. The complementary *right i-reserve* worm covers all the sharers on the right side and travels in the network using west-first routing. The sharers along the column containing the home node can be divided into two half columns: upper and lower with respect to the home node. Each of the half column, but not both, can be covered by either of the i-reserve worms. This scheme achieves the greatest reduction in volume of network traffic among all the proposed grouping schemes. Assume d_{dp} i-reserve worms are initiated at a home node on average for an invalidation. This leads to: $T_{inv,dp} = \alpha + (d_{dp}+1)\alpha_m + 2h_{dp}\beta + 2(l_{dp}-1)\gamma + 2\delta$ and $H_{inv,dp} = 2d_{dp}h_{dp}$. Figures 6.11(a) and 6.11(b) illustrate the request phase and the acknowledgment phase using the DP scheme.

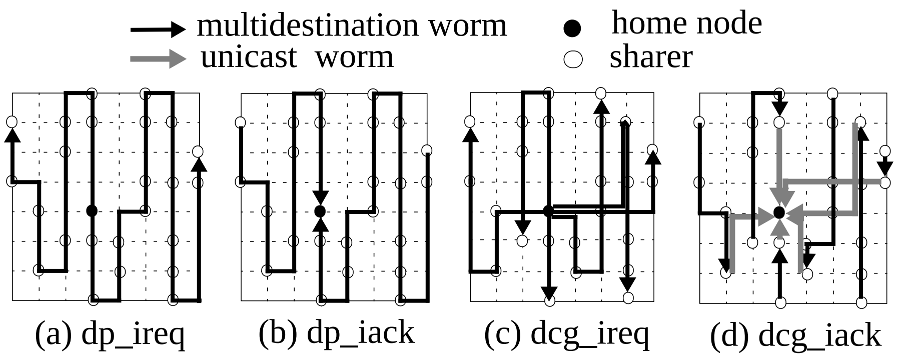


Figure 6.11: Turn model routing grouping schemes.

Density-dependent Column Grouping Scheme

Two major drawbacks of the DP scheme are: 1) high latency and 2) complicated routing headers for the multidestination worms. A *density-dependent column grouping* (DCG) scheme tries to balance the number of destinations covered by each worm. In addition to statically partitioning the network into c adjacent columns each, this scheme restricts the maximum number of destinations a multidestination-based worm can cover in the c columns to a certain threshold value (d). When the threshold is reached, more than one i-reserve worms are used to cover destinations in the c columns. Figures 6.11(c) and 6.11(d) illustrate the request phase and the acknowledgment phase of the invalidation transaction in the DCG scheme with $c = 2$ and $d = 6$. Assume d_{dcg} i-reserve worms are initiated at a home node on average for an invalidation. This leads to: $T_{inv,dcg} = \alpha + (d_{dcg} + 1)\alpha_m + 2h_{dcg}\beta + 2(l_{dcg} - 1)\gamma + 2\delta$ and $H_{inv,dcg} = 3d_{dcg}h_{dcg}$.

grouping scheme	home occup.	avg dist.	# of mesg	total hops
`unicast`	46	4.0	46	186
`d_ud`	22	4.5	31	99
`d_sc`	16	6.6	32	106
`t_dp`	4	22.0	4	88
`t_dcg`	12	8.5	18	102

Table 6.1: Comparison of different grouping schemes.

6.4.4 Comparison of Grouping Schemes

We compared these schemes by using the four measurements proposed earlier in Section 6.1. For a fair comparison across different schemes, if worms traveled in a chained fashion, they are counted as one message with a distance equivalent to the sum of distance traveled by component worms; while in the calculation of the number of total messages they are counted individually. Table 6.1 summarizes the results for the sample invalidation transaction, introduced in Fig. 6.1. A simple observation from this table is that all the proposed schemes cut down the volume of network traffic for the invalidation effectively with respect to unicast scheme. Furthermore, it can be observed from Table 6.1 and derived from earlier latency expressions that the SC scheme is more efficient than the UD scheme while the DCG scheme is more efficient than the DP scheme.

6.4.5 Systems with Other Topologies

All the schemes developed so far have focused on a 2D mesh system. When we apply our framework to other k-ary n-cube systems, more design flexibility is available for the grouping. In this subsection, we look at grouping schemes for a 3D mesh.

Axis-Based One-step Reservation Scheme

Let us first consider a full-map directory in a 3D DSM system. Consider the network using the popular dimensional order (X-Y-Z) routing algorithm. Let us organize the presence bit array in the directory using Z-Y-X indexing order with Z index growing the fastest and X index the slowest. As a concrete example system, consider a $3 \times 3 \times 3$ mesh. The sharing nodes, 11 in this example, are distributed as shown in Fig. 6.12. Assume that the cache coherence protocol sets the presence bits of sharing nodes in the directory during a sequence of remote read operations. When an invalidation occurs due to a subsequent write operation, the presence bits are checked along the Z dimension, specifically, along the column as shown in Fig. 6.12. If no bits in a single column is set, no invalidation activity is required for that column. Otherwise, an up-turn or down-turn multidestination reservation worm is sent by the home node to cover the sharing nodes above or below the home node in that column, as shown in Fig. 6.12(a).

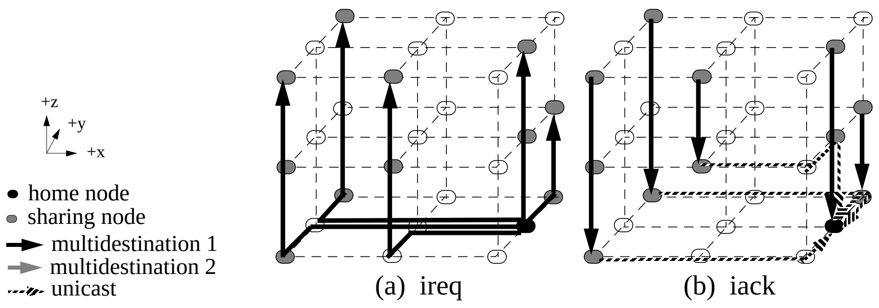


Figure 6.12: Axis-based one-step reservation (AB1R) scheme in a 3D mesh supporting dimensional order routing using full-map directory. The two phases of an invalidation transaction: (a) request and (b) acknowledgment.

An i-ack gather worm is initiated by the last destination of each reservation worm to collect the associated acknowledgments. A subsequent unicast message sends the i-ack count back to the home node, as shown in Fig. 6.12(b). Let us denote such a scheme as an *axis-based one-step reservation (AB1R)* scheme.

In the example, the above scheme uses 5 invalidation reservation and 9 gather/unicast (14 in total) messages to complete the invalidation. While a unicast-based scheme needs 11 invalidation request and 11 acknowledgment (22 in total) messages to implement the same invalidation. Hence, the network traffic is reduced by one third. The grouping algorithm for the multidestination worms is no more complicated than a unicast worm. The smaller number of messages in the request phase of the invalidation helps to cut down the home node occupancy and the latency of the entire invalidation. In general, the reduction in network traffic and latency for any invalidation can be achieved in a similar fashion.

Axis-Based Two-step Reservation Scheme

Now let us consider the network using an *all-but-X-negative-first* routing algorithm. Let us organize the presence bit array in the directory using Z-X-Y indexing order. When an invalidation occurs, the presence bits are first checked along the Z dimension. A sharing node having the smallest or largest Z index (top or bottom) is identified as a potential column leader. Depending on the position of the home node, either the top or the bottom sharing node is chosen to be the actual column leader. Instead of initiating a separate multidestination reservation worm for each column which contains a sharing node, like the previous scheme, we divide the leaders into groups according to their Z index values. In other words, the column leaders are divided into separate X-Y planes. Using the presence bit information, each group can be covered by one or multiple multidestination reservation messages (multidestination messages of type 1). In addition to the coherence information, such a multidestination message contains a copy of those presence bits of the columns whose leaders are covered by the multidestination worm. Once receiving the reservation message of type 1, a column leader extracts the presence bits of its own column and sends a multidestination reservation message of type 2 along the Z axis to cover the sharing nodes in the column.

An i-ack gather worm is initiated for each reservation worm of type 2 to collect the associated acknowledgments within a column. Similar to the column-based one-step reservation scheme, a subsequent unicast message sends the i-ack count back to the home node. Figures 6.13(a) and 6.14(b) illustrate the request phase and acknowledgment phase of this scheme on the same example transaction of Fig. 6.12. Let us denote such a scheme as an *axis-based two-step reservation (AB2R)* scheme.

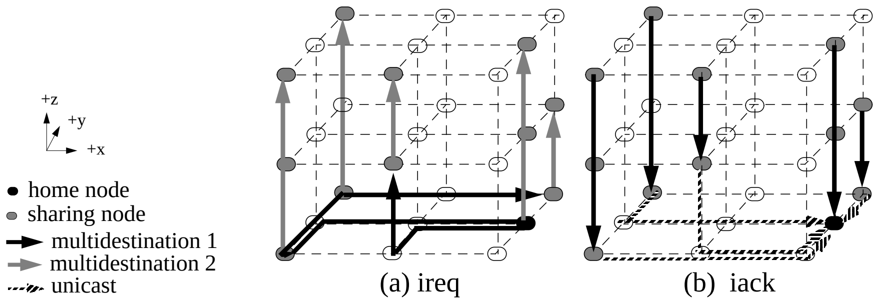


Figure 6.13: Axis-based two-step reservation (AB2R) scheme in a 3D mesh supporting *all-but-X-negative-first turn model* routing using full-map directory. The two phases of an invalidation transaction: (a) request and (b) acknowledgment.

For the current example, this scheme uses 7 invalidation reservation and 9 gather/unicast (16 in total) messages to complete the invalidation. It is to be noted that the home node starts 3 messages, 2 of type 1 multidestination and 1 of type 2 multidestination. Generally, comparing to the unicast-based scheme, the network traffic is reduced. However, the grouping algorithm for multidestination worm is relatively complex. The home node occupancy can be reduced dramatically since such a grouping is expected to be fast. This leads to a potentially significant reduction in invalidation latency.

6.4.6 Plane-Based Two-step Reservation Scheme

As yet another alternate scheme, let us consider the network using an *all-but-X-negative-first* routing algorithm. Let us organize the presence bit array in the directory using X-Y-Z indexing order. When an invalidation occurs, the presence bits are checked along the X-Y planes. If there exist some sharing nodes in a plane, a sharing node is selected to be the leader for this plane. One simple selection policy is to pick the sharing node with the shortest distance from the home node using the presence bit information. The leaders of all the planes which contain sharing nodes are grouped into a few paths. The home node first sends multiple multidestination reservation messages (multidestination messages of type 1) to these leaders. Again, in addition to the coherence information, such a multidestination message contains a copy of those presence bits of the planes whose leaders are covered by the multidestination worm.

After receiving the reservation message of type 1, a column leader extracts presence bits of its own plane and sends one or multiple multidestination reservation messages of type 2 to cover the sharing nodes in the X-Y plane. An i-ack gather worm is initiated for each reservation worm of type 2 to collect the associated acknowledgments within a column. A subsequent unicast message sends the i-ack count back to the home node. Figures 6.14(a) and 6.14(b) illustrate the request phase and acknowledgment phase of this scheme on the same example transaction. Let us denote such a scheme as a *plane-based two-step reservation (PB2R)* scheme.

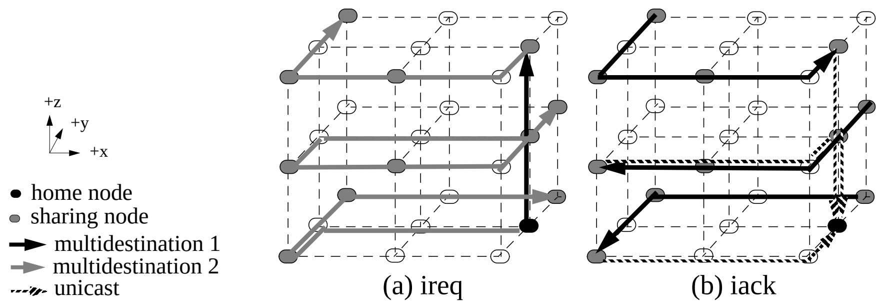


Figure 6.14: Plane-based two-step reservation (PB2R) scheme in a 3D mesh supporting all-but-X-negative-first turn model routing using full-map directory. The two phases of an invalidation transaction: (a) request and (b) acknowledgment.

For the current example, this scheme uses 4 invalidation reservation and 6 gather/unicast (10 in total) messages to complete the invalidation. It is to be noted that the home node starts 2 messages, 1 of type 1 multidestination and 1 of type 2 multidestination. Unfortunately, sophisticated combinatorial groupings have to be done for both types of multidestination messages, with type 1 at the home node and type 2 at the leaders. Generally, comparing to the unicast-based scheme, the network traffic is reduced. But node occupancy of the involved node, especially the leaders, may increase rapidly and diminish any reduction in invalidation latency. To remedy such drawbacks, we propose the following two enhancements.

1. Allow non-sharing nodes to be a plane leader. We can make the type 1 multidestination reservation messages to only travel along the Z-axis. Hence, the grouping of step 1 can be

done very fast by virtually a simple check on each plane for sharing nodes. The time spent at the leader for type 2 multidestination reservation messages remains almost unchanged. It is to be noted that the non-sharing nodes only participate in the request phase but not in the acknowledgment phase. Thus, no extra invalidation messages are introduced by any non-sharing node.

2. Allow multiple gather worms triggered by a single reservation worm. This sometimes simplifies the groupings for type 2 reservation worm and the subsequent i-ack gather worms. Therefore, the occupancy at the intermediate nodes and the invalidation latency could be reduced.

6.5 Simulation Experiments and Results

6.5.1 Simulation Setup

We have evaluated our proposed schemes using the simulation testbed. Table 6.2 lists the relevant system parameters used. It is noted that, for unicast messages, we assumed 5 processor clock cycles as the startup time. For multidestination messages this time was assumed to be 10 clock cycles. We considered an 8×8 system.

Parameter	Values	Time
Processor	1 cycle	5 ns
Cache access	1 cycle	5 ns
Cache block size	16 bytes	
Cache block fill time	8 cycles	40 ns
Cache set associativity	1	
Cache size per node	64 Kbytes	
Memory word width	4 bytes	
Memory block access time	8 cycles	40 ns
Directory check	2 cycles	10 ns
Directory check and update	4 cycles	20 ns
mesg startup (unicast)	5 cycles	25 ns
mesg startup (multidest)	10 cycles	50 ns
mesg dispatch	2 cycles	10 ns
Control mesg size (unicast)	4 bytes	
Control mesg size (multidest)	6 bytes	
Data message size	20 bytes	
Injection channels per node	2	
Consumption channels per node	4	
I-ack entries per router	4	
Channel width (Flit size)	2 bytes	
Link propagation delay	1 cycle	5 ns
Router switch delay	1 cycle	5 ns
Router delay (header)	4 cycles	20 ns

Table 6.2: Default System parameters used in the simulation.

We used three applications: Barnes (128 bodies, 4 steps), LU (128×128 matrix, 8×8 blocks), and All-Pairs-Shortest-Path (128×128 matrix, 50% connection).

6.5.2 Results and Discussions

We monitored three important parameters: the number of messages used for invalidation, the total number of network messages, and the overall execution time. In order to perform comparative evaluation, we used dimension order system with unicast-based message passing (d-uni) as the base case (100%). All other results are compared and reported against this base case.

Figure 6.15 shows the number of invalidation messages used. It can be seen that all multidestination schemes reduce the number of messages required for invalidation compared to the unicast-based (uni) message-passing. Turn-model with dual-path grouping (DP) always leads to maximum reduction (up to 95%).

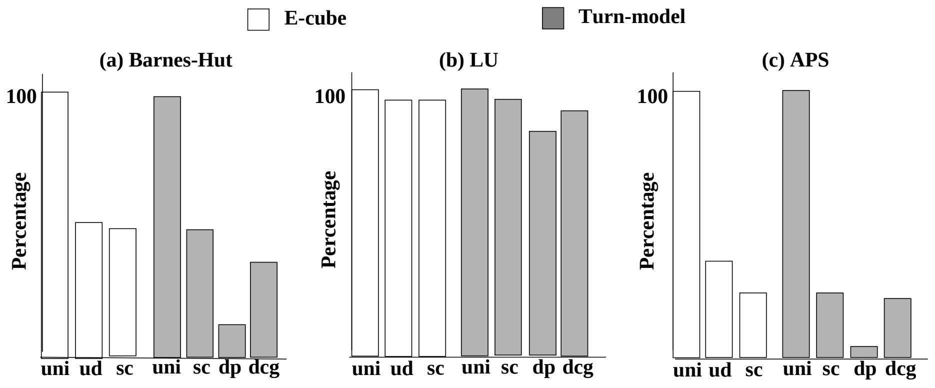


Figure 6.15: Comparing the number of invalidation messages required under the unicast-based and multidestination-based schemes.

Figure 6.16 shows the total number of network messages used. It can be observed that reduction in the number of total network messages varies significantly depending on the ratio of invalidation messages and other messages. For application like LU-decomposition which exhibits very low number of invalidations, the gain is very minimal (0.6% at the best). For other two applications, different grouping schemes lead to 20-40% reduction.

Figure 6.17 shows the overall execution time. It can be observed that substantial reduction in network messages do not necessarily lead to reduction in the overall execution time. This depends on the critical path of execution. However, some grouping schemes like SC and DCG are able to reduce the overall execution time up to 15%.

To understand the impact of i-ack buffer size on the system performance, we ran the Barnes application with different number of i-ack entries at each router interface. Figure 6.18(a) shows the normalized i-ack misses over the total number of reservation attempts. Figure 6.18(b) shows the normalized overall execution time. It can be observed that with only a few i-ack entries (2-4), the i-ack misses can be dramatically reduced without apparent execution-time loss.

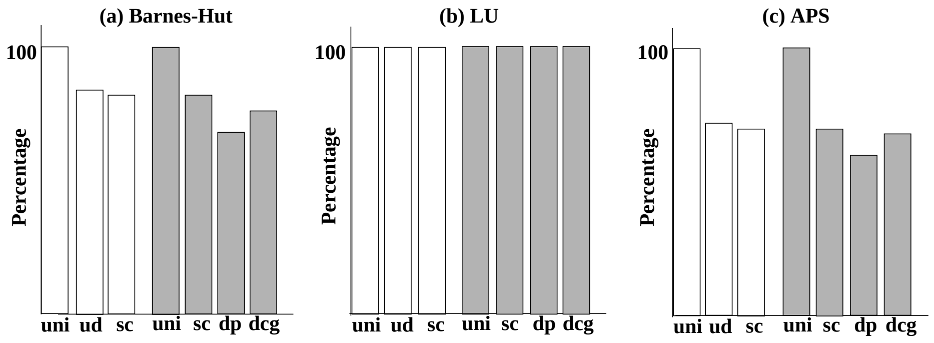


Figure 6.16: Comparing the total number of messages.

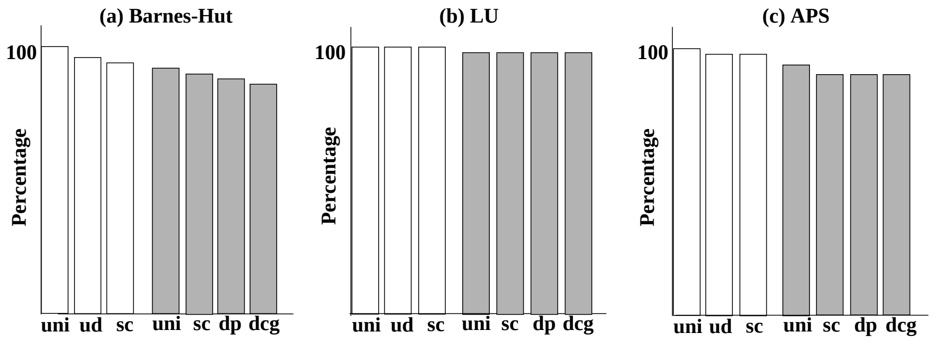


Figure 6.17: Comparing the overall execution time.

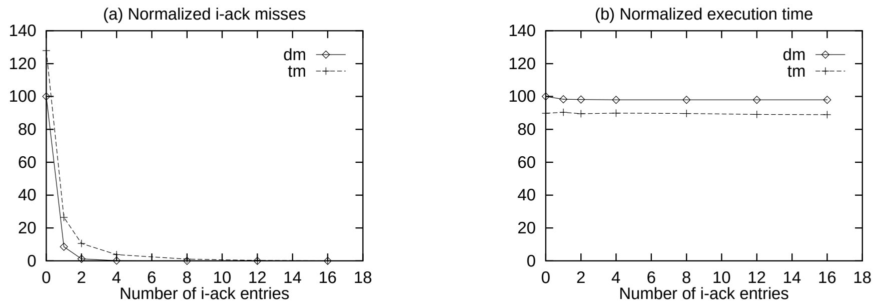


Figure 6.18: The impact of i-ack buffer size in Barnes application.

6.6 Summary

In this chapter we have introduced a new multidestination message-passing approach to implement directory-based cache coherency in wormhole distributed shared memory (DSM) systems. By applying the BRCP model, reservation and gather worms were used to distribute invalidation messages and to collect acknowledgments. Compared to the conventional approach, this new method produces less number of messages, less network traffic, and reduced occupancy at home nodes. New grouping schemes were proposed to generate these multidestination worms to implement the full-map protocol. Simulation results demonstrate considerable potential for these schemes to be applied to current generation DSM systems. In the next chapter, we investigate limited directory coherence schemes with this new approach.

CHAPTER 7

REDUCING INVALIDATION OVERHEAD IN LIMITED DIRECTORY SCHEMES

In this chapter, we look into how to extend the multidestination-based invalidation mechanisms proposed in the last chapter for full-map schemes to limited directory coherence schemes. Due to the large amount of storage required by a full-map directory scheme [91, 134], such a scheme is not practical for large scale systems. Therefore, many cost-effective *limited directory* schemes using smaller amount of storage have been proposed in the literature.

The basic idea behind limited directory schemes is to handle invalidations in an intelligent manner when the directory overflow occurs. Using a smaller number of messages in case of directory overflow is critical to the system performance. Otherwise, both network traffic and node occupancy [61] increase, leading to increase in *write-latency* and degradation in the overall performance. Examples of some limited-directory schemes are: coarse-vector [56], Limitless [81], Superset [3], and Eviction [24]. These schemes use either hardware or software mechanisms to detect and manage directory overflow.

The existing limited directory schemes can be broadly classified into two categories: broadcast-based and non-broadcast-based. In broadcast-based schemes (dir_iB), when the number of sharers goes beyond i, directory overflow occurs and invalidation messages are sent to all nodes in the system. However, on systems supporting only unicast message passing, a broadcast requires a large number of message transfers in the system and it quickly leads to performance degradation. Thus, researchers have proposed non-broadcast-based schemes like coarse-vector (dir_iCV_r) [56]. In this scheme, when directory overflow occurs, the storage space for i entries are reorganized and used as *region* bits. During invalidation, such region bits help in significantly reducing the number of messages needed to be sent to nodes with possible sharers.

