TOKEN COHERENCE

by

Milo M. K. Martin

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2003

© Copyright by Milo M. K. Martin 2003

All Rights Reserved

Acknowledgments

I have received an incredible amount of support, encouragement, help, and mentoring throughout my time in graduate school. Many of the people I've met during this graduate school journey have greatly influenced my research and forever changed me as a person.

First and foremost I thank my wife, Denise, for all her loving support, encouragement, understanding, and patience throughout all of the ups and down of graduate school. She is my inspiration and the greatest source of joy in my life. I also thank my parents for their encouragement, constant interest in my life, and for not asking "how much longer until you graduate" too many times. They have always been for there for me.

My advisor, Mark Hill, has been an incredible mentor. The depth of knowledge (and perhaps wisdom) that I've gained through my close interaction with Mark is incredible. Not only have I learned about computer architecture, but about many aspects of performing research, communicating with others, time management, and achieving balance in life. Most importantly, I have learned a great deal by simply observing his excellent example. My research and non-research life has forever been changed by Mark's mentoring.

Many of the other faculty members at the University of Wisconsin have also profoundly affected me. I've benefited greatly both from directly interacting with these individuals and from the thriving research environment which they established. I would like to thank the other members of my dissertation committee. Charlie Fischer, Mikko Lipasti, Guri Sohi, and David Wood all provided useful insight into my dissertation research and provided valuable feedback during the process. I especially appreciate their flexibility in scheduling my dissertation defense. As co-leader of the Wisconsin Multifacet Project with Mark Hill, David Wood was like a second advisor to me. He greatly contributed to my success in graduate school, and I have learned much

from him. Charlie Fischer and Jim Goodman were also excellent mentors to me, especially in the critical early years of graduate school. Michael Gleicher, Remzi Arpaci-Dusseau, and Ras Bodik provided excellent information and advice about the academic job search process.

I've met many wonderful and interesting students while in graduate school. Although I can't possibly mention everyone who has enriched my graduate school experience or provided moral support, I wish to specifically thank a few individuals. Dan Sorin and I worked incredibly closely: we co-authored several papers and together developed much of the early Multifacet simulation infrastructure and tools. Dan greatly influenced my research, and ultimately it was during a discussion with him on a plane ride back from HPCA in 2002 that I came up with the idea of Token Coherence. Our complementary set of skills created a prolific collaboration, and hope to collaborate with him again someday. I also thank the other current and former members of the Multifacet Project for their help and encouragement, especially Anastassia Ailamaki, Alaa Alameldeen, Brad Beckmann, Ross Dickson, Michael Marty, Carl Mauer, Kevin Moore, Manoj Plakal, and Min Xu.

Fellow "Architecture Mafia" members Adam Butts, Ravi Rajwar, Amir Roth, Dan Sorin, and Craig Zilles have become life-long friends. They providing an incredible sounding board for brainstorming new research ideas, developing half-baked ideas, and discussing computer architecture in general. They have become close friends, whose friendships have been among the most valuable outcomes of my time in graduate school. I especially thank Craig for putting up with me as an officemate for all those years, and Amir for agreeing to put up with me for years to come as a colleague at Penn.

I've also benefited greatly from interacting technically with many members of the computer industry. While interning one summer at IBM, Steven Kunkel, Hal Kossman and the other engineers I worked with introduced me to server workloads, the challenges of multiprocessor system design, and the important role cache-coherence protocols play in such systems. Without their influence, I would not have pursed this avenue of research. Discussions with Peter Hsu during his one-semester visit to Wisconsin and since that time have been insightful and encouraging. Joel Emer has provided incredible insight during conversations about my research. It was Joel Emer and Shubu Mukherjee's challenge at HPCA in 2000 to improve my previous work that directly

inspired Token Coherence. I have also learned a great deal about computer architecture through conversations with Allan Baum, Kourosh Gharachorloo, Anders Landin, Shubu Mukherjee, and Aaron Spink.

My computer science education began long before graduate school. I especially thank Max Hailperin, Karl Knight, and Barbara Kaiser, computer science professors from my days at Gustavus Adolphus College. I also thank Chuck Fink, my advisor during my two summers working at Argonne National Lab as an undergraduate intern. Chuck was the first to expose me to research.