In either of the above schemes, it is shown in the literature that 3-5 pointers are necessary to obtain performance comparable to a full-map directory. However, such guidelines are based on networks supporting only unicast message-passing. Since multidestination message-passing allows fast broadcast, we first analyze whether dir_iB schemes with less than three pointers ($i = 3$) can deliver performance comparable to the full-map scheme. We propose an efficient two-level broadcast scheme with multidestination messages and evaluate different dir_iB schemes, $1 \leq i \leq 3$. Simulation-based evaluations are done for two different system sizes (4×8 and 8×8) and four different applications from the SPLASH2 benchmark [146]. The simulations consider detailed instruction count for computational steps, detailed network behavior including flit-level link contention and network interface contention for message passing steps, and synchronization overhead. The results demonstrate that the m_dir_1B scheme (1 pointer with multidestination message passing-based broadcast support) is powerful enough to deliver performance closer to that of the full-map scheme with unicast message-passing. The performance of m_dir_1B is also shown to be closer to that of $u\_dir_3CV_r$ scheme (coarse vector with three pointers and implemented using unicast messages).

Next, we study the impact of coarse-vector organization under multidestination message passing. Two new schemes ($m\_dir_iCB_1$ and $m\_dir_iCB_2$) are developed to define suitable regions based on the inherent capability of multidestination worms to implement broadcast/multicast. These regions are defined along single/multiple columns such that one or two multidestination worms can cover all the nodes in a region. Such region-based schemes reduce the number of messages required for invalidation. Simulation experiments for the above systems and applications demonstrate $m\_dir_1CB_1$ scheme to be better than m_dir_1B and $u\_dir_1CV_r$. With a larger number of pointers ($i = 3$), all coarse vector schemes (using unicast or multidestination messages) are demonstrated to deliver better performance.

This study proves the fact that efficient limited directory schemes can in fact be designed by taking advantage of multidestination message passing on wormhole networks. It indicates that limited directory schemes with only one pointer are sufficient to deliver performance close to that of the full-map scheme. Such efficient limited directory schemes provide new guidelines for designing scalable DSM systems with reduced cost.

This chapter is organized as follows. Section 7.1 provides an overview of directory schemes. Section 7.2 proposes new limited directory schemes under multidestination message passing. Simulation experiments and results are presented in Section 7.3. Related work is reviewed in Section 7.4. Finally, concluding remarks are presented in Section 7.5.

7.1 Limited Directory Schemes

In this section, we first provide the motivation behind having limited directory schemes for DSM systems. Next, we briefly provide an overview of the existing limited directory schemes, and discuss their advantages and disadvantages.

7.1.1 Motivation Behind Limited Directories

Two important issues are typically considered in the design of a directory scheme for any DSM system. The first is the invalidation/update traffic generated by the scheme. Less accurate sharing information kept in a directory usually requires additional network bandwidth to send extra invalidations/updates. The second issue is the overhead, especially the memory storage overhead, required by the scheme. More accurate information means more memory used by a directory, thus higher overhead. These two issues must be addressed in a balanced fashion to get a suitable directory scheme for a system with reasonably good performance. The design decision also has a strong impact on the scalability of the system.

Numerous directory schemes have been proposed or implemented in the past. They generally fall into two broad classes: *full-map* and *limited*. In a *full-map* scheme, every processing node in the system is represented by one bit (*present bit*) in a fixed position in a directory. This scheme keeps the precise sharing information for each memory block at any given time. When a write occurs, invalidations/updates are sent to only those processing nodes that actually hold valid copies. Such a scheme uses a minimum amount of messages (network traffic) to carry out invalidation/update. However, the amount of memory overhead for maintaining the *full-map* directory is quite high. Thus, this scheme is only suitable for small or medium-sized systems.

In order to have reduced overhead for maintaining the directory, many limited pointer schemes have been proposed in the literature [56, 81, 3, 24]. These schemes typically contain a small number

(i) of pointers for any given memory block. When a memory block is cached by no more than i processing nodes at a given time, the scheme behaves exactly the same as that of the full-map scheme. However, when a block is cached by more than i processing nodes, some mechanism is used to handle the *pointer overflow* [134]. For a system with P processing nodes, each pointer needs $\log P$ bits for identification of the sharing node of a block. Assuming B memory blocks per processing node, the total storage requirement for directories is $iPB\log P$ bits. This is considerably less than the P^2B bits, required for a full-map directory.

The limited directory schemes usually perform reasonably well in practice because of the fact that most tuned parallel applications show a low degree of sharing for a dominant majority of data objects most of the time. Limited pointer schemes differ in the way they handle pointer overflow. Depending on the design choice, each scheme produces significantly different amount of invalidation and data traffic. There are two major classes of limited directory schemes. These are presented next.

7.1.2 Broadcast Scheme (dir_iB)

In this scheme, when the pointer overflow occurs, a broadcast bit is set in the directory entry [134]. Subsequent reads to this block get the expected copy of the block leaving the associated directory entry state unchanged. The first write to the block after the pointer overflow triggers invalidations to be sent to all processing nodes in the system. Most of these invalidation messages go to processing nodes which have no copy of the block. These useless messages waste a great amount of network bandwidth and also put heavy load on the home node directory controller, thus delaying the completion of the invalidation and the write. Indirectly, they may also significantly slowdown subsequent (but otherwise unrelated) reads/writes to memory blocks due to hardware resource contentions. This dir_iB scheme performs poorly if the typical degree of sharing in an application is just larger than i. However, this scheme is advantageous due to its simplicity and low cost of implementation.

7.1.3 Coarse Vector Scheme (dir_iCV_r)

In the coarse vector scheme [56], when pointer overflow occurs, the memory used for storing the pointers is reorganized to store a coarse bit vector. Each bit in this vector represents a predefined fixed region consisting of r processing nodes. When there is at least one node in a region which has a valid cached copy, the corresponding region bit is set. Once a write occurs later, invalidations are sent to all the nodes in a region if the region bit is set. The value of r is typically determined by the number of bits available in a directory entry. The coarse vector scheme results in some extra invalidation traffic. However, such extra invalidations can be kept to a minimum by using regions with fewer nodes. Earlier research [56] has shown that for a medium sized DSM system using 3-4 pointers per directory entry, the dir_iCV_r scheme can perform reasonably well compared to the full-map scheme.

Several other limited directory schemes have been proposed in the literature. These include Limitless [81], Superset [3], Eviction [24], and Dynamic-vector [100]. Our main objective in this chapter is to demonstrate how efficient limited directory schemes can be designed by taking advantage of multidestination message passing support from the underlying network. Thus, without loss of generality, we only consider two representative limited directory schemes (broadcast schemes and coarse-vector scheme) in this chapter. In Sections 7.2 and 7.3, we show how dir_iB scheme can perform better with multidestination message passing-based broadcast support. Similarly, we show how variations to dir_iCV_r scheme can be designed to take advantage of multidestination message passing-based broadcast/multicast to deliver better performance. Before showing these new schemes, we provide a brief overview of multidestination message passing in the following section.

7.2 Limited Directory Schemes with Multidestination Messages

In this section, we present three variations of limited pointer schemes which work effectively with multidestination message passing. All these schemes use the multidestination mechanisms discussed in Section 6.3 of Chapter 6, especially the MR-MA framework. The main idea is to use multidestination messages to send the invalidations and collect the acknowledgments as required by the coherence protocols. First we show a variation of the dir_iB scheme. Next, we show how the coarse vector scheme (dir_iCV_r) [56] can be enhanced. For the rest of the chapter, we focus on 2D mesh DSM systems with dimensioned order based routing. However, the proposed variations are general and can be equally applied to other k-ary n-cube topologies (3D and higher dimensional meshes/tori) and other routing schemes (planar [26], and fully-adaptive [43], and turn-model [55]). As an example, extending the mechanisms to 3D topology is presented in Section 7.2.4.

7.2.1 Enhanced Broadcast Scheme

This scheme is similar to the dir_iB scheme previously discussed in Section 7.1 except that the invalidation broadcasts and acknowledgment collections are handled using multidestination DSM mechanisms. In order to take greater advantage of such mechanisms, the scheme uses hierarchical multidestination messages to broadcast invalidations. Let us define the row in which the home node exists as *home row.* Let a *row leader* be a node which can cover the remaining nodes of a row with a single multidestination multicast worm. In systems supporting dimensional order routing, it can be easily observed that the nodes at either end of a row are the row leaders.

Under the enhanced scheme, to broadcast invalidation messages after the pointer overflow occurs, the home node generates a maximum of two multidestination worms called *upturn* and *downturn* worms respectively. The *upturn* worm covers the row leaders of all rows north of the *home row.* Similarly, the *downturn* worm covers the row leaders of all rows south of the *home row.* The row leader of the home row is typically covered by the *upturn* worm. If no *upturn* worm is generated (for nodes on the top row) then the row leader of the home row is covered by the *downturn* worm. These two worms are called *level-1 inval* messages, as shown in Fig. 7.1. When each row leader receives a *level-1* inval message, it generates a reservation worm[14] and a gather worm to cover the nodes along its row. The reservation worms are called *level-2 inval* messages, and the gather worms are called *level-2 ack* messages. It is noted that each of these multidestination worms conforms to the base dimensional order routing.

A reservation worm always moves ahead of its corresponding gather worm. There are two advantages with this design. First, it leads to a simpler hardware for handling reservation and gather worms at the routers. Second, it results in a pipelining effect between the worms. Such pipelining reduces the effective turnaround latency starting from the time the invalidations are spawned to the time the acknowledgments are collected along a row. This pipelining effect also shortens the period during which an i-ack buffer entry at a router interface is held for an invalidation. This reduces the average number of i-ack buffer entries required per router and the percentage of i-ack misses. Once a gather worm reaches the end node on its path, it gets absorbed by the node and

[14]A reservation worm is a multicast worm which also tries to reserve an i-ack buffer at the router interfaces of its destinations, as discussed in Section 6.3.

a unicast message is sent back to the home node. Such messages are identified as *miack* messages in Fig. 7.1. Each *miack* message carries a count reflecting the number of acknowledgments its corresponding gather worm collects on its way.

For a detailed understanding of how the m_dir_iB scheme works and its inherent benefits, let us consider a typical invalidation broadcast on an 8×8 DSM system, as shown in Fig. 7.1. For easier illustration, all gather worms are assumed to have 100% i-ack hits, as discussed in Section 6.3. In our simulation experiments, we consider both i-ack hits and i-ack misses with a fixed number of i-ack entries. The number in each square bracket next to a message identifies the communication step in which the message propagation takes place. It can be easily observed that an invalidation message reaches each node of the system in no more than three communication steps and the acknowledgments are collected in two additional communication steps. Thus, the entire event of invalidation and acknowledgment collection completes in five communication steps. In this example, only 26 messages are used in the network for invalidation and acknowledgment collection. In general, for a $k \times k$ mesh, $3k + 2$ messages are required. In the u_dir_iB scheme (dir_iB scheme with unicast message passing), the corresponding invalidation and acknowledgment collection in this 8×8 mesh completes in 64 communication steps with 126 messages. The 64 steps are based on a best case assumption where maximum overlapping occurs between the home node sending invalidation messages and receiving acknowledgment messages. In general, $2(k^2 - 1)$ messages with k^2 communication steps (best case) are required for a $k \times k$ mesh using the u_dir_iB scheme.

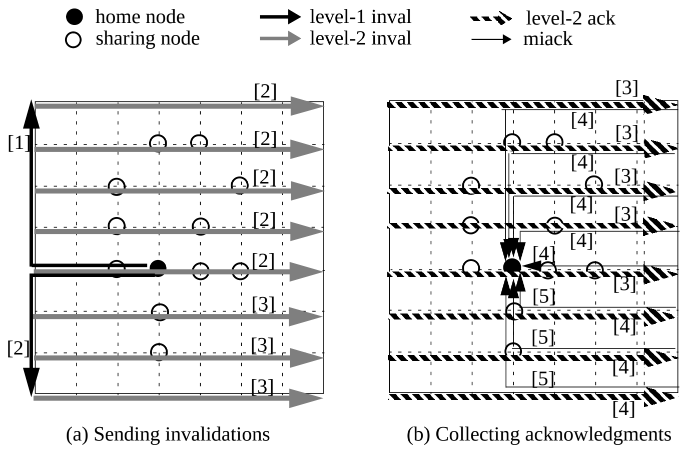


Figure 7.1: An example of invalidation broadcast using the m_dir_iB scheme.

Let us compare the home node occupancy [61] between these two schemes. It can be observed that a home node in the u_dir_iB scheme requires handling of $2(k^2 - 1)$ unicast messages. In the m_dir_iB scheme, it needs to handle only $(k+2)$ messages. Such a reduction in both network traffic and home node occupancy makes the m_dir_iB scheme quite attractive for reducing invalidation latency as well as write latency. The simulation results in Section 7.3 confirm these observations.

In the scheme mentioned and illustrated above, *row leaders* are assumed to be the left-end nodes in every row. However, this may put extra load on these nodes as well as extra load on +x links. To alleviate such bottlenecks, we propose a little variation to this scheme. In the new version, depending on whether the home node is on the left or right half of the mesh, we select the left or the right node in a row as the row leader. This variation also reduces the distances traveled by the two level-1 invalidation worms. Since the propagation time of these two worms is on the critical path of the entire invalidation broadcast, the new scheme helps in improving the invalidation latency. We use this variation for the m_dir_iB scheme in our simulation experiments described in Section 7.3.

7.2.2 Enhanced Coarse Vector Schemes

The principle behind the coarse vector scheme, as discussed in Section 6.3, is to rearrange the pointer bits into region bits when the pointer overflow occurs. In systems using unicast message passing, such regions are determined in a naive manner with an attempt to minimize the size of each region. In this section, we first show how regions can be defined in an intelligent manner under multidestination message passing. Next, we propose two variations to the dir_iCV scheme.

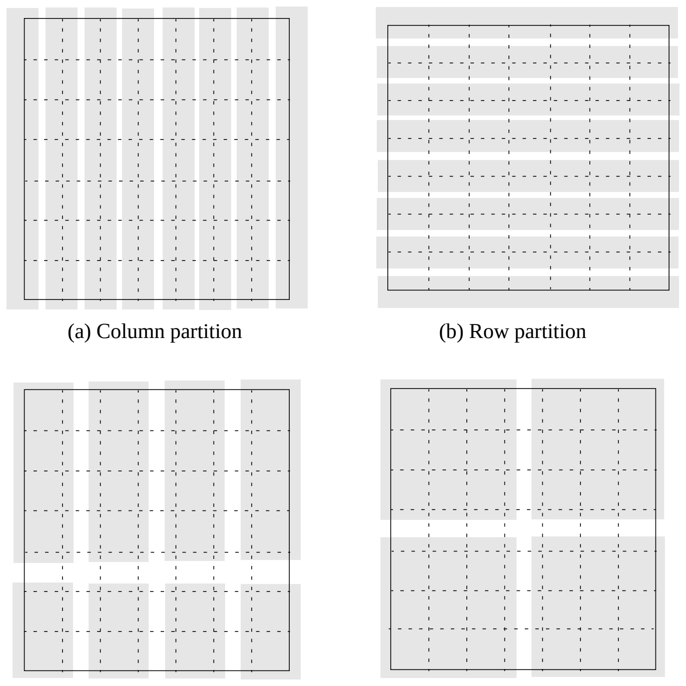
Region Partitioning under Multidestination Message Passing

Under multidestination message passing, partitioning a system into regions depends on three objectives: 1) regions should be efficiently covered using a few multidestination worms, 2) the traversal paths for these worms should be minimal, and 3) the paths should conform to base routing. For dimensional order routed meshes, simple multidestination paths are along row or column. Thus, region partitioning can also be done in a similar manner.

Figure 7.2 shows different partitioning schemes for an 8×8 mesh with different numbers of available pointers. If the directory has 2 pointers (6-bit wide each) then the system can be partitioned into a maximum of 12 regions after pointer overflow occurs. Figures 7.2(a) and (b) show how a system can be partitioned into 8 regions. In these figures, regions are defined along columns and rows, respectively. Figures 7.2(c) show a partitioning scheme where two-neighboring sub-columns are combined together to define a region. Finally, Fig. 7.2(d) shows a partitioning scheme with only four regions. Such a partitioning can be used when the directory has only one pointer with 6-bits. Next, we describe two variations to the coarse vector scheme using different schemes for region partitioning.

Equal-sized Region Scheme

This scheme (defined as multidestination column broadcast scheme 1) uses equal sized regions where each region is a fixed number of adjacent columns. For example, on an 8×8 system with two pointers, this scheme uses a single column as a region as shown in Fig. 7.2(a). With one pointer, two adjacent columns are defined as a region. For sending invalidation messages to those regions having at least one sharer, the home node sends out at most two *level-1 inval* multidestination multicast worms along its row in opposite directions. This is illustrated in Fig. 7.3(a). Nodes along the *home row* are defined as *column leaders* for their respective columns. The eastbound worm covers the *column leaders* to the east of the home node whose region bits are set. Similarly, the westbound worm covers *column leaders* to the west of the home node whose region bits are set. Each column leader, on receiving a *level-1 inval* message, generates two (northbound and southbound) *level-2 inval* messages and two (northbound and southbound) *level-2 ack* (gather) messages covering the nodes along its column. If the column containing the home node needs to be invalidated, the home node serves as its column leader. It should be noted that each of these messages is a multidestination worm conforming to the base dimensional order routing.



(a) Column partition

(b) Row partition

(c) 2-neighboring subcolumn partition

(d) 4-adjacent subrow partition

Figure 7.2: Examples of different region partitioning schemes under multidestination message passing for an 8×8 mesh.

Let us consider the time required to implement the complete invalidation event as shown in Fig. 7.3. In this example, only 4 communication steps are required to send invalidation messages to all regions having at least one sharer. It takes an additional 3 communication steps to collect the acknowledgments. Thus, the entire event gets completed in 7 steps. A total number of 26 messages are required to complete this event. Let us compare this performance with a u_dir_2CV scheme on this example sharing pattern. Assuming a similar region partitioning being used by the u_dir_2CV scheme, it takes 32 communication steps (with maximum overlap between sending invalidation messages and receiving acknowledgment messages) and 62 messages to implement the invalidation event. Thus, compared to the u_dir_2CV scheme, the proposed $m\_dir_iCB_1$ scheme clearly demonstrates potential to reduce home node occupancy, number of invalidation messages, network traffic, and invalidation latency. Simulation results in Section 7.3 confirm such benefits and show overall performance improvement in program execution time.

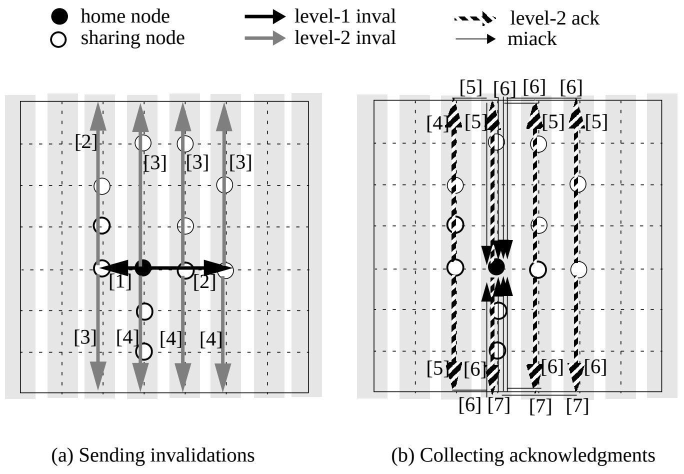


Figure 7.3: Illustration of an example invalidation event using the $m\_dir_iCB_1$ scheme on an 8×8 mesh. The home node and the sharers are assumed to be the same as shown in Fig. 7.1.

Variable-sized Region Scheme

In the above scheme, a column leader is required to generate up to four multidestination worms and inject them sequentially into the network. We propose a new scheme, defined as $m\_dir_iCB_2$ (multidestination column broadcast scheme 2), to solve this problem. In this scheme, the home node divides each column into two sub-columns according to the home node's position. The upper sub-column contains the nodes to the north of the home node as well as the node on the *home row.* The lower sub-column contains the nodes to the south of the home node. A region contains a fixed number of sub-columns, which are either upper sub-columns or lower sub-columns. Figure 7.4 illustrates such a region partitioning for an 8×8 mesh with 2 pointers. This partitioning consists of 8 regions.

For invalidation, the home node generates at most two multidestination *level-1 inval* messages, one westbound and one eastbound. This is similar to the $m\_dir_iCB_1$ scheme described earlier.

However, in this scheme, a *level-1 inval* message contains information about whether a sharer exists in the upper or lower sub-column corresponding to the column leaders being traversed by this message. Each column leader, after receiving the *level-1 inval* message, generates one or two *level-2 inval* and *level-2 ack* message pairs. Each pair covers the nodes along its sub-column. The sub-columns containing the home node are serviced by the home node, if required. It is noted that due to the additional information being carried in *level-1 inval* messages, less number of *level-2* messages are generated by the column leaders.

Similar to the earlier schemes, let us consider the time required to implement the complete invalidation event as shown in Fig. 7.4. In this example, only 4 communication steps are used to send invalidation messages and an additional 3 communication steps are used to collect acknowledgments. However, compared to the $m\_dir_iCB_1$ scheme, it uses less number of messages (20 instead of 26) to achieve the invalidation event. Thus, this scheme promises better performance compared to the $m\_dir_iCB_1$ scheme.

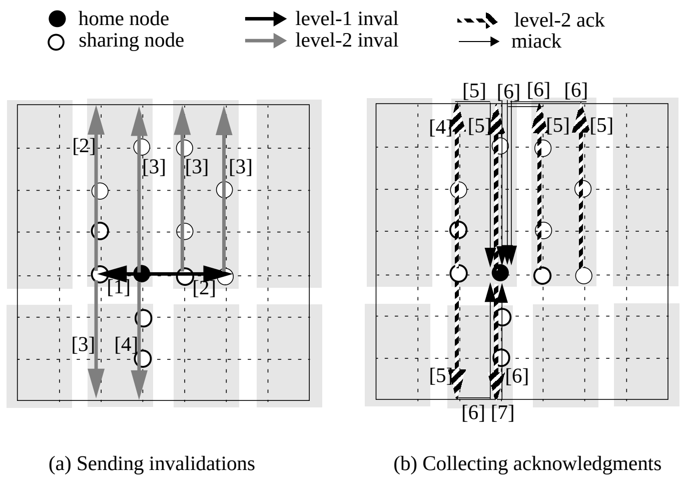


Figure 7.4: Illustration of an example invalidation event using the $m\_dir_iCB_2$ scheme on an 8×8 mesh. The home node and the sharers are assumed to be the same as shown in Fig. 7.1.

7.2.3 Overall Comparison

Let us compare the performance of the proposed schemes. It can be easily observed that $m\_dir_iCB_2 > m\_dir_iCB_1 > m\_dir_iB$. (where $>$ indicates better performance). As the performance for these schemes increases, the hardware complexity required at the directory controller also keeps increasing. For the m_dir_iB scheme, the directory controller requires nothing more than the hardware for a full-map directory controller with additional hardware for pointer overflow detection. Thus, the hardware complexity of the directory controller for the m_dir_iB scheme is the same as that of the u_dir_iB scheme. For this scheme, the routing headers for *level-1 inval*, *level-2 inval*, and *level-2* ack messages are fixed and simple. For the $m\_dir_iCB_1$ scheme, simple hardware to extract complete or partial column address (the region ID) from a node address must be added to the directory controller required by the m_dir_iB scheme. For the $m\_dir_iCB_2$ scheme, additional information also needs to be carried by *level-1 inval* messages to the column leaders. In our simulation experiments, we have considered such an increase in message lengths as well as increased time for initiating multidestination messages.

7.2.4 Systems with Other Topologies

In this subsection, we look at grouping schemes for a system with 3D mesh as an example for systems with other topologies.

Axis-Region One-step Reservation Scheme

Now, let us first consider the network using a popular e-cube (X-Y-Z) routing algorithm. Let us define a region as the set of nodes having the same values in X-Y index pair. In other words, a region consists of all the nodes along the same Z axis. Let the directory stores region presence bits using Y-X indexing order. Assume that the region presence bits are set suitably after a sequence of remote read operations. When an invalidation occurs, the region presence bits are checked. Depending on the position of the home node, a multidestination reservation worm is sent by the home node to cover each region which contains at least one copy of the block, as shown in Fig. 7.5(a). Note that a pseudo-sharing node is a node within a sharing region but does not actively share the memory block. The acknowledgment phase, as shown in Fig. 7.5(b), proceeds identically to the (AB1R) scheme. Let us denote such a scheme as an *axis-region one-step reservation (AR1R)* scheme.

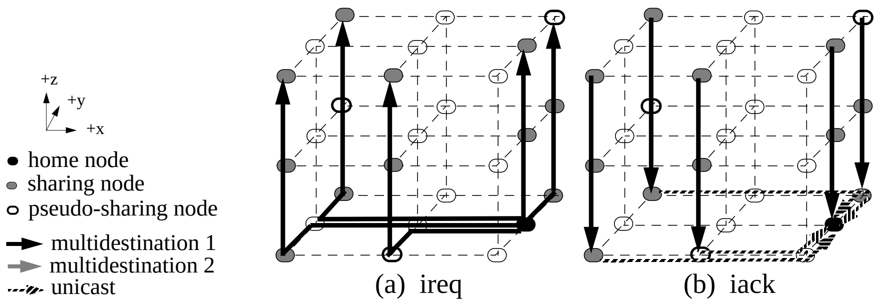


Figure 7.5: Axis-region one-step reservation (AR1R) scheme in a 3D mesh supporting e-cube routing using limited directory. The two phases of an invalidation transaction: (a) request and (b) acknowledgment.

In the above example, the AR1R scheme uses 5 invalidation reservation and 9 gather/unicast (14 in total) messages to complete the invalidation. While a unicast-based scheme needs 14 invalidation request and 14 acknowledgment (28 in total) messages to implement the same invalidation. Hence, the network traffic is reduced by one half. The grouping algorithm for the multidestination worms is simpler than a unicast worm. In general, significant reduction in network traffic and latency for invalidation transaction can be expected.

Axis-Region Two-step Reservation Scheme

In the second scheme, let us consider the network using an *all-but-X-negative-first* routing algorithm. Let us again define a region as the set of nodes along the same Z axis. Let the directory stores region presence bits using X-Y indexing order. When an invalidation occurs, depending on the distance from the home node, either top or bottom X-Y plane is chosen to the plane of potential leaders. A node lying on the intersection of this plane and a region whose presence bit is set becomes the actual leader for that region. Using the presence bit information, region leaders can be covered by one or multiple multidestination reservation messages (multidestination messages of type 1). Once receiving the reservation message of type 1, a region leader sends a multidestination reservation message of type 2 along the Z axis to cover the nodes in the region, as shown in Fig. 7.6(a). The acknowledgment phase, as shown in Fig. 7.6(b), proceeds as before. Let us denote such a scheme as an *axis-region two-step reservation (AR2R)* scheme.

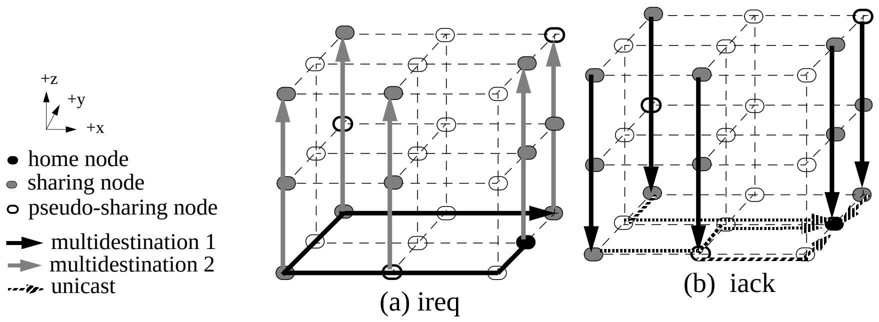


Figure 7.6: Axis-region two-step reservation (AR2R) scheme in a 3D mesh supporting all-but-X-negative-first turn model routing using limited directory. The two phases of an invalidation transaction: (a) request and (b) acknowledgment.

In the above example, the AR2R scheme uses 6 invalidation reservation and 9 gather/unicast (15 in total) messages to complete. It is noted that the home node starts 2 messages, 1 of type 1 multidestination and 1 of type 2 multidestination, for the invalidation. The grouping algorithm for the multidestination worms of type 1 is to be done on a 2D mesh topology, which can be done efficiently as evaluated in the last chapter. In general, compared to the unicast-based scheme, significant reduction in network traffic and latency for invalidation transaction can be expected using this scheme.

Plane-Region Two-step Reservation Scheme

Lastly, let us again consider the network using an *all-but-X-negative-first* routing algorithm. Let us define a region as the set of nodes in the same X-Y plane. Let the directory stores region presence bits using Z indexing order. When an invalidation occurs, the presence bits are used directly by the home node to send one or multiple multidestination reservation messages (multidestination messages of type 1) to the leaders of sharing planes. Once receiving the reservation message of type 1, a plane leader sends a multidestination reservation message of type 2 to cover all the nodes in the plane, as shown in Fig. 7.7(a). The acknowledgment phase, as shown in Fig. 7.7(b), proceeds as before. Let us denote such a scheme as a *plane-region two-step reservation (PR2R)* scheme.

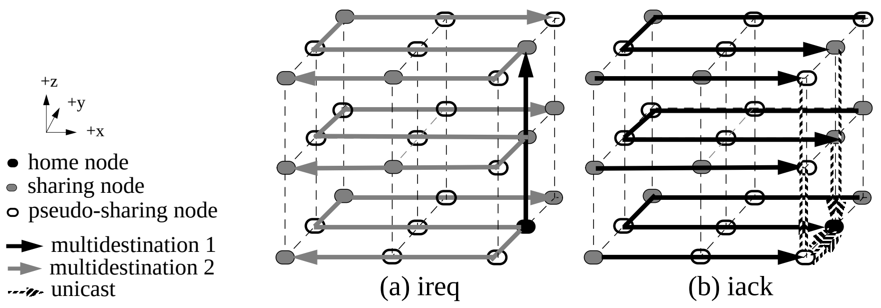


Figure 7.7: Plane-region two-step reservation (PR2R) scheme in a 3D mesh supporting all-but-X-negative-first turn model routing using limited directory. The two phases of an invalidation transaction: (a) request and (b) acknowledgment.

In the above example, this scheme uses 7 invalidation reservation and 11 gather/unicast (18 in total) messages to complete the invalidation. While a unicast-based scheme needs 26 invalidation request and 26 acknowledgment (52 in total) messages to implement the same invalidation. It is noted that the home node starts 3 messages, 1 of type 1 multidestination and 2 of type 2 multidestination. Clearly, the grouping for the type 1 multidestination is very simple. The grouping for the type 2 multidestination can be done statically. Thus, no noticeable delay should incur at the region leaders. Also, very little storage space is needed for the region presence bits. However, due to the large granularity of the region, the average occupancy of a node for an invalidation might increase. Whether this scheme delivers better performance compared to the Alewife's LimitLESS scheme or not depends on a lot of system parameters and characteristics of the applications. Detailed simulation (not carried out in this thesis for this scheme) may be required for a more precise evaluation.

It is noted that any of these region schemes can be incorporated statically (using hardware mode-switch) or dynamically (using software interrupt-driven mode-switch) with a few hardware sharing node pointers in each directory entry. Such a combination fits the invalidation patterns of most DSM applications and offers the most cost-effective system.

The goal of this subsection is to demonstrate potential extensions of our framework to a 3D mesh and other topologies. We do not emphasize the above schemes being optimal.

7.3 Performance Evaluation

7.3.1 Objectives and Simulation Experiments

To verify the effectiveness of some of our proposed schemes, we considered two system sizes: 8×8 and 8×4. Limited directory organizations with 1-3 pointers were considered. Depending on the pointer memory available in a directory entry, we did an optimal partitioning for each scheme. If the home node is the sole sharer in a region, a naive coarse vector scheme tries to cover all the nodes in the home node's region. Since invalidation to the home node does not require a network message, such a naive approach may generate unnecessary network traffic. To overcome this, we assumed one bit to be used as a *home bit* for all coarse vector schemes (u_dir_iB, $m\_dir_iCB_1$, and $m\_dir_iCB_2$). Table 7.1 shows the directory memory usage for all 32 schemes considered in this evaluation.

For all simulation experiments, we assumed system parameters representing the current trend in processor, memory, and interconnection network technologies. The following parameters were used: processor clocked at 200 MHz, communication link bandwidth of 200 Mbytes/sec, and router delay of 20.0 nanosec. For unicast-based message-passing, we assumed 10 processor clock cycles as the communication overhead (the startup time). For multidestination-based messages, this overhead was assumed to be 15 clock cycles. Table 7.2 lists all system parameters used in our simulation.

We considered four applications: Barnes (2K bodies, 4 steps), LU (128×128 matrix, 8×8 blocks), Radix (256K keys, 512 radix, 512K max), and Water (512 molecules, 4 steps).

7.3.2 Simulation Results

For each experiment, we observed five important parameters: total number of invalidation messages (including messages for acknowledgment collection), total number of network messages, average latency per invalidation, average latency per write operation, and overall execution time. For the overall execution time, we also observed the amount of time spent for computation, read, write, and synchronization. For a given application and system size, we normalized the above five parameters with respect to those obtained for the full-map unicast implementation for a fair comparison.

Barnes

Figure 7.8 shows the five parameters for the Barnes application on an 8×8 system. For each parameter, results of 16 different schemes are presented. This includes five schemes for each pointer (i=1, 2, and 3) and the full-map scheme. Results for all schemes are normalized to the full-map scheme (100%).

From Fig. 7.8(a) it can be observed that multidestination based schemes are able to significantly reduce invalidation traffic compared to the unicast based scheme. The m_dir_1B scheme performs the best for 1 pointer where as the $m\_dir_1CB_2$ scheme performs the best for 3 pointers. Most noticeably, the m_dir_1B scheme produces only about 20% of the invalidation traffic produced by the u_dir_1B scheme. Similarly, the $m\_dir_1CB_1$ scheme produces around 40% of the invalidation traffic produced by the $u\_dir_1CV_8$ scheme. Fig. 7.8(b) shows the impact on the overall network traffic. Similar observations can also be made here. The m_dir_1B scheme is able to bring down the overall network traffic closer to the full-map scheme.

Figures 7.8(c) and (d) show the impact of these schemes on average invalidation latency and average write latency, respectively. It can be observed that the m_dir_iB schemes are able to reduce both latencies considerably compared to the u_dir_iB and u_dir_iCV schemes. Fig. 7.8(e) shows the overall execution time. It can be observed that the multidestination-based schemes are able to perform closer to the full-map scheme. The m_dir_1B scheme shows closer performance to the full-map scheme. The $m\_dir_2CB_1$ and $m\_dir_2CB_1$ schemes are able to perform even better than the full-map scheme. The m_dir_1B scheme is able to perform within 5% of the $u\_dir_3CV_4$ scheme. Looking at the breakups in execution time, it can be observed that the reduction in overall execution time is sensitive to the reduction in write latency time. The multidestination schemes are able to

Scheme	Topology	Pointer Bits	Vector Bits	Home Bit	Unutilized Bits
$u\_fullmap$	8×8	64	N/A	N/A	0
u_Dir_1B	8×8	6	N/A	N/A	6
$u\_Dir_1CV_{16}$	8×8	6	4	1	1
m_Dir_1B	8×8	6	N/A	N/A	6
$m\_Dir_1CB_1$	8×8	6	4	1	1
$m\_Dir_1CB_2$	8×8	6	4	1	1
u_Dir_2B	8×8	12	N/A	N/A	12
m_Dir_2B	8×8	12	N/A	N/A	12
$u\_Dir_2CV_8$	8×8	12	8	1	3
$m\_Dir_2CB_1$	8×8	12	8	1	3
$m\_Dir_2CB_2$	8×8	12	8	1	3
u_Dir_3B	8×8	18	N/A	N/A	18
m_Dir_3B	8×8	18	N/A	N/A	18
$u\_Dir_3CV_4$	8×8	18	16	1	1
$m\_Dir_3CB_1$	8×8	18	8	1	9
$m\_Dir_3CB_2$	8×8	18	16	1	1
$u\_fullmap$	8×4	32	N/A	N/A	0
u_Dir_1B	8×4	5	N/A	N/A	5
m_Dir_1B	8×4	5	N/A	N/A	5
$u\_Dir_1CV_8$	8×4	5	4	1	0
$m\_Dir_1CB_1$	8×4	5	4	1	0
$m\_Dir_1CB_2$	8×4	5	4	1	0
u_Dir_2B	8×4	10	N/A	N/A	10
m_Dir_2B	8×4	10	N/A	N/A	10
$u\_Dir_2CV_4$	8×4	10	8	1	1
$m\_Dir_2CB_1$	8×4	10	8	1	1
$m\_Dir_2CB_2$	8×4	10	8	1	1
u_Dir_3B	8×4	15	N/A	N/A	15
m_Dir_3B	8×4	15	N/A	N/A	15
$u\_Dir_3CV_2$	8×4	15	16	1	1
$m\_Dir_3CB_1$	8×4	15	8	1	6
$m\_Dir_3CB_2$	8×4	15	16	1	1

Table 7.1: Directory memory usage for different schemes.

Parameter	Values	Time
Processor	1 cycle	5 ns
Cache access	1 cycle	5 ns
Cache block size	16 bytes	
Cache block fill time	32 cycles	160 ns
Cache set associativity	1	
Cache size per node	128 Kbytes	
Memory word width	4 bytes	
Memory block access time	32 cycles	160 ns
Directory check	5 cycles	25 ns
Invalidation message generation time	10 cycles	50 ns
Directory check and update	10 cycles	50 ns
Outgoing message startup (unicast)	10 cycles	50 ns
Outgoing message startup (multidestination)	15 cycles	75 ns
Incoming message dispatch	5 cycles	50 ns
Control message size (unicast)	6 bytes	
Control message size (multidestination)	8 bytes	
Data message size	22 bytes	
Injection channels per node	2	
Consumption channels per node	4	
I-ack buffer entries per router	2	
Channel width / Flit size	2 bytes	
Link Propagation	1 cycle	5 ns
Router switch delay	1 cycle	5 ns
Routing time	4 cycles	20 ns
Virtual networks	2	
Remote Unlock Latency: 8×4	40 cycles	200 ns
Remote Unlock Latency: 8×8	50 cycles	250 ns
Barrier Latency: 8×4	80 cycles	400 ns
Barrier Latency: 8×8	100 cycles	500 ns

Table 7.2: Default system parameters used in the simulation.

reduce write latency and hence, the overall execution time. These results clearly show the benefits of the multidestination based limited directory schemes.

Fig. 7.9 shows the corresponding results for Barnes on a 32-processor system. Similar trends can also be observed here. As shown in Fig. 7.9(a), for all pointers, the m_dir_iB scheme is able to reduce maximum amount of invalidation traffic. However, with respect to the invalidation latency and write latency, the coarse vector multidestination schemes ($m\_dir_iCB_1$ and $m\_dir_iCB_1$) are able to perform better than the m_dir_iB schemes. This indicates that reducing the number of invalidation messages need not directly lead to reduction in invalidation and write latencies. Fig. 7.9(e) shows the impact of all schemes on the overall execution time. It can be observed that the m_dir_1B and $m\_dir_1CB_1$ schemes are able to come closer (within 1-2%) to the full-map and $u\_dir_3CV_2$ schemes. This clearly suggests that limited directory schemes with a single pointer each can be very effective when used in conjunction with the multidestination message passing mechanism.

LU

Figure 7.10 shows the results of LU on an 8×8 system. LU is a computationally intensive application. Thus, the schemes demonstrate different characteristics. As the number of pointers increases, all schemes are able to reduce the invalidation traffic. For 1-pointer, the $m\_dir_1CB_1$ and $m\_dir_1CB_2$ schemes are able to perform the best. However, with 3 pointers, the m_dir_3B scheme is able to reduce the invalidation and overall network traffic considerably. Figs. 7.10(c) and (d) show the impact on the invalidation latency and write latency, respectively. It can be observed that with 1-pointer, the $m\_dir_1CB_1$ scheme is able to reduce the latencies considerably. Compared to the u_dir_1B scheme, the $m\_dir_1CB_1$ scheme is able to reduce the average invalidation latency by a factor of 7.0 and average write latency by a factor of 3.2. Fig. 7.10(e) shows the impact on execution time. Since the LU is a computationally intensive application, few changes are observed with a higher number of pointers. With 1-pointer, the m_dir_1B scheme is able to reduce the overall execution time by 18% compared to the u_dir_1B scheme. The $m\_dir_1CB_1$ scheme is able to come closer to the full-map scheme.

Figure 7.11 shows the results for LU on an 8×4 system. Here the benefits of multidestination message passing are less compared to the 64 processor system. For 1-pointer, the multidestination schemes are able to reduce the invalidation traffic, total network traffic, average invalidation latency, and write latency. For more pointers, these advantages diminish. With respect to the overall execution time, as shown in Fig. 7.11(e), it can be observed that all schemes are able to perform equally well.

Radix

Figure 7.12 shows the results for Radix on an 8×8 system. It can be observed that with 1-pointer, the m_dir_1B scheme is able to reduce the number of invalidation messages by a factor of 5.0 compared to the u_dir_1B scheme. The $m\_dir_1CB_1$ scheme is able to reduce the number of invalidation messages by a factor of 1.8 compared to the $u\_dir_1CV_8$ scheme. With more number of pointers, all schemes are able to perform equally well. Similar trends are also observed with respect to total network traffic, average invalidation latency, and memory latency. Figure 7.12(e) shows the impact on the overall execution time. It can be seen that the m_dir_1B scheme is able to reduce program execution time by a factor of 3.47 compared to the u_dir_1B scheme. Similarly, the $m\_dir_1CB_1$ scheme is 76% better than the $u\_dir_1CV_8$ scheme. The $m\_dir_1CB_1$ scheme is able

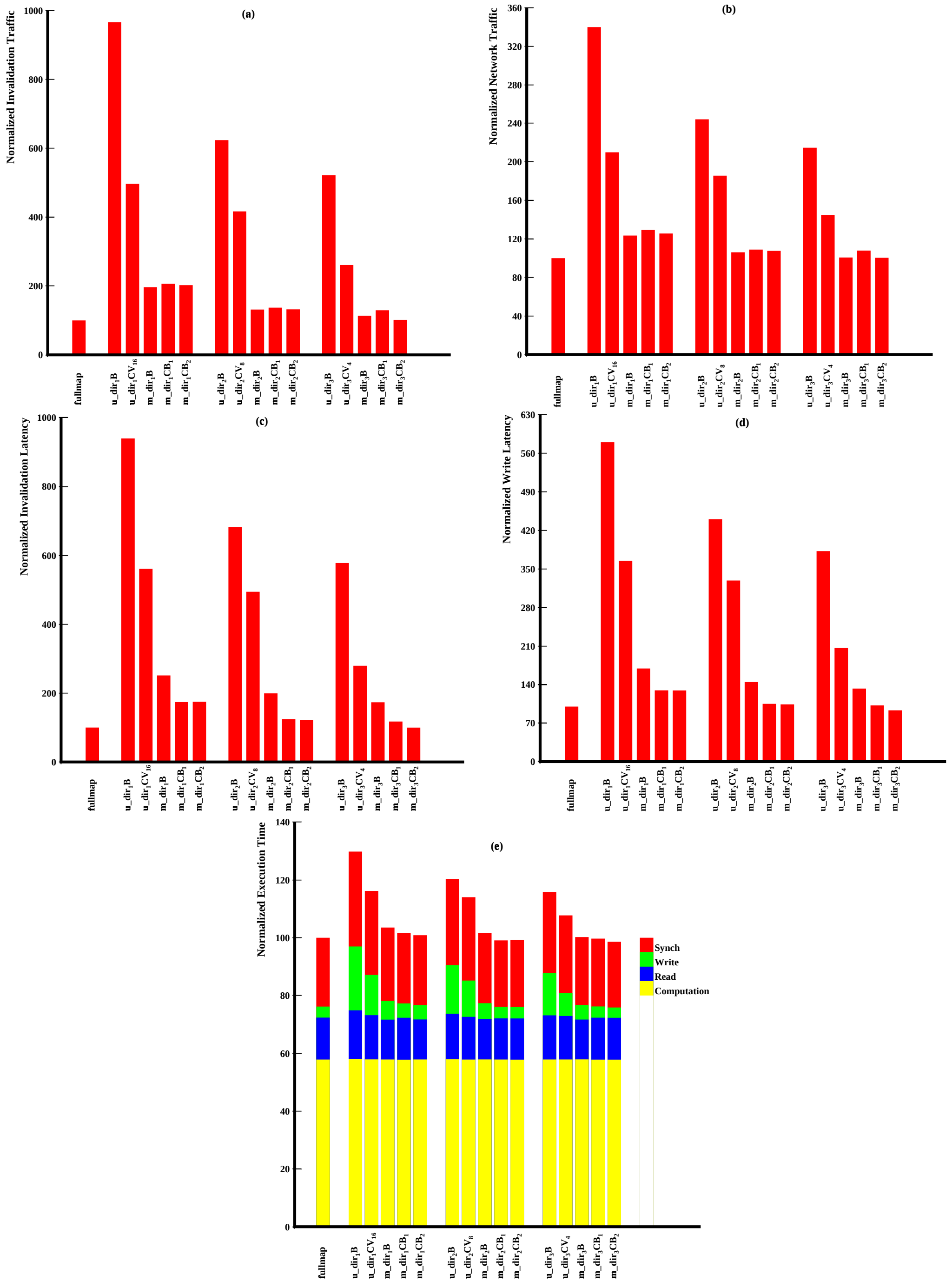


Figure 7.8: Performance comparison of different schemes on a 64 processor system using Barnes.