I've also benefited greatly from the world-class software and hardware support. Both the UW's Computer Systems Lab (CSL) staff and Condor Project staff have been invaluable in providing and supporting the computer hardware and software environment in the department. I also thank Virtutech AB for their support of Simics, especially Peter Magnusson, Andreas Moestedt, and Bengt Warner.

My graduate work has been financially supported by a wide variety of sources. I was supported by an IBM Graduate Fellowship for five consecutive years. This work was also supported by a Norm Koo Graduate Fellowship, and National Science Foundation Grants (CCR-0324878, EIA-9971256, EIA-0205286, EIA-9971256, CCR-0105721 and CDA-9623632) and donations from Intel Corporation, IBM, and Sun Microsystems.

Table of Contents

Page

List of Tables x

List of Figures xii

Abstract xiv

1 Introduction & Motivation 1

1.1 Cache Coherence Protocols 1
1.2 Three Desirable Attributes 2
1.2.1 Attribute #1: Low-latency Cache-to-cache Misses 3
1.2.2 Attribute #2: No Reliance on a Bus or Bus-like Interconnect 4
1.2.3 Attribute #3: Bandwidth Efficiency 6
1.3 Token Coherence: A New Coherence Framework 7
1.3.1 Decoupling Performance and Correctness 7
1.3.2 Correctness Substrate 8
1.3.3 Performance Policies 9
1.4 Achieving the Three Desirable Attributes 10
1.4.1 TOKENB 10
1.4.2 TOKEND 10
1.4.3 TOKENM 11
1.5 Differences from Previously Published Versions of this Work 12
1.6 Contributions 12
1.7 Dissertation Structure 13

2 Understanding and Extending Traditional Coherence Protocols 15

2.1 Coherent Caching in Multiprocessor Systems 16
2.2 Base States and Abstract Operation of Cache Coherence Protocols 17
2.2.1 The MODIFIED, SHARED, and INVALID States 17
2.2.2 The OWNED State 19
2.2.3 The EXCLUSIVE State 20

Page

2.2.4 Optimizing for Migratory Sharing 22
2.2.5 Upgrade Requests 23
2.3 Interconnection Networks 25
2.3.1 Types of Interconnects: Buses, Indirect, Direct 25
2.3.2 Interconnect Routing 26
2.3.3 Ordering Properties: Unordered, Point-to-Point, Totally-Ordered 27
2.3.4 Our Interconnect Implementations: TREE and TORUS 28
2.4 Snooping Protocols 30
2.4.1 Snooping Protocol Background and Evolution 30
2.4.2 Advantages of Snooping Protocols 31
2.4.3 Disadvantages of Snooping Protocols 32
2.4.4 Comparing Snooping Protocols with Token Coherence 32
2.4.5 Our Snooping Protocol Implementation: SNOOPING 33
2.5 Directory Protocols 34
2.5.1 Directory Protocol Background 34
2.5.2 Advantages of Directory Protocols 36
2.5.3 Disadvantages of Directory Protocols 37
2.5.4 Comparing Directory Protocols with Token Coherence 38
2.5.5 Our Directory Protocol Implementation: DIRECTORY 39
2.6 A Non-Traditional Protocol: AMD's Hammer 40
2.6.1 The Hammer Protocol 41
2.6.2 Our Implementation: HAMMEROPT 42
2.7 Protocol Background Summary 43

3 Safety via Token Counting 44

3.1 Token Counting Rules 44
3.1.1 Simplified Token Counting Rules 45
3.1.2 Invariants for Simplified Token Counting 46
3.1.3 Memory Consistency and Token Coherence 48
3.1.4 The Owner Token and Revised Token Counting Rules 48
3.1.5 Rules for Supporting the EXCLUSIVE State 50
3.1.6 Supporting Upgrade Requests, Special-Purpose Requests, and I/O 53
3.1.7 Reliability of Token Coherence 54
3.1.8 Opportunities Enabled by Token Counting 55
3.2 Token Storage and Manipulation Overheads 55
3.2.1 Token Storage in Caches 56
3.2.2 Transferring Tokens in Messages 56
3.2.3 Non-Silent Evictions Overheads 58

Page

3.2.4 Token Storage in Memory 58
3.2.5 Overhead Summary 65

4 Starvation Freedom via Persistent Requests 66

4.1 Centralized-Arbitration Persistent Requests 68
4.2 Showing That Persistent Requests Can Prevent Starvation 70
4.2.1 Deadlock-Free Message Delivery 70
4.2.2 Receiving All Tokens 72
4.2.3 Receiving Valid Data 72
4.2.4 Persistent Request Deactivation Requirement 73
4.2.5 Summary 73
4.3 Banked-Arbitration Persistent Request 73
4.4 Introducing Persistent Read Requests 74
4.5 Distributed-Arbitration Persistent Requests 75
4.6 Improved Scalability of Persistent Requests 78
4.7 Preventing Reordering of Persistent Request Messages 79
4.7.1 Problems Caused by Reordered Activations and Deactivations 79
4.7.2 Solution#1: Point-to-point Ordering 80
4.7.3 Solution#2: Explicit Acknowledgments 81
4.7.4 Solution#3: Acknowledgment Aggregation 82
4.7.5 Solution#4: Large Sequence Numbers 83
4.7.6 Summary of Solutions 84
4.8 Persistent Request Summary 85

5 Performance Policies Overview 86

5.1 Obligations 86
5.2 Opportunities via Transient Requests 87
5.3 Performance Policy Forecast 87
5.3.1 TokenB 88
5.3.2 TokenD 88
5.3.3 TokenM 88
5.4 Other Possible Performance Policies 89
5.4.1 Bandwidth Adaptive Protocols 89
5.4.2 Predictive Push 90
5.4.3 Multi-Block Request or Prefetch 90
5.4.4 Supporting Hierarchical Systems 91
5.4.5 Reducing the Frequency of Persistent Requests 91

Page

5.5 Roadmap for the Second Part of this Dissertation 92

6 Experimental Methods and Workload Characterization 93

6.1 Simulation Tools 93
6.2 Simulated System 94
6.2.1 Coherence Protocols 95
6.2.2 System Interconnects 95
6.3 Workloads and Measurement Methods 97
6.3.1 Methods for Simulating Commercial Workloads 97
6.3.2 Workload Descriptions 99
6.4 Workload Characterization 100
6.4.1 Characterization of our Base Coherence Protocols 100
6.4.2 Cache-to-Cache Misses Occur Frequently 103
6.4.3 The Performance Cost of Indirection 105
6.4.4 The Bandwidth Cost of Broadcasting 107

7 TOKENB: A Low-Latency Performance Policy Using Unordered Broadcast 109

7.1 TOKENB Operation 109
7.1.1 Issuing Transient Requests 109
7.1.2 Responding to Transient Requests 110
7.1.3 Reissuing Requests and Invoking Persistent Requests 110
7.2 Evaluation of TOKENB 111
7.2.1 Question#1: Are reissued and persistent requests uncommon? 113
7.2.2 Question#2: Can TOKENB outperform SNOOPING? 119
7.2.3 Question#3: Is TOKENB's traffic similar to SNOOPING? 120
7.2.4 Question#4: Can TOKENB outperform DIRECTORY or HAMMEROPT? . . 121
7.2.5 Question#5: How does TOKENB's traffic compare to DIRECTORY and HAMMEROPT? 123
7.2.6 Question#6: How frequently do non-silent evictions occur? 126
7.2.7 Question#7: Does TOKENB scale to an unlimited number of processors? . 126
7.2.8 TOKENB Results Summary 128

8 TOKEND: A Directory-Like Performance Policy 130

8.1 TOKEND's Operation 130
8.2 Soft-State Directory Implementations 132
8.2.1 A Simple Soft-State Directory 132

Page

8.2.2 A More Accurate Soft-State Directory 133
8.3 Evaluation of TOKEND 133
8.3.1 Question#1: Is TOKEND's soft-state directory effective? 134
8.3.2 Question#2: Is TOKEND's traffic similar to DIRECTORY? 135
8.3.3 Question#3: Does TOKEND perform similarly to DIRECTORY? 139
8.3.4 Question#4: Does TOKEND outperform TOKENB? 139
8.3.5 TOKEND Results Summary 140