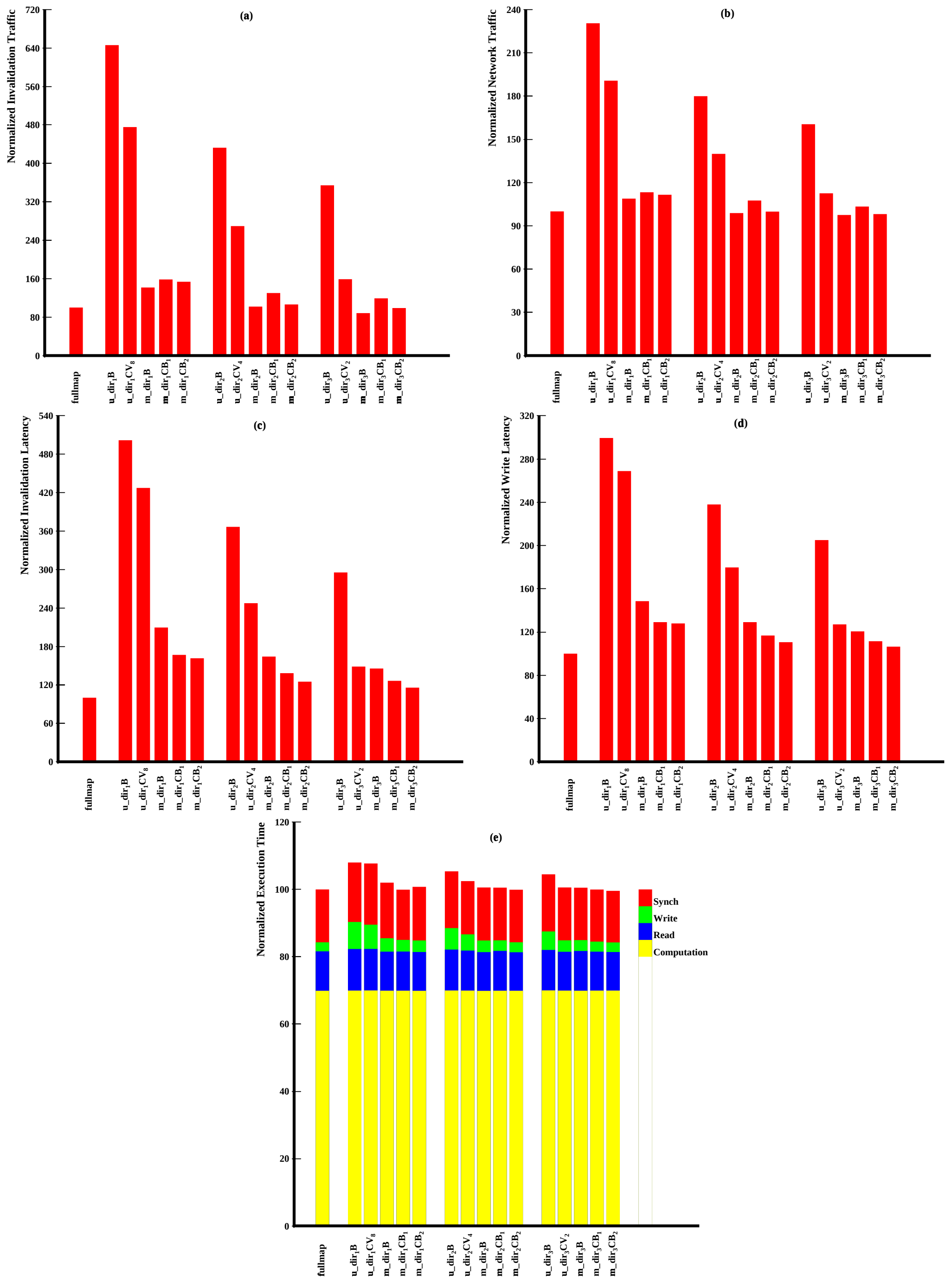


Figure 7.9: Performance comparison of different schemes on a 32 processor system using Barnes.

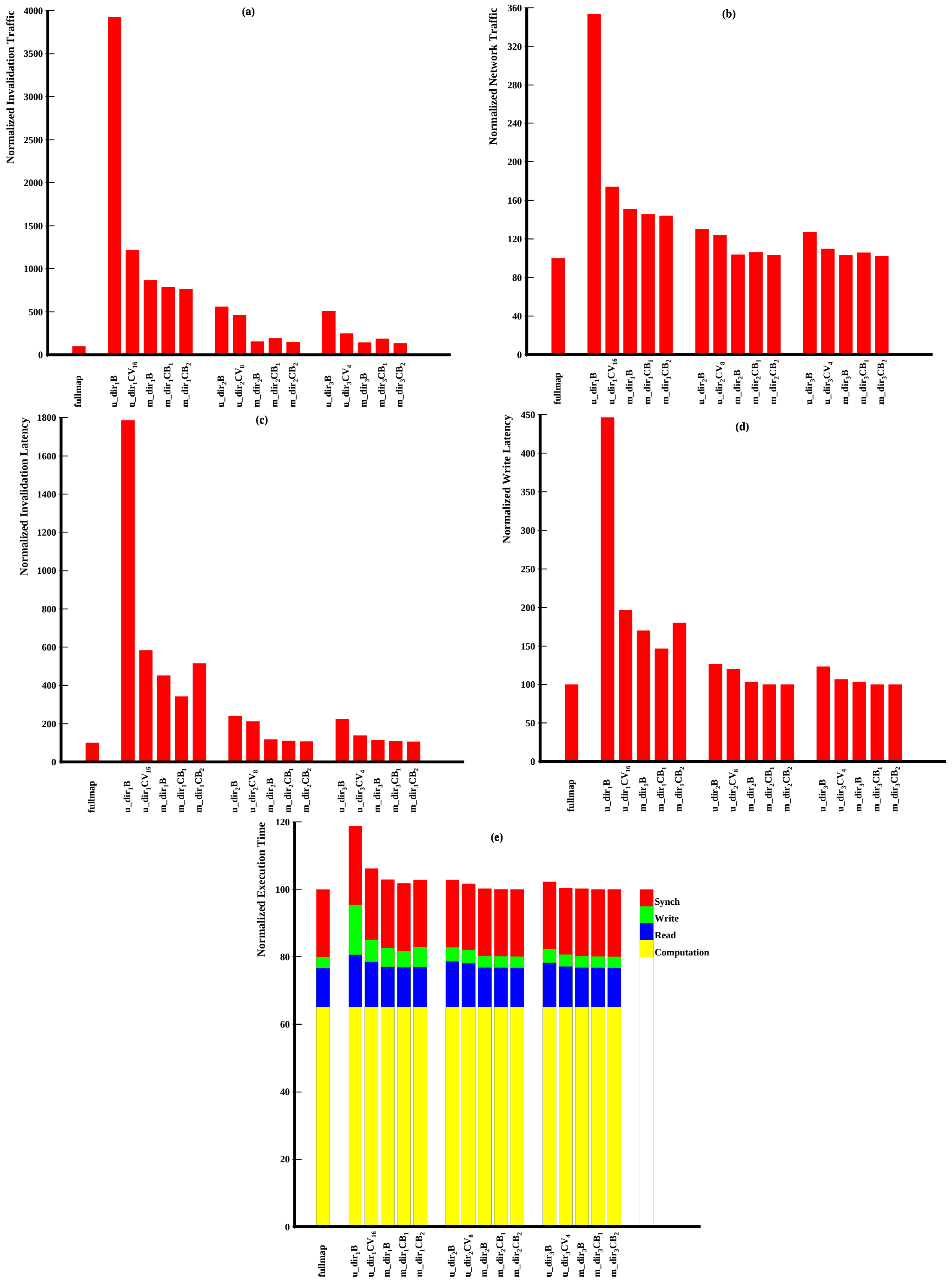


Figure 7.10: Performance comparison of different schemes on a 64 processor system using LU.

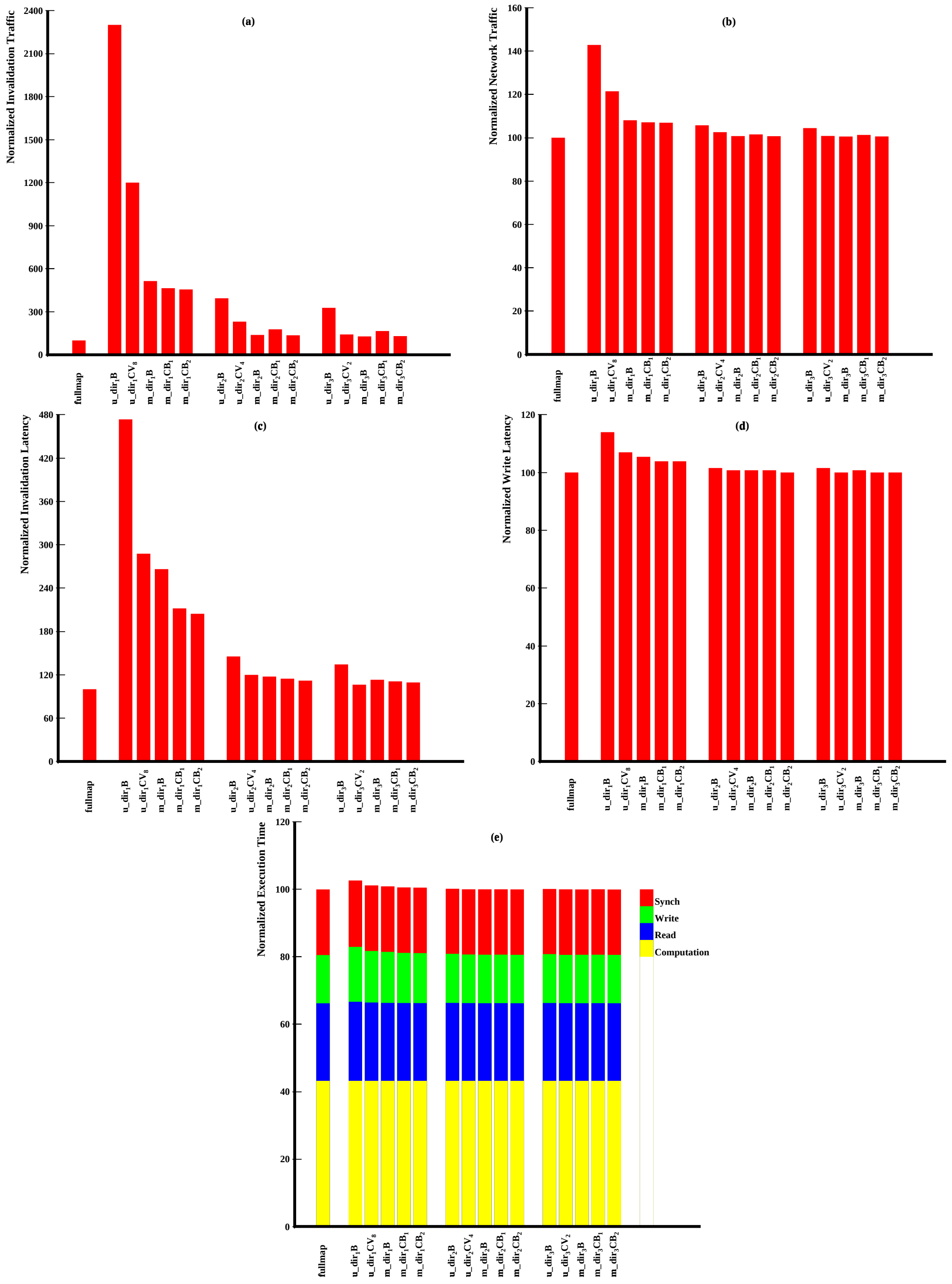


Figure 7.11: Performance comparison of different schemes on a 32 processor system using LU.

to bring the overall execution time closer to that of the full-map scheme as well as the $u\_dir_2CV_8$ and $u\_dir_3CV_4$ schemes.

Figure 7.13 shows the results of Radix on an 8×4 system. Similar trends are also observed here except that the dynamics of the schemes for 1-pointer get interchanged. With respect to 1-pointer, the $u\_dir_1CV_8$ scheme performs the best for invalidation messages, network traffic, average invalidation latency, and memory latency. However, with respect to the overall execution time, the $m\_dir_1CB_1$ scheme performs the best. The m_dir_1B scheme performs 90% better than the u_dir_1B scheme. The $m\_dir_1CB_1$ scheme performs 4% better than the $u\_dir_1CV_8$ scheme.

Water

Figure 7.14 shows the results of Water on an 8×8 system. It can be observed that the multidestination schemes are able to perform better compared to the unicast-based schemes. For 1-pointer, both the $m\_dir_1CB_1$ and the $m\_dir_1CB_2$ schemes are able to reduce the number of invalidation messages by a factor of 5.4 compared to the u_dir_1B scheme and by a factor of 1.5 compared to the $u\_dir_1CV_8$ scheme. A similar trend is observed for total network traffic. With respect to the average network latency and write latency, the $m\_dir_1CB_1$ scheme performs the best with one pointer. With more pointers, the m_dir_iB schemes perform the best. With respect to the overall execution time, with 1-pointer, the $m\_dir_1CB_1$ scheme is able to reduce the program execution time by a factor of 1.5. It comes closer to the execution time of the full-map scheme (within 5%) and coarse vector schemes (within 5%).

Figure 7.15 shows the results for Water on an 8×4 system. With 1-pointer, the m_dir_1B scheme is able to reduce number of invalidation messages by a factor of 7.7 compared to the u_dir_1B scheme. As the number of pointers increases, the m_dir_1B scheme is able to demonstrate its superiority. Similar trends are also observed for the overall network traffic. With respect to the average invalidation latency and write latency, the multidestination schemes are able to perform better. For 1-pointer, the m_dir_1B scheme is able to reduce the write latency by a factor of 1.9 compared to the u_dir_1B scheme. Similarly, the $m\_dir_1CB_1$ and $m\_dir_1CB_2$ schemes are able to reduce write latency by a factor of 1.3 compared to the $u\_dir_1CV_8$ scheme. With respect to the overall execution time, for 1-pointer case, all schemes except the u_dir_1B scheme perform equally well and they are comparable to the full-map scheme.

The results mentioned above show the following five common trends: 1) multidestination message passing can be used for both broadcast and coarse vector schemes to deliver better performance, 2) benefits of multidestination message passing are better with lower number of pointers, 3) multidestination message passing schemes with 1-pointer can deliver performance closer to the full-map directory, 4) performance of multidestination message passing schemes with 1-pointer comes closer to coarse vector and broadcast schemes with higher number of pointers, 5) benefits of multidestination message passing increase with an increase in system size. These trends suggest that future generation systems can take advantage of multidestination message passing with fewer pointers (even 1) to build scalable and cost-effective DSM systems.

7.4 Related Work

A number of other limited directory schemes have been proposed in the literature. Some representative ones are eviction [24], superset [3], gray-code [103], and dynamic-vector [100]. The dir_iNB [24] scheme tries to completely avoid any invalidation broadcast by evicting an existing sharing node under pointer overflow. The superset scheme [3] allows tri-state information with

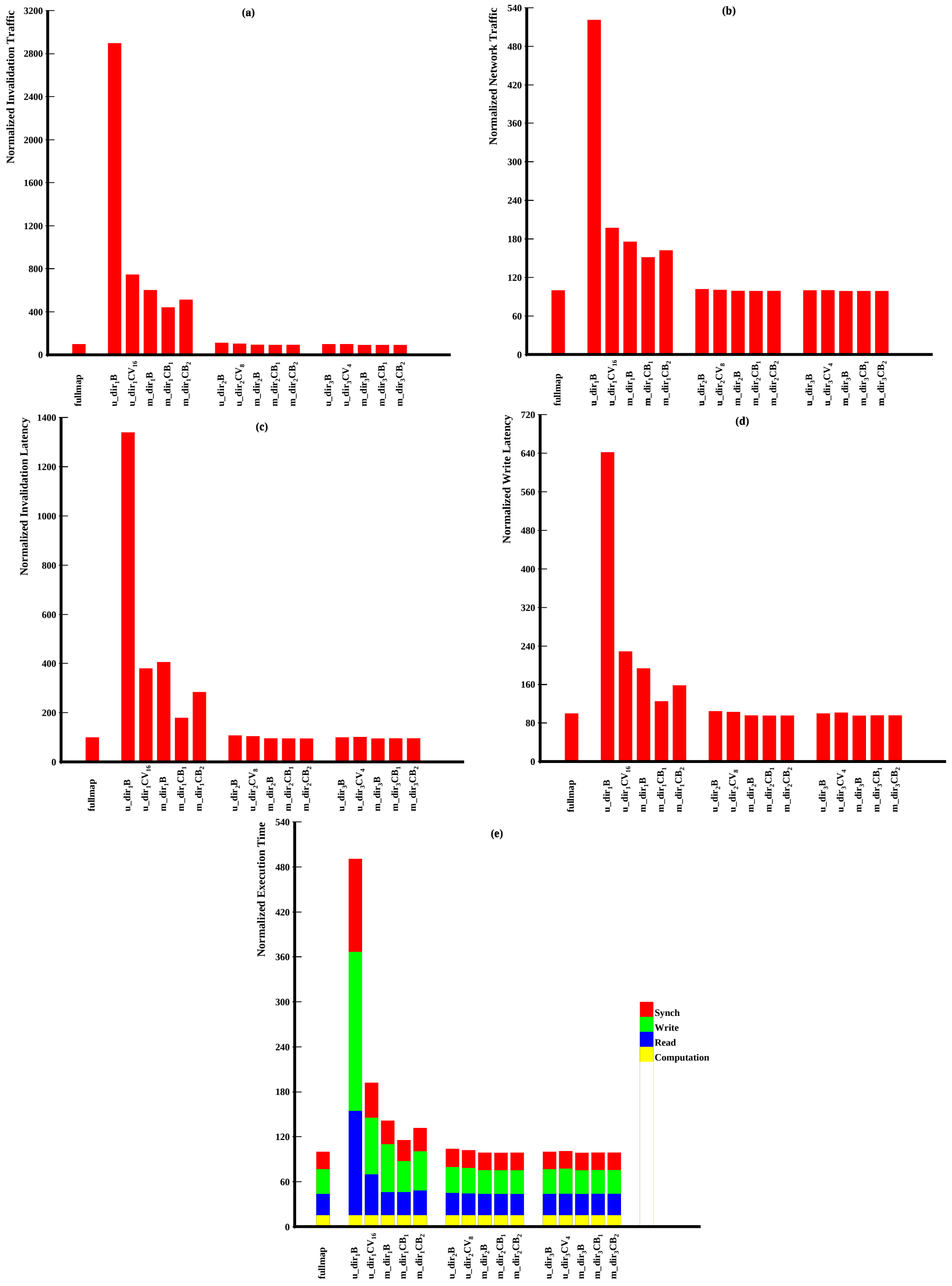


Figure 7.12: Performance comparison of different schemes on a 64 processor system using Radix.

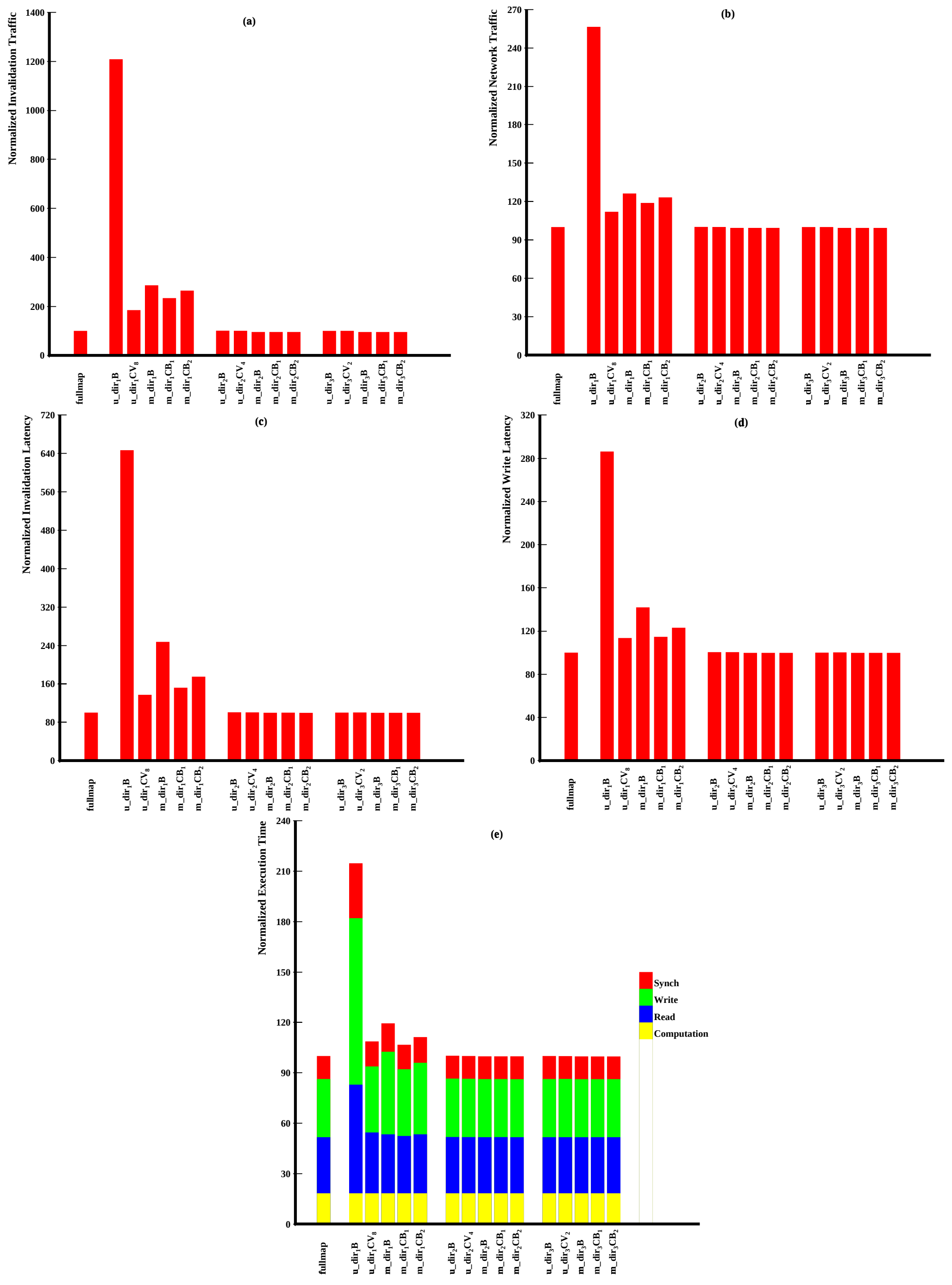


Figure 7.13: Performance comparison of different schemes on a 32 processor system using Radix.