9 TOKENM: A Predictive-Multicast Performance Policy 142

9.1 TOKENM's Operation 143
9.2 Destination-Set Predictors 145
9.2.1 Predictor Goals 145
9.2.2 Our Approach 146
9.2.3 Common Predictor Mechanisms 146
9.2.4 Three Specific Predictor Policies 147
9.2.5 Capturing Spatial Predictability via Macroblock Indexing 150
9.3 Evaluation of TOKENM 150
9.3.1 Question#1: Does TOKENM Use Less Traffic than TOKENB? 150
9.3.2 Question#2: Does TOKENM Outperform TOKEND? 152
9.3.3 Question#3: Is TOKENM Always Better than TOKENB and TOKEND? . . 154
9.3.4 TOKENM Results Summary 155
9.4 Related Work 156

10 Conclusions 157

10.1 Token Coherence Summary and Conclusions 157
10.2 Future Directions and Further Challenges 158
10.2.1 How Else Can Systems Exploit Token Coherence's Flexibility? 159
10.2.2 Is There a Better Way to Prevent Starvation? 159
10.2.3 Does Token Coherence Simplify Coherence Protocol Implementation? . . . 160
10.3 Reflections on Cache-Coherent Multiprocessors 162
10.3.1 Optimize For Migratory Sharing 162
10.3.2 Decouple Coherence and Consistency 163
10.3.3 Avoid Reliance on a Total Order of Requests 164
10.3.4 Revisit Snooping vs. Directory Protocols 166
10.3.5 Design Cost-Effective Multiprocessor Systems 167
10.3.6 The Increasing Importance of Chip Multiprocessors 170