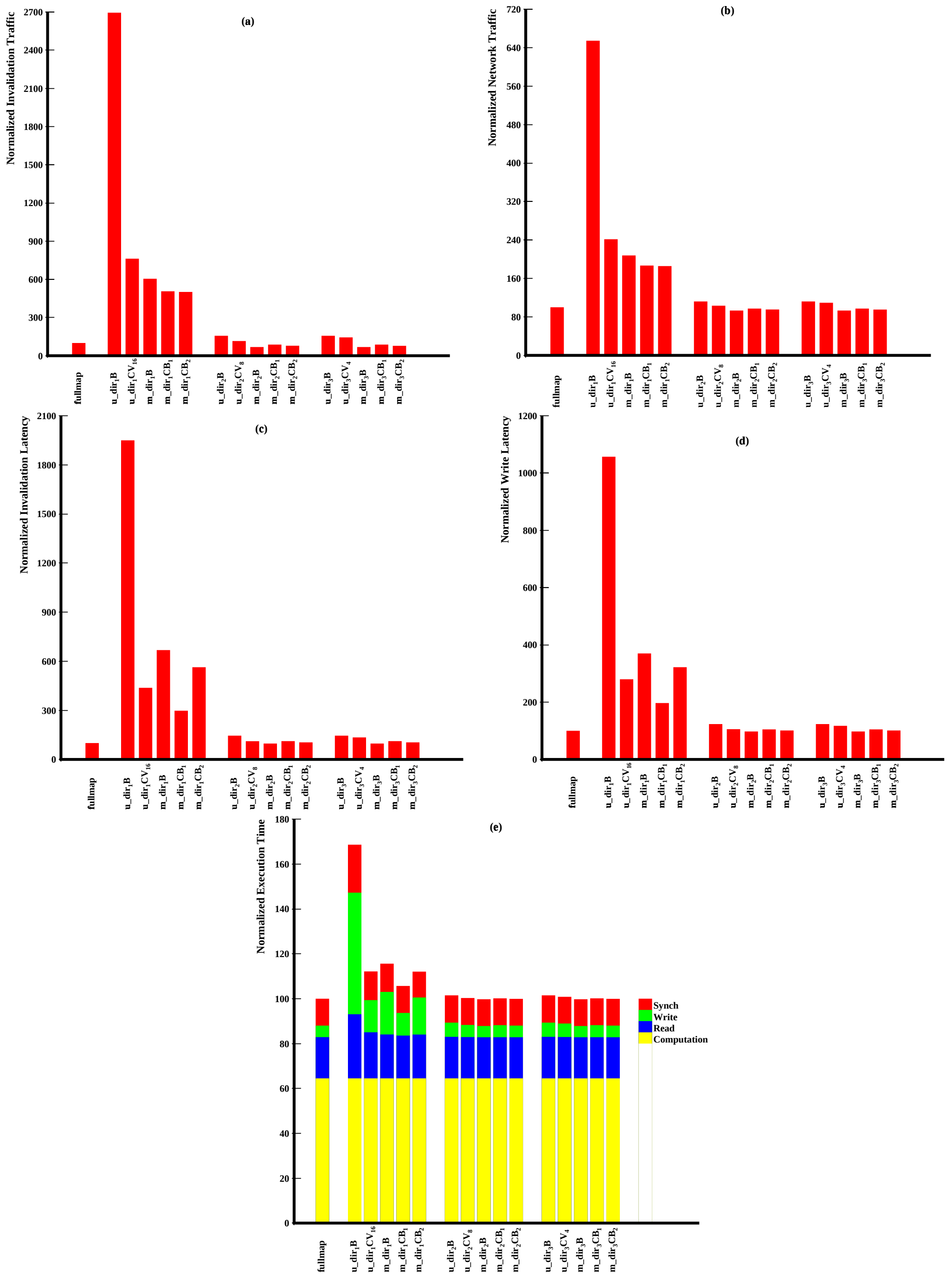


Figure 7.14: Performance comparison of different schemes on a 64 processor system using Water.

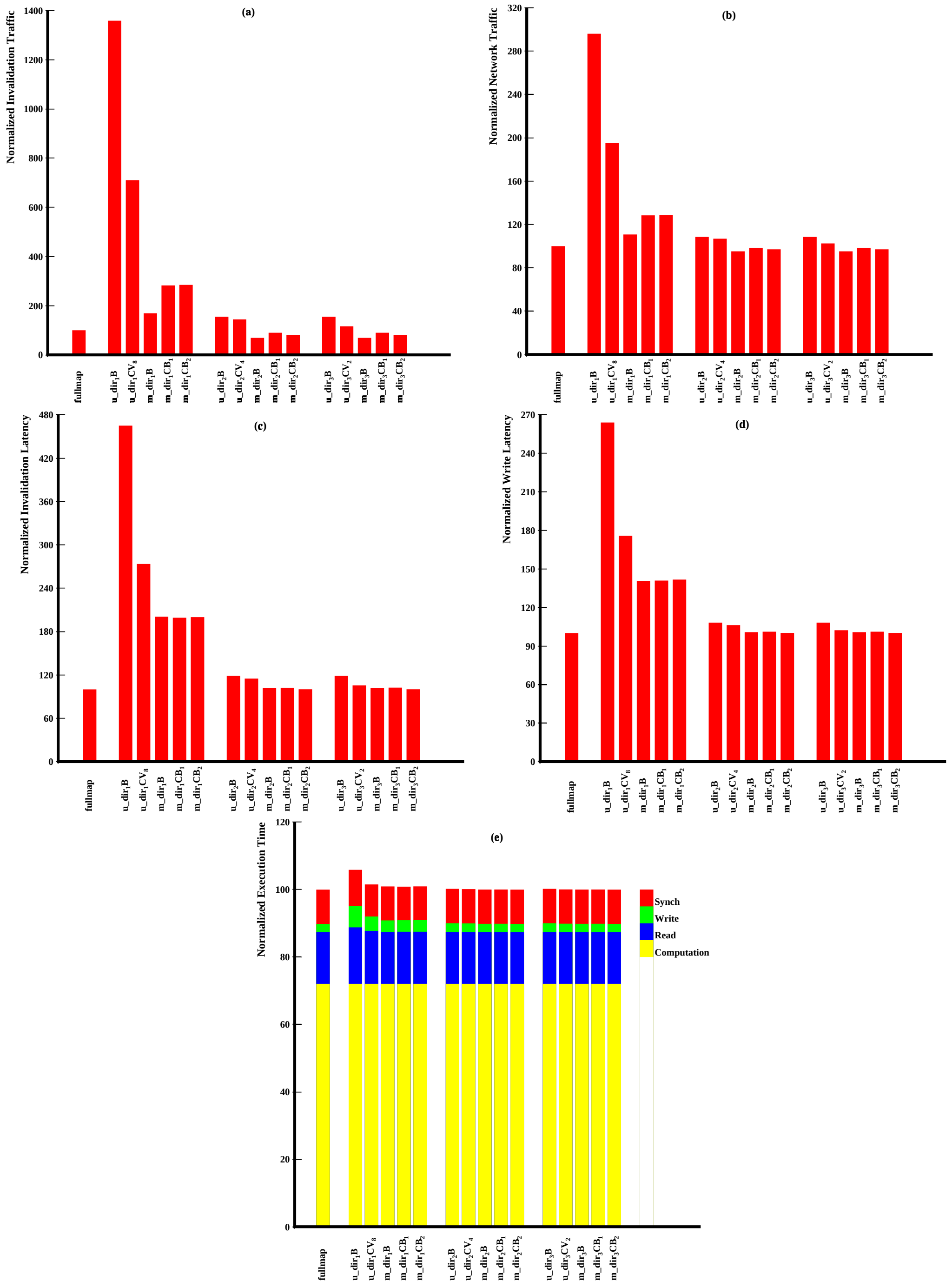


Figure 7.15: Performance comparison of different schemes on a 32 processor system using Water.

each directory entry being $2 \log P$ bits. A variation known as *gray-code* scheme [103] optimizes the superset scheme for near-neighbor sharing. A dynamic-vector scheme (dir_iDV_r) [100] optimizes the dir_iCV_r scheme by allowing the coarse vector granularity to vary dynamically according to the sharing pattern.

All of the above schemes have been proposed in the context of unicast message-passing. Our main objective in this chapter has been to demonstrate how efficient limited directory schemes can be designed by taking advantage of multidestination message passing support from the underlying network. Thus, instead of evaluating all limited directory schemes, we have considered two representative ones. For all other schemes, our multidestination message passing mechanisms can also be applied to improve the performance. The resulting complexity and performance gain can vary from one scheme to another depending on how well the multidestination messages are used in the schemes.

In addition to the hardware schemes discussed so far, there are some interesting software assisted limited directory schemes, like LimitLESS scheme [81] and dir_1SW^+ [148]. The main focus of such schemes is to use the existing hardware as effectively as possible to cope with lower sharing degrees and to deal with larger sharing degrees through software. In this chapter, we have focused only on hardware assisted limited directories. The concepts can also be extended to the software assisted schemes.

Besides hardware cache coherency, much research has been done for complete software DSM systems [93, 5]. Here the emphasis has been on avoiding false sharing as well as reducing coherence traffic. For networks supporting multicasting capability, our proposed schemes can also be used for reducing coherence traffic.

In the context of evaluating cache-coherence protocols together with network behavior, Bhuyan et al. have recently proposed novel schemes [83, 95]. They have shown that a single invalidation message with implicit multiple destinations along hierarchical rings, statically defined using embedded virtual channels on a hypercube machine, can be used to broadcast invalidation messages quickly. A distributed hierarchical directory organization/protocol is cleverly used in the proposed scheme. In our work, we do not use any such embedding. Instead, we only use base-routing-conformed paths to define the multidestination worms. We also use a combination of multidestination and gather worms to reduce the invalidation latency considerably.

7.5 Summary

Most existing limited directory schemes use only point-to-point network messages to implement invalidations. Such a design leads to severe performance degradation when the number of pointers per directory entry is very small. New generation networks are providing architectural support for efficient collective communication operations. In this chapter, we investigate the performance potential by exploring such support in designing more cost-effective limited directory schemes for DSM systems. The study is carried out for wormhole k-ary n-cube networks supporting the *multidestination* message passing mechanism. Our emphasis has been on variations of the u_dir_iB and u_dir_iCV schemes that work efficiently with (complete/selective) broadcast and gather operations with multidestination messages. We have proposed three schemes, the m_dir_iB, $m\_dir_iCB_1$, and $m\_dir_iCB_2$, and evaluated them for 32 and 64 processor systems with SPLASH2 benchmarks and compared their performance with limited directory schemes using unicast message passing (u_dir_iB and u_dir_iCV). We have shown that 1) the m_dir_1B (with a single pointer) can reduce the program execution time by up to a factor of 3.47 compared to the u_dir_1B scheme, and 2) the performance of the m_dir_1B (with *one* pointer) scheme is close to the performance of the u_dir_3CV scheme (with three pointers) and to the performance of the fully-mapped scheme. For 1-3 pointers, the $m\_dir_iCB_1$ and $m\_dir_iCB_2$ schemes are shown to further reduce program execution time compared to the m_dir_iB scheme. For 1 pointer, $m\_dir_1CB_1$ performs better than the u_dir_1CV scheme by a factor of up to 1.66. These results indicate that limited directory schemes with only a single pointer can be designed to work efficiently on new generation wormhole DSMs with multidestination message passing. Such results provide strong guidelines to build future scalable DSM systems in a cost-effective manner.

CHAPTER 8

DESIGNING UNBALANCED REQUEST AND REPLY NETWORKS

Having studied network support for specific high overhead transactions at the protocol layer such as cache invalidation in the previous two chapters, we now look at methods to enhance the network itself to make the protocol layer more efficient. Specifically, we examine the virtual channel mechanism in DSM systems to obtain more performance benefits.

Virtual channels [41] have been proposed as an attractive mechanism to avoid deadlocks [8, 43], provide adaptivity [20], increase throughput [44], and reduce latency in wormhole routed networks [44]. In recent years, several commercial network switching products [49, 124, 144] have incorporated virtual channel mechanism as a major feature for improving network performance. Out of these products, the SGI Spider interconnect [49] is directly geared towards cache-coherent DSM systems.

Similar to many other studies in interconnection networks, studies on using virtual channels have focused on synthetic traffic (uniform/non-uniform/hot-spot), arbitrary message lengths, and message generation intervals following some probabilistic distributions. Some crucial characteristics of DSM systems including the *cause-effect* relationship between messages (remote read/write request message followed by a reply message), *bi-modal* traffic (short messages reflecting control messages in a cache-coherency protocol and long messages reflecting transfer of cache-lines), and the periodic generation of messages by processors (based on the computational granularity) have not been considered. Most of the studies also ignore the impact of the network interface (NI) and assume that the NI contains infinite number of physical injection and consumption channels. Therefore, using the virtual channel mechanism in DSM systems is a topic worth further investigation.

Recently, an application-driven study [142] has shown that only a negligible performance benefit exists in using virtual channels and adaptive routing in a DSM system. According to this study, the enhancements due to virtual channels and adaptive routing might not be justified considering the associated increase in router complexity. However, four important factors have been overlooked in this study:

- An equal number of virtual channels have been used in the request and reply networks, ignoring the fact that a typical request message is much shorter than a typical reply message in a DSM system. The request and reply messages travel in separate (virtual) networks to avoid deadlocks [91]. Since the volume of traffic is not the same in the request and reply networks, it is not clear whether equal number of virtual channels assigned to each of these two networks can deliver good performance.
- Network interfaces having a single injection channel and a single consumption channel have been used to connect nodes to routers having multiple virtual channels. This puts a severe

limitation on the number of live messages in the network at any given time. Thus, increasing the number of virtual channels inside the network does not promise a performance benefit.

- In-order message delivery has been enforced by the network interface adapter, and a simple FIFO cache coherence protocol has been used. In such a design, messages arriving out-of-order (due to virtual channels) have to wait at the network interface adapter until the arrivals of their logical preceding messages. Such an approach prevents the directory controller of the destination node from processing these messages in a timely fashion. This results in an increase in effective message latency and nullifies the latency reduction gained by the virtual channel mechanism.

- A performance degradation model for implementing the virtual channel mechanism has been used in the evaluation [9]. However, new techniques [49] are available today to design routers with a moderate number of virtual channels (up to 4 or 5) without increasing the routing decision delay for the header flit. Using these techniques, all tail flits can be properly pipelined and moved at the full link speed regardless of the routing delay of the header flit.

This leads to a set of interesting problems: 1) whether virtual channels can provide performance benefit for DSM systems and 2) how to allocate a set of virtual channels between the request and reply networks to obtain maximum performance benefit.

In this chapter, we study how to effectively use the virtual channel mechanism in DSM systems. We propose two design guidelines: 1) having a larger number of virtual channels in the reply network than that in the request network and 2) using an equal number of virtual channels inside the network and at the network interface. We perform application-driven simulations to evaluate various design alternatives following the above guidelines. Our results show that a network configuration consisting of one request virtual channel, two reply virtual channels, and three virtual channels per injection/consumption channel provides the best design from a cost-performance perspective. We also study the impact of several critical system parameters (such as cache line size, switch routing delay, and network topology) on the new designs. Overall, this study demonstrates that a carefully coordinated design using the virtual channel mechanism can considerably improve the performance of a DSM system.

The rest of this chapter is organized as follows. Section 8.1 estimates communication requirements and identifies the limitations in the current DSM systems when multiple virtual channels are used. Section 8.2 proposes three solutions for resolving each of the limitations. Results of our simulation evaluation on design alternatives are discussed in Section 8.3. Section 8.4 briefly reviews related work. Concluding remarks are made in Section 8.5.

8.1 Limitations of Current Approach

In this section we discuss the limiting factors in the design of current generation DSM systems.

8.1.1 Unbalanced Usage of Request and Reply Networks

Communication Behavior in DSM Systems

As discussed in Chapter 3, the communication behavior in a DSM system is affected by three factors: the cache coherence protocol, system configuration, and application. The cache coherence protocol decides the message chain for each memory block state transition. The system configuration decides the sizes of messages. The application decides the total number and the mixture of various memory block state transitions.

Regardless of the specifics of any directory-based write-invalidation coherence protocol used in a DSM system, all messages generated by the protocol can be classified into four broad categories: *authority*, *replacement*, *invalidation*, and *adjustment*. The authority category contains the basic types of messages that involve the permissions for using data. The replacement category contains the types of messages related to cache replacement. The invalidation category contains the messages occurring only in cache invalidation. The adjustment category contains the messages related to accessing semi-locked (compatible accesses to a block in wait states) or fully-locked (conflicting accesses to a block in wait states) cache lines or main memory blocks. Most known cache coherence protocols use a subset or a variation of these four categories of messages.

As described before, traffic in a DSM system has a bi-modal distribution on message size. The short messages are used for exchanging various types of control information while the long messages are used for data transfer. A short message normally contains information about the type of the message, the ID of the requester, and the block address of the target global memory, in addition to a few extra bits for network routing and fault-tolerance. A long message normally contains a complete copy of a memory block (cache line) besides all the information required for a short message. Thus, once a system configuration is fixed, the sizes of the short and long messages in the system are fixed. Let us denote the sizes of these messages as L_s and L_l, respectively.

Table 8.1 shows the characteristics of traffic generated by each of the four categories in the request and reply networks. The numbers denoted in the table are defined as the total number of messages in each category during the complete execution of an application.

	Request Network			Reply Network		
Category	Typical Message	Size	Cnt	Typical Message	Size	Cnt
Authority	shared-read-request exclusive-read-request exclusive-upgrade-request update-home-request	L_s	n_1^q	shared-data-reply exclusive-data-reply update-data-reply exclusive-grant[1]	L_l	n_1^p
Replacement				clean-replacement[1] dirty-replacement	L_l	n_2^p
Invalidation	invalidation-request	L_s	n_3^q	invalidation-ack	L_s	n_3^p
Adjustment				reject-signals wait-signals	L_s	n_4^p

Table 8.1: The characteristics of traffic generated by each of the four generic categories of cache coherence messages in the request network and the reply network.

[1]This is a short message.

Estimated Load on Request and Reply Networks

Let us estimate the traffic load (volumes) on the request and reply networks. We denote the load on the request network as V^q and the load on the reply network as V^p. Using the notations from Table 8.1, we have the following equations:

$$\begin{aligned} V^q &= n_1^q L_s + n_3^q L_s && (8.1) \\ V^p &= n_1^p L_l + n_2^p L_l + n_3^p L_s + n_4^p L_s && (8.2) \end{aligned}$$

Assuming only point-to-point messages are used in the DSM system (based on current DSM systems), the following relationships hold:

$$\begin{aligned} n_1^q &= n_1^p + n_4^p && (8.3) \\ n_3^q &= n_3^p && (8.4) \end{aligned}$$

Combining Eqns. (8.1-8.4), we have the following relationship:

$$V^q < V^p \qquad (8.5)$$

Eqn. (8.5) indicates that the traffic load on the reply network and the request network is not balanced.
With a closer look, we can observe following three useful relationships:

- The value of n_2^p depends on behavior of the application and the cache size, associativity, etc. Because the number of (dirty) replacement can not be larger than the combined number of exclusive-data-reply and exclusive-grant messages, the following relationship hold:

$$n_2^p < n_1^p \qquad (8.6)$$

- The value of n_3^p depends on the read/write ratio of the application and the application sharing pattern (the number of sharers invalidated on each write). In a full-map directory system, the following relationship holds. The reason being the number of invalidation requests/replies equals to the number of sharers at any given moment, which can not be larger than the number of shared-data-reply messages.

$$n_3^p < n_1^p \qquad (8.7)$$

- The value of n_4^p depends on the application sharing pattern and coherence protocol. However, each reject/wait signal must corresponds to an authority request. Thus, we have:

$$n_4^p < n_1^q \qquad (8.8)$$

It is noted that Eqns. (8.6-8.8) are quite conservative. Based on the results from our DSM system simulations and the common wisdom, over the course of an application execution, it can be assumed that $n_2^p \ll n_1^p$, $n_3^p \ll n_1^p$, and $n_4^p \ll n_1^q \approx n_1^p$. Combining these assumptions and Eqns. (8.1-8.4), we have the following much simplified approximation:

$$V^q/V^p \approx L_s/L_l \qquad (8.9)$$

Eqn. (8.9) indicates that the traffic load on the reply network is much heavier than that on the request network over the whole execution course of an application in a DSM system — our target state of operation for the system.

Latencies in Request and Reply Networks

The previous discussions on the load imbalance on request and reply networks show that the bandwidth requirements of the two networks are quite different. Assuming uniform traffic and equal resource being provided in both networks, the average message latencies in these two networks can be illustrated as shown in Fig. 8.1 according to the theoretical performance model for networks [44]. It indicates that the reply network typically operates at a higher load due to heavier traffic and leads to increased latency for reply messages.

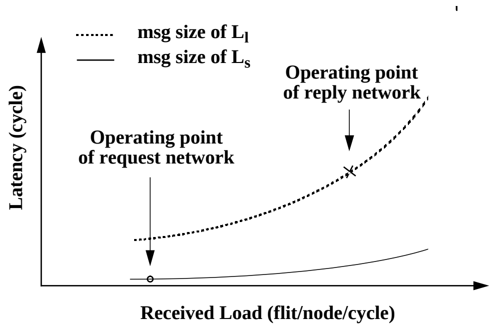


Figure 8.1: Comparing the theoretical latencies of messages in the request and reply networks by assuming these two networks being physically separated.

A fundamental characteristic of DSM systems is the cause-effect relationship between the request and reply messages. The latency of a round trip (i.e., a request message followed by the associated reply message) is critical for system performance. Current generation implementations of DSM systems provide equal resources to both networks, leading to low latency for request messages but high latency for reply messages. Such implementations often deliver sub-optimal system performance.

It is noted that the interaction between messages in the two virtual networks when they share a single physical network has not been accounted in Fig. 8.1. Nevertheless, the above discussion indicates that allocating equal resources such as virtual channels between the request and reply networks, as all current DSM systems do, is not a good design strategy.

8.1.2 Bottleneck at the Network Interface

Now, let us focus on the limitation of current generation NIs. Communication in a network with multiple virtual channels using the conventional NI is done in the following way. First, at the sending side, a message is constructed within the sending buffers at the NI. Second, the (physical) injection channel at the NI gets reserved, and then a worm associated with the message starts moving into the network in a flit by flit fashion. As the header flit of the worm moves through each router towards its destination, a virtual channel gets reserved at each hop. Once the header flit reaches the NI of the destination, the (physical) consumption channel and a receiving buffer gets reserved. Finally, all the tail flits move towards the destination in a pipelined fashion.

During the entire process, contention can occur whenever the injection channel, the consumption channel, or any virtual channel inside the network is not available. Bubbles are inserted into the worm when some of the reserved virtual channels share the underlying physical channels with other worms in the network. All these factors contribute to an increase of the time interval that an injection/consumption channel must be reserved for transferring a message. Such an increase of the time interval to transfer a message forces an injection/consumption channel to remain idle for a longer period of time. Thus, the usage of virtual channels in the network may reduce contention inside the network. But it aggravates contention at the NI. An example of such contention is illustrated in Fig. 8.2. Such contention at the NI has also been identified in the context of distributed memory systems in a separate research [16].