Page

Bibliography 172

Appendix A: Differences from Martin *et al.*, ISCA 2003 183

List of Tables

Table Page

2.1 MSI State Transitions . . . 18

2.2 MOSI State Transitions . . . 19

2.3 MOESI State Transitions . . . 21

2.4 MOESI State Transitions Optimized for Migratory Sharing . . . 23

6.1 Simulation Parameters . . . 96

6.2 Non-Token Coherence Protocol Results for the TREE Interconnect. . . . 101

6.3 Non-Token Coherence Protocol Results for the TORUS Interconnect. . . . 102

7.1 TOKENB Results for the TREE Interconnect. . . . 114

7.2 TOKENB Results for the TORUS Interconnect. . . . 115

7.3 TOKENB Reissued Requests. . . . 115

7.4 Distribution of Evictions per State. . . . 126

7.5 Results from an Analytical Model of Traffic: TOKENB versus DIRECTORY. . . . 128

8.1 TOKEND Results for the TORUS Interconnect. . . . 134

8.2 TOKEND Reissued Requests. . . . 135

8.3 Results from an Analytical Model of Traffic: TOKEND versus DIRECTORY. . . . 139

9.1 Predictor Policies . . . 149

9.2 TOKENM Results for the TORUS Interconnect. . . . 151

Table Page

9.3 TokenM Reissued Requests. 155

List of Figures

Figure Page

1.1 Characterizing Common Protocols in Terms of Three Desirable Attributes. 3

1.2 Interconnection Network Topologies. 5

1.3 Pictorial Dissertation Overview. 8

1.4 Characterizing Performance Policies in Terms of Three Desirable Attributes. 11

2.1 Interconnection Network Topologies. 29

4.1 Single-Arbiter System. 68

4.2 Arbiter-based Persistent Request Example. 69

4.3 Multiple-Arbiter System. 74

4.4 Distributed-Arbitration System. 76

4.5 Distributed Persistent Request Example. 77

4.6 Using Explicit Acknowledgments. 82

4.7 Algorithm for determining when a recipient should ignore an incoming message. . . . 84

6.1 Miss Rate vs. Cache Size. 103

6.2 Runtime vs. Cache Size. 106

6.3 Degree of Sharing Histogram. 108

7.1 A Diminishing Weighted Average. 112

7.2 Implementing a Diminishing Weighted Average in Hardware. 113

Figure Page

7.3 Runtime of TOKENB and TOKENNULL. 116

7.4 Endpoint Traffic of TOKENB and TOKENNULL. 117

7.5 Interconnect Traffic of TOKENB and TOKENNULL. 118

7.6 Runtime of SNOOPING and TOKENB. 119

7.7 Endpoint Traffic of SNOOPING and TOKENB. 121

7.8 Interconnect Traffic of SNOOPING and TOKENB. 122

7.9 Runtime of DIRECTORY, HAMMEROPT, and TOKENB. 123

7.10 Endpoint Traffic of DIRECTORY, HAMMEROPT, and TOKENB. 124

7.11 Interconnect Traffic of DIRECTORY, HAMMEROPT, and TOKENB. 125

7.12 An Analytical Model of the Traffic of TOKENB and DIRECTORY. 127

8.1 Endpoint Traffic of DIRECTORY and TOKEND. 136

8.2 Interconnect Traffic of DIRECTORY and TOKEND. 137

8.3 An Analytical Model of the Traffic of TOKEND and DIRECTORY. 138

8.4 Runtime of DIRECTORY, TOKEND, and TOKENB 140

9.1 A Bandwidth/Latency Tradeoff. 142

9.2 Destination-Set Predictors as Bandwidth/Latency Tradeoffs. 148

9.3 Endpoint Traffic of TOKENM, TOKEND, TOKENB and DIRECTORY. 152

9.4 Interconnect Traffic of TOKENM, TOKEND, TOKENB and DIRECTORY. 153

9.5 Runtime of TOKENM, TOKEND, TOKENB and DIRECTORY. 154

Abstract

Token Coherence is a framework for creating cache-coherent multiprocessor systems. By decoupling performance and correctness, Token Coherence can simultaneously capture the best aspects of the two predominant approaches to coherence: directory protocols and snooping protocols. These two approaches to coherence have a different set of attractive attributes. Snooping protocols have low-latency and direct processor-to-processor communication, whereas directory protocols are bandwidth efficient and do not require a bus or other totally-ordered interconnect. Token Coherence captures the best aspects of both of these traditional approaches to coherence by creating a correctness substrate that (1) enforces safety (using a new technique we call *token counting*) and (2) prevents starvation (using an infrequently-invoked operation we call a *persistent request*). These two mechanisms form a correctness substrate that provides a foundation for implementing many *performance policies*. These performance policies focus on making the system fast and bandwidth-efficient, but have no correctness responsibilities (because the substrate is responsible for correctness). This decoupling of responsibility between the correctness substrate and performance policy (1) enables the development of performance policies that capture many of the desirable attributes of snooping and directory protocols and (2) provides ample opportunity for other performance policies that result in better cache-coherence protocols.

The most important contribution of this dissertation is the observation that simple token counting rules can ensure that the memory system behaves in a coherent manner. Token counting specifies that each block of the shared memory has a fixed number of tokens and that the system is not allowed to create or destroy tokens. A processor is allowed to read a block only when it holds at least one of the block's tokens, and a processor is allowed to write a block only when it holds all of its tokens. These simple rules prevent a processor from reading the block while another processor is writing the block, ensuring coherent behavior at all times. This guarantee of safe system behavior forms the foundation of the new coherence framework that we call Token Coherence.

Chapter 1

Introduction & Motivation

The performance and cost of database and web servers are important because the services they provide are increasingly a part of our daily lives. Many of these servers are shared-memory multiprocessors (or clusters of shared-memory multiprocessors). Shared-memory multiprocessors use a cache coherence protocol to coordinate the many caches distributed throughout the system as part of providing a consistent view of memory to the processors. This dissertation focuses on improving the cache coherence protocol because of its effect on both the cost and performance of shared-memory multiprocessors.

This chapter briefly describes cache coherence protocols (Section 1.1) and identifies three desirable attributes for coherence protocols (Section 1.2). We then present Token Coherence, our new framework for coherence protocols (Section 1.3). We show how this framework enables protocols that capture all three desirable attributes simultaneously (Section 1.4), and in Section 1.5 we describe the differences between this dissertation and previously published versions of this work [80, 81]. We conclude the chapter by presenting the contributions (Section 1.6) and structure of this dissertation (Section 1.7).

1.1 Cache Coherence Protocols

A coherence protocol provides a consistent view of memory by segmenting the shared address space into blocks and controlling the permissions for locally cached copies of these blocks.

Invalidation-based cache coherence protocols[1] manage these permissions to enforce the *coherence invariant*. Informally, the coherence invariant states that (1) no processor may read the block while another processor is writing the block, and (2) all readable copies must contain the same data. To enforce this invariant, current protocols encode the specific permissions and other attributes of blocks in caches using a subset of the MODIFIED, OWNED, EXCLUSIVE, SHARED, and INVALID (MOESI) coherence states [116].

Even though modern protocols adopt these basic states (or a subset of these states), a designer must choose one of several specific approaches for manipulating and controlling these states. Today, the two most common approaches to cache coherence are snooping protocols and directory protocols. *Snooping protocols* broadcast requests to all processors using a bus or bus-like interconnect (*i.e.*, one that provides a total order of requests). This ordered broadcast both (1) unambiguously resolves potentially conflicting requests, and (2) directly locates the block even when it is in another processor's cache. In contrast, *directory protocols* send requests only to the home memory which responds with data or forwards the request to one or more processors. This approach reduces bandwidth consumption, but increases the latency of some misses. Chapter 2 presents a more in-depth discussion the cache coherence problem (Section 2.1), the MOESI coherence states, multiprocessor interconnects (Section 2.3), snooping protocols (Section 2.4), directory protocols (Section 2.5), and other approaches to coherence (Section 2.6).

1.2 Three Desirable Attributes

In our view, workload and technology trends point toward a new design space that provides opportunities to improve the performance and cost of multiprocessor servers by moving beyond traditional snooping and directory protocols. We explore this design space by identifying three desirable attributes of cache coherence protocols driven by workload and technology trends. As illustrated in Figure 1.1, neither predominant approach to coherence captures all three of these attributes: two are captured by directory protocols and one is captured by snooping. This deficiency

[1] This dissertation considers only invalidation-based coherence protocols because recent systems have overwhelmingly chosen invalidation-based protocols over the alternatives.

Low-Latency Cache-to-Cache Misses

No Reliance on a Bus or Bus-Like Interconnect

Bandwidth Efficiency

(a) Snooping Protocols

Low-Latency Cache-to-Cache Misses

No Reliance on a Bus or Bus-Like Interconnect

Bandwidth Efficiency

(b) Directory Protocols

Figure 1.1 Characterizing Common Protocols in Terms of Three Desirable Attributes. This figure illustrates the three desirable properties described in Section 1.2 and which of these attributes are exhibited by snooping (part a) and directory protocols (part b). Each triangle represents a different protocol, and each vertex corresponds to a different attribute. The shaded portion of the triangle shows the attributes exhibited by the corresponding protocol. As shown in this figure, the set of these desirable attributes captured by snooping and directory protocols is disjoint, and thus neither protocol has all the desired properties.

of the current approaches motivates our creation of a new framework for coherence that enables protocols to capture all three of the attributes described next.

1.2.1 Attribute #1: Low-latency Cache-to-cache Misses

Many commercial workloads exhibit abundant thread-level parallelism, and thus using multiple processors is an attractive approach for increasing their performance. To efficiently support the frequent communication and synchronization in these workloads, servers should optimize the latency of *cache-to-cache misses* [17]. A cache-to-cache miss is a miss—often caused by accessing shared data—that requires another processor's cache to supply the data. To reduce the latency of cache-to-cache misses, a coherence protocol should ideally support direct cache-to-cache misses. For example, snooping protocols support fast cache-to-cache misses by broadcasting all requests to find the responder directly. In contrast, directory protocols indirectly locate remote data by placing a directory lookup and a third interconnect traversal on the critical path of cache-to-cache misses.