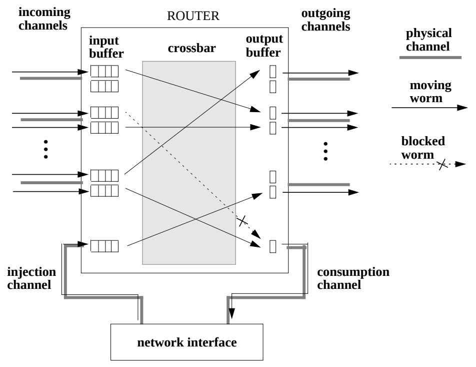


Figure 8.2: An example snapshot of a typical router and the network interface supporting one physical injection/consumption channel. There are three concurrently bypassing worms in the router. One worm is being injected from the NI to the router into the network. Two worms are terminating at this router.

8.2 Proposed Solutions

In this section, we propose a set of solutions which remove the limitations discussed in the previous section. These solutions lead to better designs for DSM systems. In the following section, we evaluate the new designs through simulation and establish guidelines for designing better systems.

8.2.1 Unbalanced Partitioning of Virtual Channels

As mentioned earlier, the contention in one virtual network (say the reply network) can affect the traffic in the other (the request network) because a single physical network is shared. In Section 8.1, we observed that the load in the reply network is considerably higher than that in the request network. Thus, to obtain better performance, our basic idea is to reduce the average message latency in the reply network significantly with little or no increase in the average message latency in the request network. Intuitively, more resources are needed in the reply network than the request network due to the higher volume of traffic in the reply network. We propose to accomplish this by using more virtual channels in the reply network. This provides the DSM system of having:

1) more reply messages (worms) in the system at any given time and 2) more opportunities for a reply worm to move around when contention occurs. Given a set of virtual channels per physical channel, an optimal unbalanced partitioning of the channels between the request and reply networks is difficult to derive using analytical modeling because it depends on too many factors. We will evaluate such interesting design alternatives using simulation in Section 8.3.

8.2.2 Supporting Virtual Injection/Consumption Channels at the NI

We propose to alleviate the bottleneck at the NI by supporting the virtual channel mechanism over the physical injection/consumption channel. We call these new virtual channels as *virtual injection/consumption channels*. At the router side, the implementation requires that the port used by injection/consumption channel be designed in a way similar to any ports in the router. At the NI side, the implementation can be done with little additional cost. Since there are only two different sizes of messages (both relatively small), the total storage required for multiple simultaneous sending/receiving buffers is quite affordable. Simple control logic can be devised to operate the sending/receiving buffer as a set of small parallel units instead of a large strict FIFO. With such virtual channels at the NI, a worm is allowed to use the physical injection/consumption channel to transfer the flits from/into a different sending/receiving buffer when its preceding messages have not yet been entirely injected/consumed. Therefore, more than one worm can be injected/consumed at the NI at the same time in an interleaved manner.

8.3 Simulation Experiments and Results

In this section, we present detailed simulation results comparing the performance of DSM system using various network and network interface configurations. These configurations are based on our solutions proposed in the previous section.

8.3.1 Simulation Experiments

This section describes our basic methodology and the default system and application parameters used in simulation experiments.

Simulation Environment

The hardware cache coherent multiprocessor we simulated has an architecture similar to the FLASH machine [85]. The system consists of 64 processing nodes interconnected in a topology of K-ary N-cube (2D mesh as default except when we study the impact of topology). Each node contains one processor. The processor is assumed to be a 300 MHz single-issue microprocessor with a perfect instruction cache and a 128 KB 2-way set associative data cache with a line size of 128 bytes. The cache is assumed to operate in dual-port mode using write-back and write-allocate policies. The instruction latencies, issue rules, and memory interface are modeled based on the DLX design [60]. The memory bus is assumed to be 8 bytes wide. On a memory block access, the first word of the block is returned in 30 processor cycles (100 ns). The successive words in the block follow in a pipelined fashion. There are 8 coalescing transaction buffers at each node used for resolving outstanding remote memory requests. Each buffer can hold one cache line and merge multiple writes into the line. The machine is assumed to use a non-FIFO coherence protocol similar to the one in the SGI Origin [88] and supporting the release consistency memory model [85, 88]. The synchronization protocol assumed in the system is the QOLB protocol, which is similar to the one used in DASH [91]. The node simulator models the internal structures and the queuing and contention at the node controller, the main memory, and the cache [110]. Table 8.2 shows the default memory hierarchy parameters, node controller occupancy delays, network parameters, and network interface parameters used in our simulations.

Memory Hierarchy Parameters		Network Parameters	
Processor frequency	300MHz	Network frequency	200MHz
Cache access	1 cycle	Channel width / Flit size	2 bytes
Cache line size (L)	128 bytes	Link Propagation	1 net cycle
Cache set associativity	2	Router switch delay	1 net cycle
Cache size per node	128 Kbytes	Routing delay	3 net cycles
Memory word width (W)	8 bytes/cycle	Physical network	1
Memory response delay	30 cycles	Virtual networks	2
Cache fill time	30+L/W cycles	Topology	2D
Memory access time	30+L/W cycles		
Size per trans. buffer	L bytes		
No. of trans. buffers	8		
Node Controller Occupancy		**Network Interface Parameters**	
Directory check	7 cycles	Outgoing message startup	15 cycles
Directory check&update	14 cycles	Incoming message dispatch	8 cycles
Each invalidation	12 cycles	Control message size	6 bytes
Message forward	3 cycles	Data message size	6+L bytes

Table 8.2: Default system parameters used in the simulation.

We used four applications — FFT (64K points), Radix (1M keys, 1K radix, 1M max), Barnes (8K bodies, 4 steps), and LU (512×512, 8×8 blocks) — in our simulations.

Network Configurations Evaluated

From the earlier discussion, it is clear that there are many possible network configurations with a given number of virtual channels. For example, consider a DSM system which supports four virtual channels. Three different partitions of the virtual channels are possible. A (3,1) partition has three lanes in the request network and one lane in the reply network. The other two partitions can be denoted as (2,2) and (1,3). Now, the question is which partitioning scheme provides the best performance? Due to the complicated interference between the two virtual networks, the answer to this question is non-trivial. Additionally, we have the flexibility of using virtual injection/consumption channels at the NI. Since the number of lanes per physical injection and consumption channel are independent of each other and independent of the number of virtual channels inside the network, it leads to a large number of network configurations. To keep this simulation study under control, we evaluated the networks with moderate number (up to 5) of virtual channels.

For convenience of discussion, let us denote a network configuration by a tuple (q, p, i, c), where q, p, i, c denote the number of virtual lanes in the request network, reply network, per injection channel, and per consumption channel, respectively. When there is only one physical injection or consumption channel, $i = 1$ or $c = 1$ is used.

Performance Metrics

We present all our simulation results in two sets: the execution time and the network latency. This helps us to correlate the behaviors of the underlying network with the overall DSM system performance and to gain important insights into the network design issues. The overall execution time is broken down into four components: the CPU computation time (C), the memory read waiting time (R), the memory write waiting time (W), and the synchronization waiting time (S). All the times we present are normalized to that of the left most system configuration for each application except when stated otherwise. The network latencies for two types (request and reply) of messages are presented in absolute time (microseconds). The presented latencies are the average of the measured raw network latencies globally clocked from the start of the injection of the first flit at the sender till the completion of the consumption of the last flit at the receiver across all the messages of the same type.

8.3.2 Simulation Results

In this section, we first evaluate the performance benefit of virtual injection/consumption channels. Next, we compare the performance of various partitioning of virtual channels in the network. Finally, we examine the impact of three critical system parameters on the overall performance.

Effect of Virtual Injection and Consumption Channels

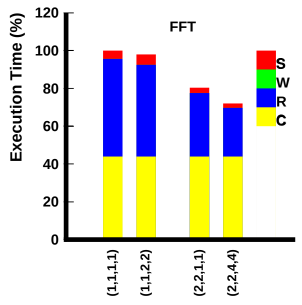
We first studied the effect of virtual injection channels and virtual consumption channels on the overall execution time of each benchmark application. The four configurations used in the experiments were: (1,1,1,1), (1,1,2,2), (2,2,1,1), and (2,2,4,4). The results are shown in Fig. 8.3.

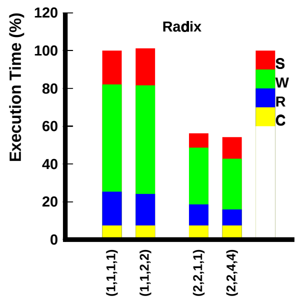
It can be observed that the overall execution time was reduced in all but one case when virtual injection/consumption channels were used compared to those configurations when they were not used. As the number of virtual channels increases and the bandwidth of the underlying physical channel remains constant, the overall system performance improves for all applications. The actual reduction in execution time varies across applications, with a maximum of 10.4% seen in FFT from configuration (2,2,1,1) to configuration (2,2,4,4).

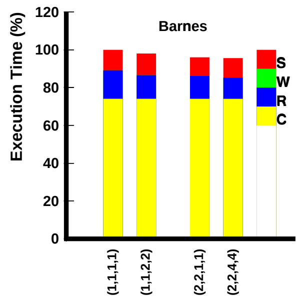
One exception occurred in Radix when the configuration was changed from (1,1,1,1) to (1,1,2,2). Instead of decreasing, the overall execution time increased by 1.1%. The reason for this abnormality could be due to a change in the critical execution path of the application.

From the timing breakdowns, it can be observed that the CPU computation time remained almost constant across all network configurations in every application. This is expected for the following reason. The amount of computation per processing node is largely fixed for a given input data set and system size for these applications. Improving network performance helps to cut down various waiting times but not the CPU computation times.

To gain more insights into the direct effect of virtual injection/consumption channels on the network performance, let us examine the changes in the average message latencies in the request and reply networks. Figure 8.4 shows the results from the same set of experiments. It can be observed that the average message latency of request/reply network is usually smaller in systems which used virtual injection/consumption channels compared to that in systems which did not use them. This trend is more prominent for systems with two virtual channels in each network. This indicates that an incoming/outgoing message is more often blocked by another incoming/outgoing message at the NI in a system which supports virtual channels inside the network but not at the NI.







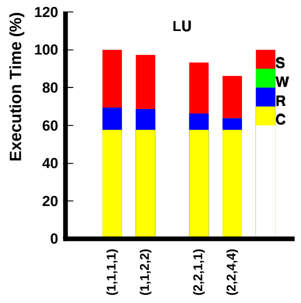
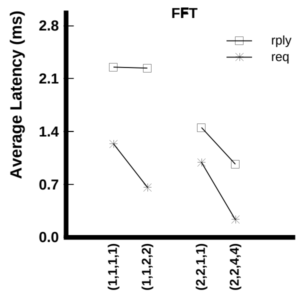
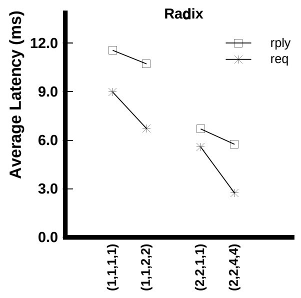
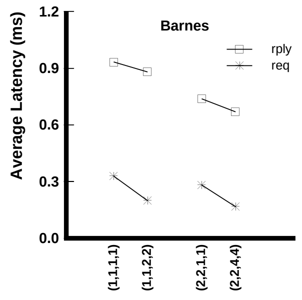


Figure 8.3: The effect of virtual injection and virtual consumption channels on the overall execution time of benchmark applications.







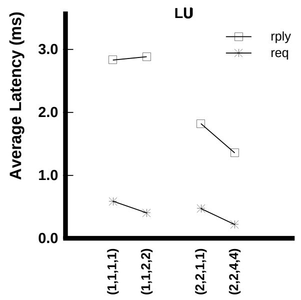


Figure 8.4: The effect of virtual injection and virtual consumption channels on the average message latencies in the request and reply networks.

Based on the above results, in the following subsections, we focus on network configurations with the number of virtual injection/consumption channels at the NI being equal to the total number of virtual channels in the network.

Effect of Unbalanced Virtual Channel Partitioning

Next, we studied the effect of virtual channel partitioning between the request and reply networks. We performed evaluations on network configurations with no more than five virtual channels. There were nine different network configurations: (2,1,3,3) and (1,2,3,3) with 3 virtual channels; (3,1,4,4), (2,2,4,4), and (1,3,4,4) with 4 virtual channels; and (4,1,5,5), (3,2,5,5), (2,3,5,5), and (1,4,5,5) with 5 virtual channels. These configurations were compared with the (1,1,2,2) base configuration. The results of the execution times are shown in Fig. 8.5.

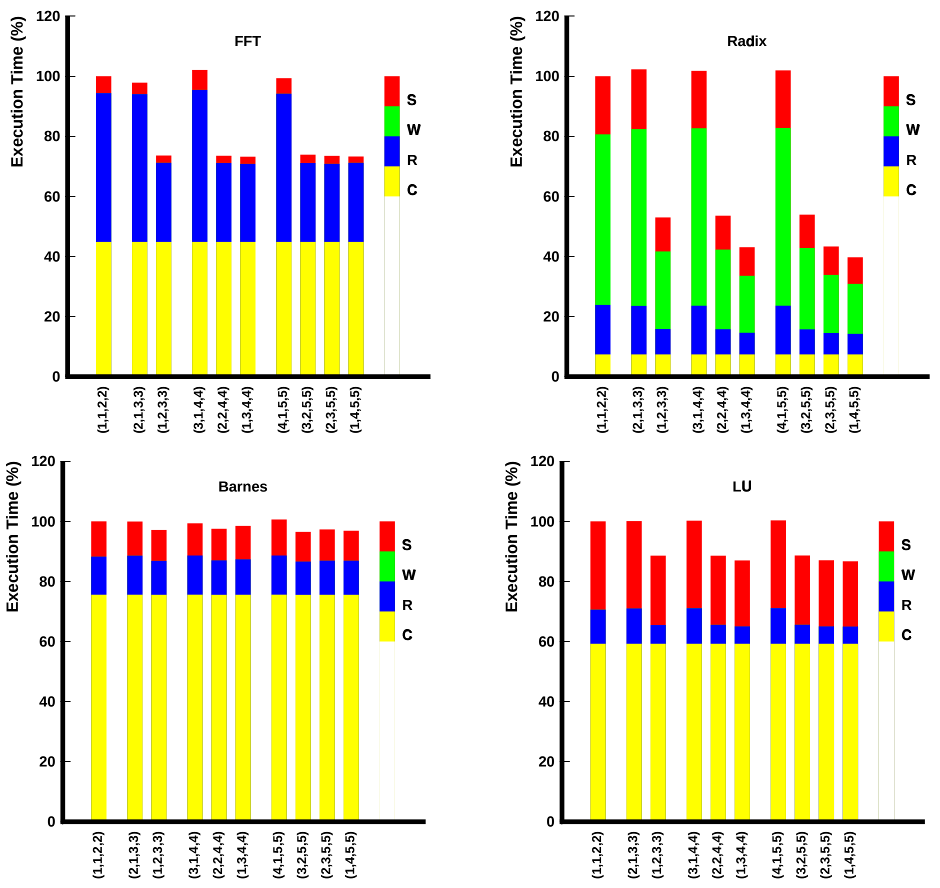


Figure 8.5: The effect of virtual channel partitioning on the overall execution time.

As expected, for all the applications, the overall execution times were reduced for those configurations which used two or more virtual lanes in the reply network. Specifically, configurations with more virtual channels in the reply network, like (1,2,3,3), (1,3,4,4), and (1,4,5,5), provided the best or close to the best performance in their respective group. The maximum reduction in the execution time of each application ranged from 3.2% in Barnes to 60.3% in Radix. The main reason behind the small performance improvement in Barnes is that the CPU computation time is very dominant (close to 80% in our base configuration) in the overall execution time. Similarly, the high performance improvement in Radix can be contributed to the dominant memory access waiting time (about 70% in the base configuration). Therefore, the maximum performance improvements of 26.8% and 13.4% in FFT (45% computation time) and LU (60% computation time) respectively are more representative.

The results also show that configuration (1,2,3,3) is an interesting design alternative from cost-performance point of view. It gains close to the maximum performance improvement by using just one more virtual channel in the reply network than the minimum required in DSM systems.

Figure 8.6 shows the average message latencies in the request and reply networks from the same set of experiments. It can be observed that the average message latency in the request network decreased initially and flattened out (or increased slightly) as more virtual channels were moved from the request network to the reply network within each group. On the other hand, the average message latency in the reply network kept decreasing towards a certain value as it obtained more virtual channels.

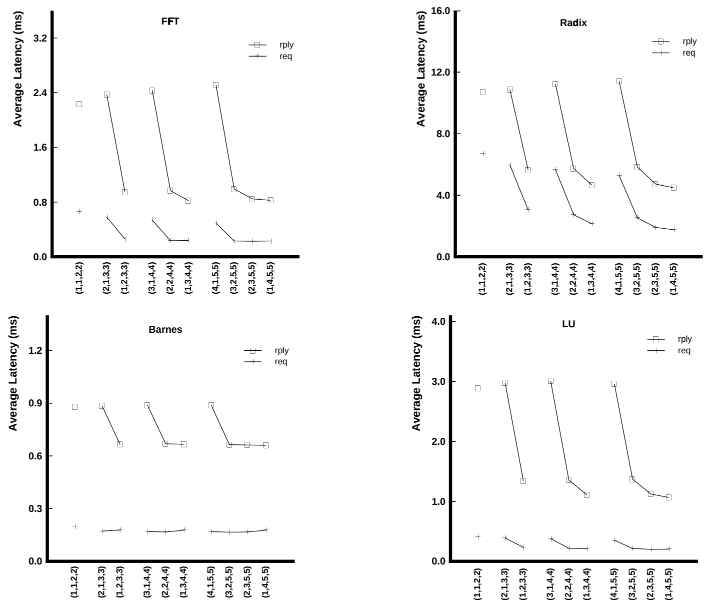


Figure 8.6: The effect of virtual channel partitioning on the average message latencies in the request and reply networks.

Another observation is that the knee points of all the curves appear at the positions when there are two virtual lanes in the reply network. It correlates to our finding that configuration (1,2,3,3) is the most promising alternative from a cost-performance perspective.

In the next three subsections, we examine how and to what extent the performance benefit of multiple virtual channels change when varying (one at a time) the values of three critical system parameters.

Impact of Cache Line Size

First, we studied the impact of cache line size on the benefit of virtual channels. Modern DSM systems tend to use large cache line size (typical 128 or 256 Bytes) to save the cost of main memory directories and cache tags and exploit the spatial locality of memory references. We selected four configurations (1,1,2,2), (1,2,3,3), (1,3,4,4), and (1,4,5,5) and performed simulations by varying cache line size from 64 to 128 and to 256 bytes. Figures 8.7 and 8.8 show the results of the application execution times and the average message latencies. All the execution times are normalized with respect to the configuration (1,1,2,2) with a line size of 128 bytes (the left most bar in the middle group) for each application. As expected, more virtual channels in the reply network are found to be beneficial. This trend becomes stronger as the cache line size grows. Specifically, maximum performance improvement increases from 23.4% to 32.7% in FFT, from 51.7% to 62.3% in Radix, from 1.6% to 9.4% in Barnes, from 13.3% (with a line of 128 bytes) to 18.6% in LU. According to a study done in [35], the execution time of a DSM application shows a bathtub behavior as the cache line size changes with a pollution point around 64 bytes. Thus, it is reasonable to expect similar or larger performance gain when some other cache line size is used in a real system.

It can again be observed that the configuration (1,2,3,3) remains attractive for a wide range of cache line sizes. The difference between the performance of this configuration and that of the optimal configuration using more number of virtual channels grows slowly as the cache line size increases. This is shown quite clearly from the changes in the sharpness with which the curves turn at the knee points in Fig. 8.8.

Impact of Routing Delay

Next, we studied the impact of routing decision delay on the benefit of virtual channels. Some researchers believe that the network speed can not remain unchanged as more number of virtual channels are supported. In an earlier study [9], a performance model of 30% slowdown in network cycle time for adding each virtual channel has been proposed. However, a more careful analysis on the problem reveals that the slowdown is mainly caused by the routing delay of the header flit at a router. As the network technology advances and better pipelining techniques are developed, several commercial routers [49, 124] supporting a moderate number of virtual channels have been designed successfully without noticeably increasing the network cycle time and routing delay. However, for an in-depth investigation, we selected two configurations (1,3,4,4) and (1,4,5,5) and performed simulations by varying routing delay from 3 network cycles to 4 network cycles (33% slowdown) and using the same network cycle time. We did not select (1,2,3,3) configuration because there are good reasons to believe that no noticeable increase in routing delay is needed to increase the number of virtual channels from 2 to 3. Figures 8.9 and 8.10 show the results of the application execution times and the average message latencies, respectively. It can be observed that such an increase in the routing delay causes marginally increase in the overall execution time (less than 2.8% in all our experiments). This indicates that the performance of DSM systems is not so sensitive to reasonably increased routing delay. However, the performance is quite sensitive to the overall network cycle time (or link cycle time) increase as reported in [32, 35, 142].

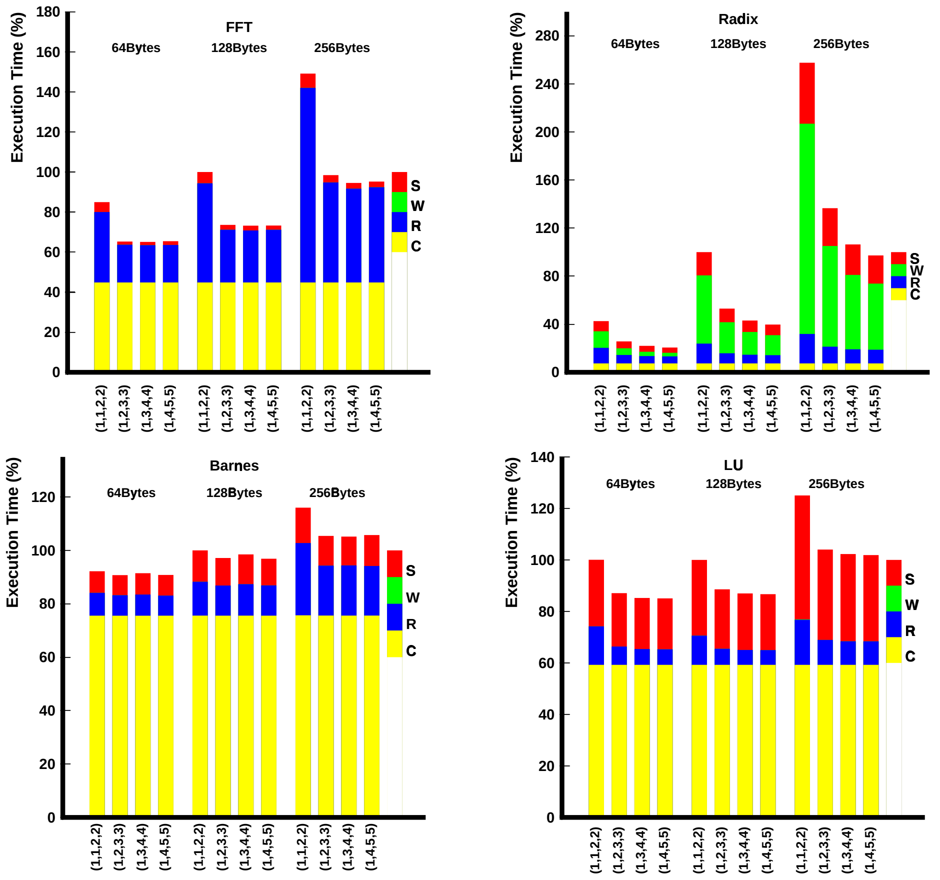


Figure 8.7: Changing trend of the effect of virtual channels on the overall execution times under different cache line sizes.