1.2.2 Attribute #2: No Reliance on a Bus or Bus-like Interconnect

Unfortunately, snooping protocols rely on a bus or bus-like interconnect to enable their fast cache-to-cache transfers, and such interconnects are not a good match with two important technology trends: high-speed point-to-point links and increasing levels of integration. As discussed briefly below and more extensively in Section 2.3, creating a bus-like or "virtual bus" interconnect requires the interconnect to provide a total order of requests. An interconnect provides a *total order* if all messages are delivered to all destinations in some order. A total order requires an ordering among all the messages (even those from different sources or sent to different destinations). For example, if any processor receives message A before message B, then no processor receives message B before A. Unfortunately, creating a totally-ordered interconnect that exploits both of the two important technology trends described below is infeasible using traditional techniques. For this reason, protocols that rely on a totally-ordered interconnect—such as snooping protocols—are undesirable, and protocols that do not rely on such an interconnect—such as most directory protocols—are more attractive.[2]

High-speed point-to-point links. Continued scaling of the bandwidth of shared-wire buses is difficult because of electrical implementation realities [35]. To overcome this limitation, some multiprocessor systems replace shared-wire buses with high-speed point-to-point links that can provide significantly more bandwidth per pin than shared-wire buses [59]. Although many early snooping systems relied on shared-wire buses, many recent snooping protocols use virtual bus switched interconnects that exploit high-speed point-to-point links. These interconnects provide the bus-like ordering properties required for snooping, often by ordering all requests at the root switch chip (such as the interconnect illustrated in Figure 1.2a).

Higher levels of integration. The increasing number of transistors per chip predicted by Moore's Law has encouraged and will continue to encourage more integrated designs, making "glue" logic (*e.g.*, dedicated switch chips) less desirable. Many current and future systems will integrate processor(s), cache(s), coherence logic, switch logic, and memory controller(s) on a single

[2]We describe additional, second-order reasons to avoid protocols that rely on a total order of requests in Section 10.3.3.

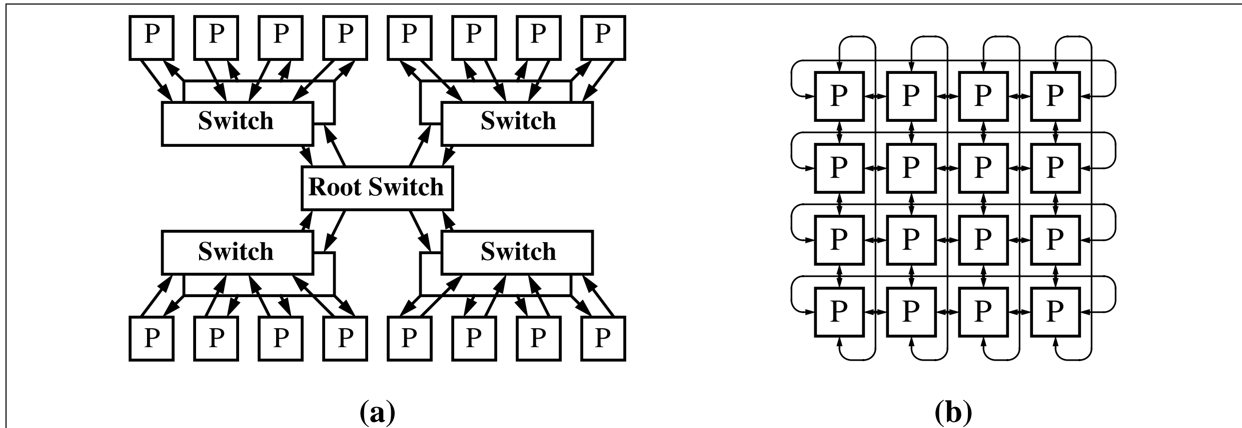


Figure 1.2 Interconnection Network Topologies. (a) 16-processor two-level tree interconnect and (b) 16-processor (4x4) two-dimensional bi-directional torus interconnect. The boxes marked "P" represent highly-integrated nodes that include a processor, caches, memory controller, and coherence controllers. The indirect tree uses dedicated switch chips, while the torus is a directly connected interconnect. For these example interconnects, the torus has lower latency (two vs. four chip crossings on average) and does not require any glue chips; however, unlike the indirect tree, the torus provides no total order of requests, making it unsuitable for traditional snooping.

die (*e.g.*, Alpha 21364 [91] and AMD's Hammer [9]). Directly connecting these highly-integrated nodes leads to a high-bandwidth, low-cost, low-latency "glueless" interconnect (such an interconnect is illustrated in Figure 1.2b).