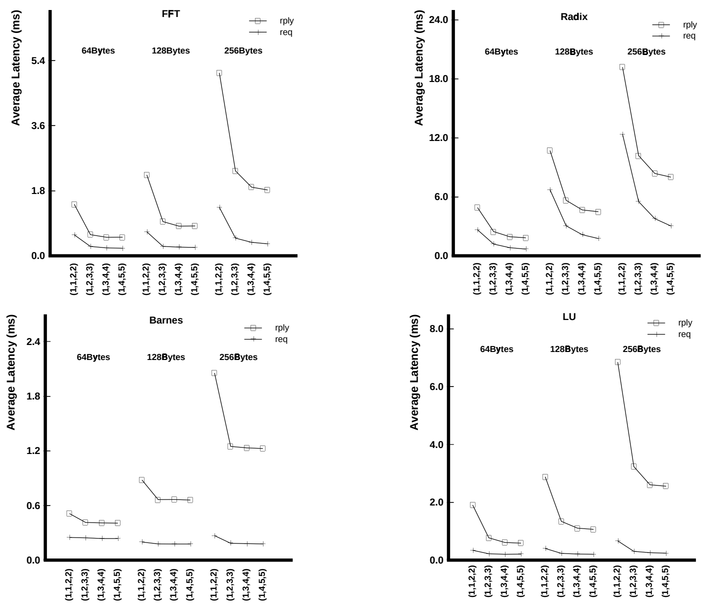


Figure 8.8: Changing trend of the effect of virtual channels on the average message latencies under different cache line sizes.

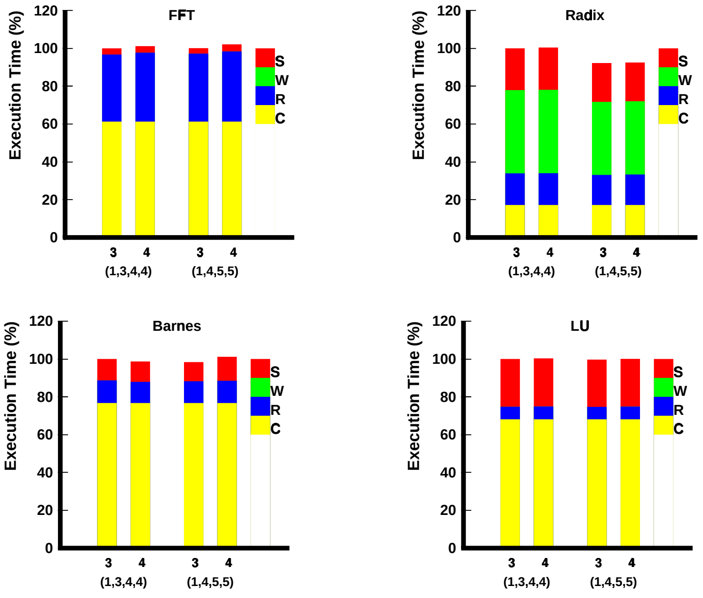


Figure 8.9: Impact of switch routing delay on the overall execution times.

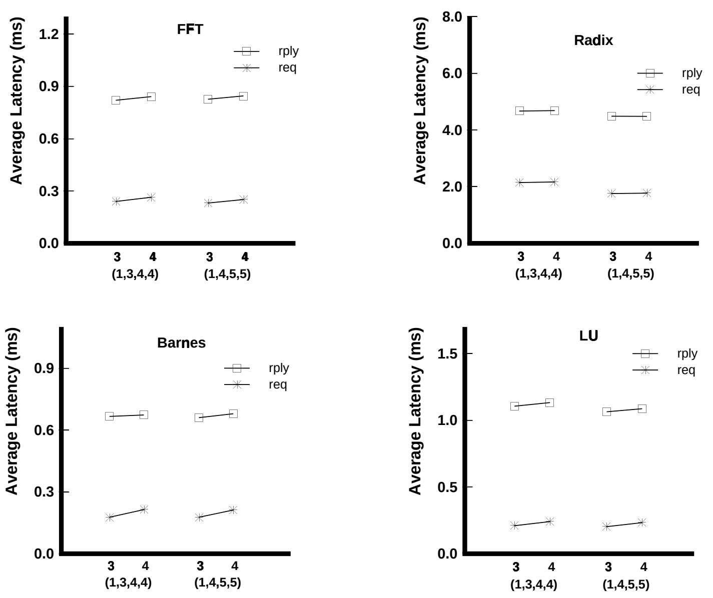


Figure 8.10: Impact of switch routing delay on the average message latencies.

Impact of Topology

Finally, we studied the impact of network topology. We focused on the k-ary n-cube topologies. Recently, there are some renewed interests of using higher dimensional k-ary n-cube topologies in DSM systems because of many different reasons [45, 88]. Thus, we selected three configurations (1,1,2,2), (1,2,3,3), and (1,3,4,4) and performed simulations by varying dimensionality from 2 to 3 to 6 and keeping the total number of processing nodes fixed at 64. Figures 8.11 and 8.12 show the results of the application execution times and the average message latencies, respectively. It can be observed that the performance benefit of using virtual channels reduces as the dimensionality increases. Specifically, the performance improvement decreases from 26.8% to 3.5% in FFT, from 57.0% to 32.8% in Radix, from 2.9% to no gain in Barnes, and from 13.1% to 4.2% in LU. This is not surprising because the network bisection bandwidth increases as the dimensionality increases (under constant link width). The actual physical bisection bandwidth increases from 6.4GB/s to 12.8GB/s to 25.6GB/s. Thus, less contention occurs in the network and virtual channels become less useful. This indicates that the benefit of contention reduction obtained by using more virtual channels can also be obtained by using more physical channels at a higher cost. This is confirmed in Fig. 8.12 which shows that the reductions in average message latencies decrease significantly from 2D to 3D to 6D in all the applications.

8.4 Related Work

Using multiple virtual channels with various buffer sizes in DSM systems was first studied in [84]. However, with more realistic assumptions on the existing technologies, the study in [142] has suggested that it may be unjustified to use virtual channel mechanism in DSM systems with

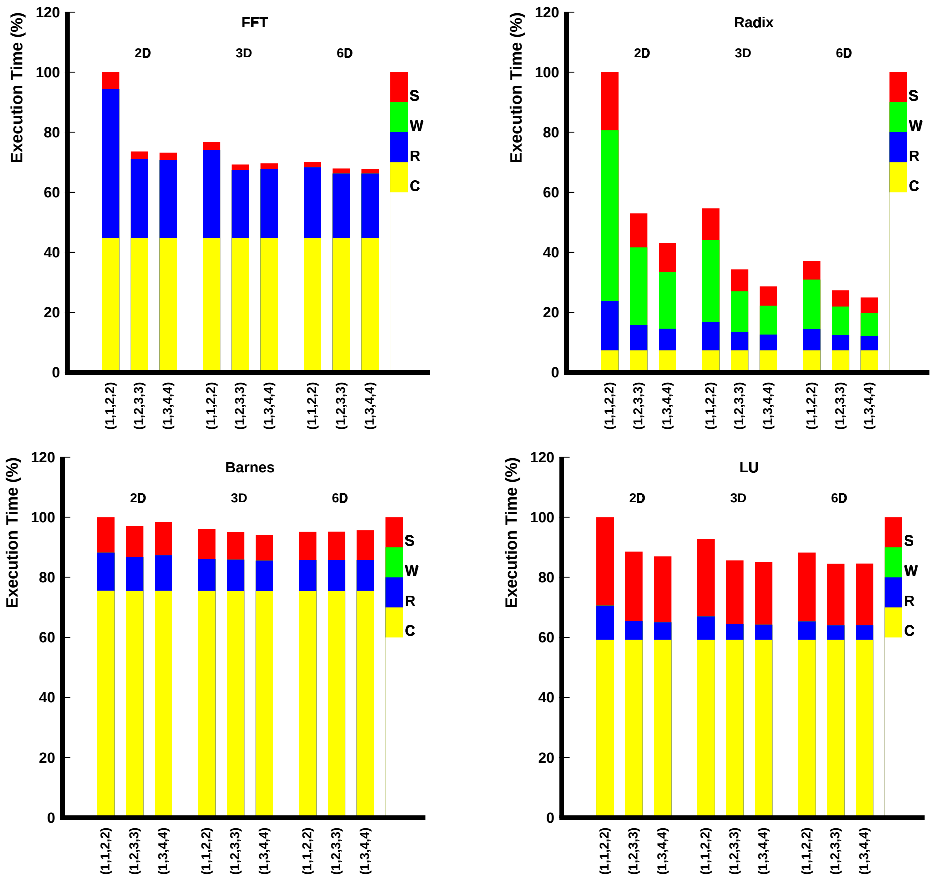


Figure 8.11: Impact of network topology on the overall execution times.

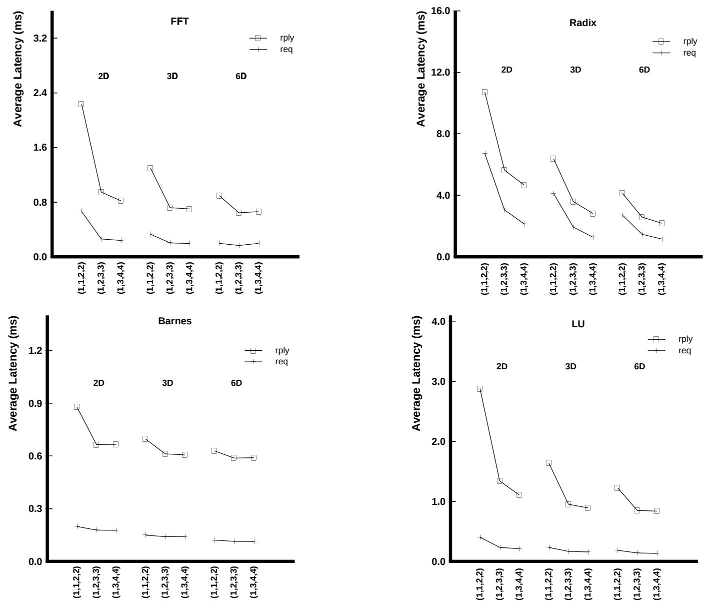


Figure 8.12: Impact of network topology on the average message latencies.

respect to cost-performance viewpoint. But, as we have shown in this study, incorporating virtual channel mechanism into DSM systems in an efficient manner can provide a significant performance improvement. The performance impact of network contention under various designs of cache and memory module, processor speed, and different network components has been reported in [35]. A set of guidelines have been proposed in [32] for designing better networks for DSM systems with two virtual networks (with one virtual channel each) and one injection and consumption channel at the network interface. Impact of multiple physical injection/consumption channels has been the focus of study in [16]. However, this study has been done with respect to uniform traffic distribution in distributed memory systems. Traffic non-uniformities in the forward and reverse multistage networks in the Cedar system have been investigated in [139]. Strategies for improving the performance of dance-hall vector multiprocessors using MIN networks have been proposed in this study.

8.5 Summary

In this chapter we have presented two simple techniques for exploiting the promising performance benefit of the virtual channel mechanism in distributed shared memory systems. Virtual injection and consumption channels can be implemented at very little additional cost to the existing designs of switches and network interfaces. Unbalanced virtual channel partitioning between the request and reply networks requires some design changes within the routers without any increase in cost. The effects of the virtual channel mechanism with both techniques have been evaluated through simulations using a set of representative benchmark applications.

Our results indicate that a network configuration consisting of one request virtual lane, two reply virtual lanes, three virtual lanes per injection/consumption channel is the most cost-effective design alternative. Overall, we have shown that the virtual channel mechanism can reduce the execution time of a DSM application significantly. In addition, we have shown that the performance benefit of using virtual channels increases with a larger cache line size, is not so sensitive to changes in routing delay, and decreases in a higher dimensional k-ary n-cube topology under constant physical link bandwidth constraint. This study demonstrates that the virtual channel mechanism can significantly improve the performance of a distributed shared memory system.

CHAPTER 9

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

In this chapter we summarize the contributions of this thesis, and suggest some directions for future research.

9.1 Summary of Research Contributions

This thesis makes the following major contributions for designing efficient communication subsystems for distributed shared memory (DSM) systems.

1. **Network Contention Analysis:** Various types of network contention at the network interface and inside the network in distributed shared memory systems are identified. A framework is presented for modeling, isolating, and analyzing the impact of these types of network contention on the performance of DSM systems. It is shown that network contention, especially contention inside network, can affect the performance of DSM systems significantly. The analysis also provides new insights into the network behavior in DSM systems.

2. **Analytical Performance Modeling:** A parameterized model is developed to estimate the performance of DSM systems. The model captures the fundamental characteristics of basic system components—the program, processor, cache, coherence scheme, memory, and network. It characterizes the complex interactions between system design parameters via a set of linear equations. Using this model, it is shown that architects can easily estimate the performance of a system and identify its bottlenecks. This model is also used to derive a set of guidelines for selecting network design parameters for future DSM systems.

3. **Designing Pipelined Node-Network Interfaces:** The overhead incurred at the node-network interface in DSM systems is an important component of remote memory access latency. A new design of pipelined node-network interface is proposed for supporting cut-through delivery and partial cache-filling, eliminating store-and-forward operations at various stages in the data reply phase. This design demonstrates significant reduction in the restart and back-to-back memory latencies compared to the conventional block-based interface.

4. **Incorporating Multiple-path Network:** Multiple network paths between a pair of nodes exist in current generation DSM systems. They are helpful in increasing network bandwidth and reducing network contention, leading to lower network latency. A novel, block correlated FIFO channel (C-FIFO) design strategy and its implementation using current technology is proposed for incorporating multiple-path networks in DSM systems. Evaluations demonstrate that this strategy and the associated implementation provide a significant cost-performance advantage over the existing design strategies and implementations.

5. **Reducing Invalidation Overhead:** The problem of high protocol processing overhead and network congestion caused by cache invalidations is analyzed in depth. A number of multidestination-based mechanisms which utilize hardware support at the router are proposed for reducing such overhead and congestion. These mechanisms can be applied to DSM systems using full-map cache coherence schemes and to systems using limited directory schemes. It is shown that with moderate hardware support at the router, the invalidation problem can be alleviated substantially.

6. **Designing Unbalanced Network:** The inherent characteristics of request and reply traffic are analyzed and traced back to the working principles of the DSM systems. To better accommodate the two different types of traffic, unbalanced network designs are proposed. Specifically, more virtual channels are shown to be suitable for the reply traffic which demands much higher network bandwidth than the request traffic. It is clearly shown that such designs can improve the overall system performance significantly compared to the conventional balanced network designs.

Overall, the contributions of this thesis have been to show that the performance of DSM systems can be enhanced significantly with moderate modifications to existing communication subsystem architectures. The performance benefits of these modifications have been established through theoretical analysis and extensive simulation-based experiments. The designs and enhancements developed in this thesis therefore hold significant promise for being implemented in current and future generation DSM systems.

9.2 Suggestions for Future Research

The topic addressed by this thesis is a fertile area for further research with immediate impact in the computer industry. Almost all major manufacturers of high performance computing systems either have or have announced their own product lines of DSM systems or some kind of hybrid systems, such as clustering symmetric multiprocessor (SMP) systems. Many interesting and practical problems in this area still wait for solutions. In this section, we describe some of the problems and provide a few suggestions.

- **Commercial Application Based System Evaluation:** Enterprise computing like on-line transaction processing, decision supporting, Internet-based web services, etc., is a major market segment for DSM systems. It is important to evaluate the impact of any architectural innovation on commercial applications [140]. Technically, porting commercial applications to our DSM testbed [110] is not much different from porting the scientific computing benchmarks such as SPLASH/SPLASH2 [127, 126]. However, two main obstacles prevent us from doing so. First, the source codes of such applications are mostly proprietary and hard to obtain. Even characteristics of such workload are commercial secrets in the industry. Second, most such applications require enormous amount of computing power and storage, making execution-driven simulation not feasible. Most recently, some research [140, 86] has focused on commercial application based evaluation using TPC-C and TPC-D benchmark suite. An interesting observation from these studies is that commercial applications tend to have more communication and less data locality. Therefore, the hardware designs and enhancements proposed in this thesis are expected to improve system performance even more. However, by including commercial applications as part of our evaluation benchmark suites, new subtle characteristics may be found, leading to design better architectural support for DSM systems.

- **Efficient Network Topologies for DSM Systems:** In this thesis, we largely assume a k-ary n-cube network. However, an untouched and interesting question is which network topologies such as k-ary n-cube, multi-stage interconnection (MIN), rings, or irregular network are efficient for DSM systems. It is a common belief that efficient topologies must have low latency and high communication bandwidth. To achieve such objectives, the topologies must have a small diameter, short average distance between nodes, and multiple distinct paths between nodes. It is also interesting to investigate the performance benefits of various collective communication mechanisms proposed for these network topologies [74, 76, 75, 129, 131, 135, 130, 132, 128]. Other important and practical considerations such as reliability, availability, serviceability, reconfigurability, etc., can further complicate the solution. The results of such a study will help identify ideal topologies with a given number of nodes, switches, and cables.

- **Efficient Network Routing Schemes for DSM Systems:** A topic closely related to the study on efficient topologies is to develop efficient routing schemes for the resultant topologies. In general, deadlock-free routing schemes [26, 43, 54, 55] and deadlock recovery schemes [8] have been well studied and some of them have been shown to provide good performance in various networks under synthetic traffic. However, very little research has been done in the context of DSM systems due to the complexity in the nature of the problem. Some preliminary results of our most recent work in this direction has been reported in [125].

- **Clustered Hybrid Systems:** A recent trend in high-performance computing industry is to build cost-effective clustered systems. Such a system consists of a hierarchical organization with a small cluster of processors forming the basic building block. The Stanford DASH [91] and the Cray T3D [77] are earlier examples. Many latest versions of the IBM SP use SMP nodes, each node consisting of 2 to 8 processors. Often the processors in such systems are interconnected by two or more levels of networks with different topologies, for example, a bus or crossbar for intra-cluster, and a MIN or k-ary n-cube for inter-cluster connection. Preliminary studies [13, 12, 109, 14, 17, 11, 65, 66] have been performed on the issues involving unicast/multicast communication primitives in clustered systems [15]. Interestingly, most clustered systems support a programming paradigm with a somewhat coherent global memory space. Therefore, how to extend the mechanisms proposed for efficient communication in this thesis to such systems is worth further exploration.

- **Supporting Coherent Memory Space Atop VIA:** Recently, virtual interface architecture (VIA) has been proposed as an industry standard for low latency and high bandwidth communication between computers in system area networks (SAN). The Fast Messages communication library [107, 106] and U-net communication messaging layer [143, 145] are two software packages with architectures very similar to the VIA. It is very possible that VIA based hardware and software libraries will be the industrial standard for internal communication in high-performance systems. How to cost-effectively support a somewhat coherent global memory space on top of VIA is a new and exciting area for future study with potentially a strong impact on industry.

BIBLIOGRAPHY

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *International Symposium on Computer Architecture*, pages 2–13, 1995.

[2] A. Agarwal and M. Cherian. Memory System Characterization of Commercial Workloads. In *International Symposium on Computer Architecture*, pages 96–406, 1989.

[3] A. Agarwal, R. Simoni, J. Hennesy, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Int'l Symposium on Computer Architecture*, 1988.

[4] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadle, D. M. Dias, and M. Snir. SP2 System Architecture. *IBM Systems Journal*, 34(2):263–272, 1995.

[5] C. Amza, A. L. Cox, et al. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[6] T. Anderson, D. Culler, and Dave Patterson. A Case for Networks of Workstations (NOW). *IEEE Micro*, pages 54–64, Feb 1995.

[7] B. S. Ang, D. Chiou, L. Rudolph, and Arvind. Message Passing Support on StarT-Voyager. Technical Report Computation Structures Group Memo 387, Massachusetts Institute of Technology, July 1996.

[8] K. V. Anjan and T. M. Pinkston. An Efficient Fully Adaptive Deadlock Recovery Scheme: DISHA. In *International Symposium on Computer Architecture*, pages 201–210, 1995.

[9] K. Aoyama and A. A. Chien. The Cost of Adaptivity and Virtual Lanes in a Wormhole Router. *Journal of VLSI Design*, 2(4):315–333, 1995.

[10] Y. Aydogan, C. B. Stunkel, C. Aykanat, and B. Abali. Adaptive Source Routing in Multistage Interconnection Networks. In *Proceedings of the International Parallel Processing Symposium*, pages 258–267, 1996.

[11] D. Basak. *Designing High Performance Parallel Systems: A Processor-Cluster Based Approach.* PhD thesis, The Ohio State University, 1996.

[12] D. Basak and D. K. Panda. Scalable Architectures with k-ary n-cube cluster-c Organization. In *Symposium on Parallel and Distributed Processing*, pages 780–787, 1993.

[13] D. Basak and D. K. Panda. Designing Large Hierarchical Multiprocessor Systems under Processor, Interconnection, and Packaging Advancements. In *International Conference on Parallel Processing*, pages I:63–66, 1994.

[14] D. Basak and D. K. Panda. Designing Clustered Multiprocessor Systems under Packaging and Technological Advancements. *IEEE Transactions on Parallel and Distributed Systems*, pages 962–978, Sept 1996.

[15] D. Basak and D. K. Panda. Designing Processor-cluster Based Systems: Interplay Between Cluster Organizations and Collective Communication Algorithms. In *Proceedings of the International Conference on Parallel Processing*, pages 271–274, 1996.

[16] D. Basak and D. K. Panda. Alleviating Consumption Channel Bottleneck in Wormhole-Routed k-ary n-cube Systems. *IEEE Transactions on Parallel and Distributed Systems*, 9, 1998.

[17] D. Basak, D. K. Panda, and M. Banikazemi. Benefits of Processor Clustering in Designing Large Parallel Systems: When and How? In *International Parallel Processing Symposium*, Chicago, IL, Aug 1996.

[18] J. Beecroft, M. Homewood, and M. McLaren. Meiko CS-2 Interconnect Elan-Elite Design. *Parallel Computing*, 20:1627–1638, Nov 1994.

[19] N. J. Boden, D. Cohen, et al. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, pages 29–35, Feb 1995.

[20] R. V. Boppana and S. Chalasani. A Comparison of Adaptive Wormhole Routing Algorithms. In *International Symposium on Computer Architecture*, pages 351–360, 1993.

[21] R. V. Boppana, S. Chalasani, and C. S. Raghavendra. On Multicast Wormhole Routing in Multicomputer Networks. In *Symposium on Parallel and Distributed Processing*, pages 722–729, 1994.

[22] T. Brewer and G. Astfalk. The Evolution of the HP/Convex Exemplar. In *Proceedings of COMPCON Spring'97: Forty-Second IEEE Computer Society International Conference*, pages 81–86, February 1997.

[23] D. C. Burger and D. A. Wood. Accuracy vs. Performance in Parallel Simulation of Interconnection Networks. In *Proceedings of the International Symposium on Parallel Processing*, April 1995.

[24] M. Censier and P. Feautier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, 27:1112–1118, December 1978.

[25] S. Chandra, J. R. Larus, and A. Rogers. Where is time spent in message-passing and shared-memory programs? In *The sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 61–73, San Jose, CA, October 1994.

[26] A. A. Chien and J. H. Kim. Planar-Adaptive Routing: Low-Cost Adaptive Networks for Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, pages 268–277, 1992.

[27] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware-Software Approach.* Morgan Kaufmann, 1998.

[28] D. Dai and D. K. Panda. Efficient Schemes for Limited Directory-Based DSMs Using Multi-destination Message Passing. Technical Report OSU-CISRC-11/96-TR61, Dept. of Computer and Information Science, The Ohio State University, Nov 1996.

[29] D. Dai and D. K. Panda. Reducing Cache Invalidation Overheads in Wormhole DSMs Using Multidestination Message Passing. In *International Conference on Parallel Processing*, pages I:138–145, Chicago, IL, Aug 1996.

[30] D. Dai and D. K. Panda. Reducing Cache Invalidation Overheads in Wormhole Routed DSMs Using Multidestination Message Passing. Technical Report OSU-CISRC-4/96-TR24, Dept. of Computer and Information Science, The Ohio State University, Apr 1996.

[31] D. Dai and D. K. Panda. Effective Use of Virtual Channels in Wormhole Routed Distributed Shared Memory Systems. Technical Report OSU-CISRC-10/97-TR46, The Ohio State University, October 1997.

[32] D. Dai and D. K. Panda. How Can We Design Better Networks for DSM Systems? In *Proceedings of the 1997 Parallel Computer Routing and Communication Workshop (PCRCW'97), Lecture Notes in Computer Science #1417*, pages 171–184, Atlanta, GA, June 1997.

[33] D. Dai and D. K. Panda. How Can We Design Better Networks for DSM Systems? Technical Report OSU-CISRC-3/97-TR19, Dept. of Computer and Information Science, The Ohio State University, March 1997.

[34] D. Dai and D. K. Panda. How Much Does Network Contention Affect Distributed Shared Memory Performance? Technical Report OSU-CISRC-2/97-TR14, Dept. of Computer and Information Science, The Ohio State University, February 1997.

[35] D. Dai and D. K. Panda. How Much Does Network Contention Affect Distributed Shared Memory Performance? In *Proceedings of the International Conference on Parallel Processing*, pages 454–461, Chicago, IL, Aug 1997.

[36] D. Dai and D. K. Panda. Effects of Network Components in Distributed Shared Memory Systems. Technical Report OSU-CISRC-8/98-TR34, The Ohio State University, August 1998.

[37] D. Dai and D. K. Panda. Evaluating Pipelined Node-Network Interface Designs for DSM System. Technical Report OSU-CISRC-8/98-TR36, The Ohio State University, August 1998.

[38] D. Dai and D. K. Panda. Exploiting the Benefits of Multiple-Path Network in DSM Systems: Architectural Alternatives and Performance Evaluation. Technical Report OSU-CISRC-8/98-TR33, The Ohio State University, August 1998.

[39] D. Dai and D. K. Panda. Exploiting the Benefits of Multiple-Path Network in DSM Systems: Architectural Alternatives and Performance Evaluation. *IEEE Transactions on Computers*, 48(2):236–244, February 1999.

[40] W. J. Dally. Performance Analysis of k-ary n-cube Interconnection Network. *IEEE Transactions on Computers*, pages 775–785, June 1990.

[41] W. J. Dally. Virtual Channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, pages 194–205, Mar 1992.

[42] W. J. Dally and C. L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, pages 547–553, May 1987.

[43] J. Duato. A New Theory of Deadlock-Free Adaptive Routing in Wormhole Networks. *IEEE Transactions on Parallel and Distributed Systems*, 4(12):1320–1331, 1993.

[44] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach.* The IEEE Computer Society Press, 1997.

[45] Jose Duato and M. P. Malumbres. Optimal Topology for Distributed Shared Memory Multiprocessors: Hypercubes Again? In *Proceedings of the Euro-Par96*, page to appear, 1996.

[46] B. Duzett and R. Buck. An Overview of the Ncube-3 Supercomputer. In *Proceedings of the Frontiers of Massively Parallel Computation*, pages 458–464, 1992.

[47] A. T. Eiriksson and K. L. McMillan. Using formal verification/analysis methods on the critical path in system design: A case study. In *Proceedings of Computer Aided Verification Conference*, Liege Belgium, 1995. Springer Verlag.

[48] E. W. Felten, R. A. Alpert, A. Bilas, M. A. Blumrich, D. W. Clark, S. N. Damianakis, C. Dubnicki, L. Iftode, and K. Li. Early Experience with Message-Passing on the SHRIMP Multicomputer. In *International Symposium on Computer Architecture (ISCA)*, pages 296–307, 1996.

[49] M. Galles. Spider: A High-Speed Network Interconnect. *IEEE Micro*, pages 34–39, January/February 1997.

[50] D. Garcia and W. Watson. Servernet II. In *Proceedings of the 2nd Parallel Computer Routing and Communication Workshop*, pages 119–135, June 1997. Available as `Lecture Notes in Computer Science #1417, Springer Verlag`.

[51] K. Gharachorloo. Towards More Flexible Architectures. In J. L. C. Sanz, editor, *Opportunities and Constraints of Parallel Processing*, pages 49–53. Springer-Verlag, 1989.

[52] K. Gharachorloo. Memory Consistency Models for Shared-Memory Multiprocessors. PhD Thesis, Stanford University, December 1995.

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

[54] C. J. Glass and L. Ni. Maximally Fully Adaptive Routing in 2D Meshes. In *Proceedings of the International Conference on Parallel Processing*, pages I:101–104, 1992.

[55] C. J. Glass and L. Ni. The Turn Model for Adaptive Routing. In *Proceedings of the International Symposium on Computer Architecture*, pages 278–287, 1992.

[56] A. Gupta, W.-D. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *International Conference on Parallel Processing*, pages I:312–321, Aug 1990.

[57] J. Heinlein et al. Coherent Block Data Transfer in the FLASH multiprocessor. In *IEEE International Parallel Processing Symposium (IPPS)*, April 1997.

[58] J. Heinlien, K. Gharachorloo, S. Dresser, and A. Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, October 1994.

[59] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. P. Singh, R. Simoni, K. Gharachorloo, J. Baxter, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The performance impact of flexibility in the stanford flash multiprocessor. In *The sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, San Jose, CA, October 1994.

[60] J. L. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2nd edition, 1996.

[61] C. Holt et al. The Effects of Latency, Occupancy, and Bandwidth in Distributed Shared Memory Multiprocessors. Technical Report CSL-TR-95-660, Stanford University, 1995.

[62] C. Holt, J. P. Singh, and J. Hennessy. Application and Architectural Bottlenecks in Large Scale Distributed Shared Memory Machines. In *Proceedings of the International Symposium on Computer Architecture*, pages 134–145, May 1996.

[63] R. Horst. ServerNet Deadlock Avoidance and Fractahedral Topologies. In *Proceedings of the International Parallel Processing Symposium*, pages 274–280, 1996.

[64] C. Hristea, D. Lenoski, and J. Keen. Measuring Memory Hierarchy Performance of Cache-Coherent Multiprocessors Using Micro Benchmarks. In *ACM Supercomputing Conference (SC97)*, October 1997.

[65] W. Hsu and P. Yew. The Performance of Hierarchical Systems with Wiring Constraints. In *Proceedings of the International Conference on Parallel Processing*, 1991.

[66] W. T. Y. Hsu and P. C. Yew. Let us Build System-Friendly Networks—Build Them Hierarchically. In *Proceedings of the 1996 ICPP Workshop on Challenges for Parallel Processing*, pages 64–73, August 1996.

[67] L. Iftode, J. P. Singh, and Kai Li. Understanding Application Performance on Shared Virtual Memory Systems. In *International Symposium on Computer Architecture*, June 1996.

[68] Intel Corporation. *Paragon XP/S Product Overview*, 1991.

[69] H. A. Jamrozik et al. Reducing Network Latency Using Subpages in a Global Memory Environment. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 258–267, October 1996.

[70] J.M.Mellor-Crummey and M.L.Scott. Algorithms for Scalable Synchronization on Shared-Memeory Multiprocessors. *ACM Computer Architecture News*, 9(1):21–65, Feb 1991.

[71] K. Johnson. The impact of communication locality on large-scale multiprocessor performance. In *International Symposium on Computer Architecture*, pages 392–402, 1992.

[72] A. Kagi, N. Abolenein, D. C. Burger, and J. R. Goodman. Techniques for Reducing Overheads of Shared-Memory Multiprocessing. In *ACM International Conference on Supercomputing*, pages 11–20, 1995.

[73] K. Kennedy, C. Bender, J. Connolly, J. Hennessy, M. Vernon, and L. Smarr. A Nationwide Parallel Computing Environment. *Communication of the ACM*, 40(11):63–72, 1997.

[74] R. Kesavan, K. Bondalapati, and D. K. Panda. Multicast on Irregular Switch-based Networks with Wormhole Routing. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA-3)*, pages 48–57, February 1997.

[75] R. Kesavan and D. K. Panda. Multiple Multicast with Minimized Node Contention on Wormhole k-ary n-cube Networks. *IEEE Transactions on Parallel and Distributed Systems.* in press.

[76] R. Kesavan and D. K. Panda. Multicasting on Switch-based Irregular Networks using Multi-drop Path-based Multidestination Worms. In *Proceedings of the 2nd Workshop on Parallel Computer Routing and Communication (PCRCW '97), Lecture Notes in Computer Science # 1417*, pages 217–230, June 1997.

[77] R. E. Kessler and J. L. Schwarzmeier. Cray T3D: A new dimension for Cray Research. In *Proc. 38th IEEE Int. Computer Conf.*, pages 176–182, Spring 1993.

[78] J. H. Kim and A. A. Chien. An Evaluation of Planar-Adaptive Routing (PAR). In *Proceedings of the Symposium on Parallel and Distributed Processing*, pages 470–478, 1992.

[79] N. Koike. NEC Cenju-3: A Microprocessor-based Parallel Computer. In *Proceedings of the 8th Int'l Parallel Processing Symposium*, pages 396–401, 1994.

[80] D. A. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas. Data Forwarding in Scalable Shared-Memory Multiprocessors. In *ACM International Conference on Supercomputing*, pages 255–264, 1995.

[81] David Kranz, Kirk Johnson, Anant Agrawal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared Memory: Early Experience. In *Proceedings of Practice and Principle of Parallel Programming.* ACM, May 1993.

[82] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the International Symposium on Computer Architecture*, pages 81–85, 1981.

[83] A. Kumar, P. Mannava, and L. N. Bhuyan. Efficient and Scalable Cache Coherence Schemes for Shared Memory Hypercube Multiprocessors. In *Supercomputing*, 1994.

[84] Akhilesh Kumar and Laxmi N. Bhuyan. Evaluating Virtual Channels for Cache-Coherent Shared-Memory Multiprocessors. In *Proceedings of the 10th ACM International Conference on Supercomputing*, May 1996.

[85] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the International Symposium on Computer Architecture*, pages 302–313, 1994.

[86] L.A.Barroso, K.Gharachorloo, and E.D.Bugnion. Adaptive Backoff Synchronization Techniques. In *International Symposium on Computer Architecture*, 1998.

[87] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7):558–565, July 1978.

[88] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly Scalable Server. In *Proceedings of the 24th IEEE/ACM Annual International Symposium on Computer Architecture (ISCA-24)*, pages 241–251, June 1997.

[89] C. E. Leiserson et al. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, 1992.

[90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 148–159, May 1990.

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

[92] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, Jan 1993.

[93] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, November 1989.

[94] T. Lovett and R. Clapp. STiNG: A CC-NUMA computer system for the commercial marketplace. In *International Symposium on Computer Architecture*, pages 308–317, 1996.

[95] P. Mannava, A. Kumar, and L. N. Bhuyan. An Efficient Implementation of Limited Directory Cache Coherence Schemes for Shared Memory Multiprocessors. In *Proceedings of the*

Fifth Workshop on Scalable Shared Memory Multiprocessors, Int'l Symposium on Computer Architecture, 1995.

[96] P. K. McKinley and D. F. Robinson. Collective Communication in Wormhole-Routed Massively Parallel Computers. *IEEE Computer*, pages 39–50, Dec 1995.

[97] P. K. McKinley, H. Xu, A.-H. Esfahanian, and L. M. Ni. Unicast-based Multicast Communication in Wormhole-routed Networks. *IEEE Transactions on Parallel and Distributed Systems*, 5(12):1252–1265, Dec 1994.

[98] Meiko Limited. *Meiko CS-2 System Overview*, 1994.

[99] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Mar 1994.

[100] W. Michael. A Scalable Coherent Cache System with A Dynamic Pointing Scheme. In *Proceedings of the Supercomputing*, pages 358–367, 1992.

[101] T. Mowry and A. Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessor. *Journal of Parallel and Distributed Computing*, June 1991.

[102] A. Mu, B. Chia, S. Kondapalli, C. Koo, J. Larson, L. Nguyen, R. Sastry, Y. Satsukawa, H.-C. Shih, T. Wicki, C. Wu, K. Yu, and X. Zhang. A 285 MHz 6-port Plesiochronous Router Chip with Non-Blocking Cross-Bar Switch. In *1996 Symposium on VLSI Circuits: Digest of Technical Papers*, pages 136–137, 1996.

[103] S. S. Mukherjee and M. D. Hill. An Evaluation of Directory Protocols for Medium-Scale Shared-Memory Multiprocessors. In *International Conference on Supercomputing (ISC)*, pages 64–74, 1994.

[104] L. Ni and P. K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *IEEE Computer*, pages 62–76, Feb. 1993.

[105] W. Oed. *Massively Parallel Processor System Cray T3D.* Cray Research GmbH, 1993.

[106] S. Pakin, V. Karamcheti, and A. Chien. Fast Messages: Efficient, Portable Communication for Workstation Clusters and Massively Parallel Processors. *IEEE Concurrency*, pages 60–73, April-June 1997.

[107] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM). In *Proceedings of the Supercomputing*, 1995.

[108] D. K. Panda. Fast Barrier Synchronization in Wormhole k-ary n-cube Networks with Multidestination Worms. *Future Generation Computer Systems*, 11:585–602, Nov 1995.

[109] D. K. Panda and D. Basak. Issues in Designing Scalable Systems with k-ary n-cube cluster-c organization. In *Proceedings of the International Workshop on Parallel Processing*, pages 5–10, 1994.

[110] D. K. Panda, D. Basak, D. Dai, R. Kesavan, R. Sivaram, M. Banikazemi, and V. Moorthy. Simulation of Modern Parallel Systems: A CSIM-based approach. In *Proceedings of the 1997 Winter Simulation Conference (WSC'97)*, pages 1013–1020, December 1997.

[111] D. K. Panda, S. Singal, and R. Kesavan. Multidestination Message Passing in Wormhole k-ary n-cube Networks with Base Routing Conformed Paths. Technical Report OSU-CISRC-12/95-TR54, The Ohio State University, December 1995. *IEEE Transactions on Parallel and Distributed Systems*. In Press.

[112] D. K. Panda, S. Singal, and P. Prabhakaran. Multidestination Message Passing Mechanism Conforming to Base Wormhole Routing Scheme. In *Proceedings of the Parallel Computer Routing and Communication Workshop*, pages 131–145, 1994. Available as `Lecture Notes in Computer Science #853, Springer-Verlag`.

[113] L. L. Peterson and B. S. Davie. *Computer Networks: A Systems Approach.* Morgan-Kaufmann, 1996.

[114] G. D. Pifarre, L. Gravano, S. A. Felperin, and J. Sanz. Fully Adaptive Minimal Deadlock-Free Routing in Hypercubes, Meshes, and Other Networks: Algorithms and Simulations. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):247–263, 1994.

[115] Fong Pong. Symbolic State Model: A New Approach for the Verification of Cache Coherence Protocols. PhD Thesis, University of Southern California, August 1995.

[116] Xiaohan Qin and Jean-Loup Baer. On the Use and Performance of Explicit Communication Primitives in Cache-coherent Multiprocessor Systems. In *Proceedings of the Third International Symposium on High Performance Computer Architecture*, February 1997.

[117] S. K. Reinhardt et al. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[118] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the International Symposium on Computer Architecture*, pages 325–337, April 1994.

[119] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the International Symposium on Computer Architecture*, May 1996.

[120] A. Rogers and Kai Li. Software support for speculative loads. In *The Fifth International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 38–50, October 1992.

[121] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, SOftware-Only Approach for Supporting Fine-Grains Shared Memory. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[122] M. D. Schroeder et al. Autonet: A High-speed, Self-configuring Local Area Network Using Point-to-point Links. Technical Report SRC research report 59, DEC, Apr 1990.

[123] H.D. Schwetman. CSIM Reference Manual (Revision 13). Technical report, Microelectronics and Computer Technology Corporation, Austin, Texas, January 1989.

[124] S. L. Scott and G. M. Thorson. The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus. In *Proceedings of the Symposium on High Performance Interconnects (Hot Interconnects 4)*, pages 147–156, August 1996.

[125] F. Silla, M. P. Malumbres, J. Duato, D. Dai, and D. K. Panda. Impact of Adaptivity on the Behavior of Networks of Workstations under Bursty Traffic. In *Proceedings of the International Conference on Parallel Processing*, Minneapolis, MN, August 1998.

[126] J. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, 1992.

[127] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, pages 5–44, March 1992.

[128] R. Sivaram, R. Kesavan, D. K. Panda, and C. B. Stunkel. Where to Provide Support for Efficient Multicasting in Irregular Networks: Network Interface or Switch? In *Proceedings of the 27th International Conference on Parallel Processing (ICPP '98)*, pages 452–459, August 1998.

[129] R. Sivaram, D. K. Panda, and C. B. Stunkel. Efficient Broadcast and Multicast on Multistage Interconnection Networks using Multiport Encoding. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing*, pages 36–45, Oct 1996.

[130] R. Sivaram, D. K. Panda, and C. B. Stunkel. Efficient Broadcast and Multicast on Multistage Interconnection Networks using Multiport Encoding. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):1004–1028, October 1998.

[131] R. Sivaram, C. B. Stunkel, and D. K. Panda. A Reliable Hardware Barrier Synchronization Scheme. In *Proceedings of the 11th IEEE International Parallel Processing Symposium*, pages 274–280, April 1997.

[132] R. Sivaram, C. B. Stunkel, and D. K. Panda. HIPIQS: A High Performance Switch Architecture using Input Queuing. In *Proceedings of the 12th International Parallel Processing Symposium*, pages 134–143, April 1998.

[133] IEEE Computer Society. IEEE Standard for Scalable Coherence Interface (SCI). *IEEE Standard 1596-1992*, 20, August 1993.

[134] Per Stenstrom. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990.

[135] C. B. Stunkel, R. Sivaram, and D. K. Panda. Implementing Multidestination Worms in Switch-Based Parallel Systems: Architectural Alternatives and their Impact. In *Proceedings of the 24th IEEE/ACM Annual International Symposium on Computer Architecture (ISCA-24)*, pages 50–61, June 1997.

[136] Y. Tamir and G. L. Frazier. Dynamically-Allocated Multi-Queue Buffers for VLSI Communication Switches. *IEEE Transactions on Computers*, 41(6):725–737, June 1996.

[137] Thinking Machine Corporation. *CM5 Technical Summary*, 1991.

[138] M. Tomasevic and V. Milutinovic. *The Cache Coherence Problem in Shared-Memory Multiprocessors: Hardware Solutions.* IEEE Computer Society Press, 1993.

[139] J. Torrellas and Z. Zhang. The Performance of the Cedar Multistage Switching Network. *IEEE Transaction on Parallel and Distributed Systems*, pages 321–336, April 1997.

[140] P. Trancoso, J. Larriba-Pey, Z. Zhang, and J. Torrellas. The Memory Performance of DSS Commercial Workloads in Shared-Memory Multiprocessors. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 250–260, February 1997.

[141] J. H. Upadhyay, V. Varavithya, and P. Mohapatra. Efficient and Balanced Adaptive Routing in Two-Dimensional Meshes. In *International Symposium on High Performance Computer Architecture*, pages 112–121, 1995.

[142] A. S. Vaidya, A. Sivasubramaniam, and C. R. Das. Performance benefits of virtual channels and adaptive routing: An application-driven study. In *Proceedings of ACM International Conference on Supercomputing (ISC'97)*, July 1997.

[143] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *ACM Symposium on Operating Systems Principles*, 1995.

[144] W.-D. Webber, S. Gold, P. Helland, T. Shimizu, T. Wicki, and W. Wilcke. The mercury interconnect architecture: A cost-effective infrastructure for high-performance servers. In *Proceedings of International Symposium on Computer Architecture (ISCA-24)*, pages 98–107, 1997.

[145] M. Welsh, A. Basu, and T. von Eicken. ATM and Fast Ethernet Network Interfaces for User-level Commun ication. In *Proceedings of 3rd International Symposium on High Performa nce Computer Architecture*, pages 332–342, February 1997.

[146] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *International Symposium on Computer Architecture*, pages 24–36, 1995.

[147] S. C. Woo, J. P. Singh, and J. Hennessy. The Performance Advantages of Integrating Block Transfer in Cache Coherent Multiprocessor. In *Proceedings of the Symposium on Operating Systems Principles*, October 1994.

[148] D. A. Wood, S. Chandra, B. Falsafi, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, S. S. Mukherjee, S. Palacharla, and S. K. Reinhardt. Mechanisms for Cooperative Shared Memory. In *Proceedings of the International Symposium on Computer Architecture*, pages 156–167, 1993.

[149] K. G. Yocum, J. S. Chase, A. J. Gallatin, and A. R. Lebeck. Cut-Through Delivery in Trapeze: An Exercise in Low-Latency Messaging. In *IEEE International Symposium on High Performance Distributed Computing (HPDC)*, August 1997.