These glueless, point-to-point interconnects are fast, but they do not easily provide the virtual bus behavior required by traditional snooping protocols.[3] Instead, most such systems use directory protocols, which provide coherence without requiring a totally-ordered interconnect. These systems maintain a directory at the home node (*i.e.*, memory) that resolves possibly conflicting requests by ordering requests on a per-cache-block basis. In contrast, snooping protocols rely on a totally-ordered interconnect to resolve conflicting requests. Unfortunately, traditional directory protocols must first send all requests to the home node (to resolve conflicting requests), adding latency to the critical path of cache-to-cache misses.

[3]Martin *et al.* [82] proposed such a scheme for recreating a total order of requests on an unordered interconnect, but it is perhaps too complicated to implement in practice.

1.2.3 Attribute #3: Bandwidth Efficiency

Bandwidth efficiency is the third—and perhaps currently the least important—desirable attribute. A cache coherence protocol should conserve bandwidth to reduce cost and avoid interconnect contention (because contention reduces performance), but a protocol should not sacrifice either of the first two attributes to obtain this less-important third attribute.

Past research has extensively studied the bandwidth efficiency of coherence protocols, especially in terms of the system's scalability (*i.e.*, the growth of system traffic as the number of processors increases). However, the workload trend towards commercial workloads has diminished the incentive to dramatically increase the number of processors in a multiprocessor system. Although some scientific workloads can use thousands of processors, many commercial workloads can only exploit smaller systems. These small multiprocessor systems represent the bulk of total multiprocessor sales [49]. Service providers that need more throughput than a moderately-sized multiprocessor can provide often create clusters of multiprocessors, because they also desire availability and know that little commercial software runs on multiprocessors with hundreds of processors. As a result, most systems sold have a small to moderate number of processors, and few truly scalable systems are sold. For example, an essay [85] estimated that, of the 30,000 Origin 200/2000 [72] systems shipped, less than 10 systems contained 256 or more processors (0.03%), and less than 250 of the systems had 128 processors or more (1%).

Instead of focusing on system scalability, this dissertation focuses on providing the first two desirable attributes while using less bandwidth than traditional snooping protocols. Snooping protocols broadcast all requests to quickly find shared data in caches or memory, but they use significantly more bandwidth than a directory protocol, which avoids broadcast. Our new coherence framework, described next, allows a system to capture the best performance aspects of snooping and directory protocols (attributes #1 and #2) while using significantly less bandwidth than snooping protocols (attribute #3).

1.3 Token Coherence: A New Coherence Framework

In an effort to capture all three of these desirable attributes, this dissertation proposes Token Coherence, a new framework for creating coherence protocols. Although several research efforts by Martin *et al.* [79, 82, 84] and other researchers (*e.g.*, [3, 4, 20, 70, 73, 114]) have attempted to evolve current approaches into protocols that capture two or more of these attributes, we ultimately discovered a more radical approach for pragmatically capturing the desired attributes.

1.3.1 Decoupling Performance and Correctness

Instead of evolving either a directory protocol or snooping protocol to mitigate their undesirable properties, we revisit fundamental aspects of cache coherence protocols by creating a coherence framework that decouples the performance and correctness aspects of the protocol. In the traditional model (illustrated in Figure 1.3a), a specific cache coherence protocol is built upon the conceptual foundation of the basic MOESI coherence states. We replace this traditional two-tiered model of coherence with a three-tiered model. At the lowest level, Token Coherence still relies on the familiar MOESI conceptual foundation. However, the Token Coherence framework introduces a *correctness substrate* that ensures safety, prevents starvation, and allows for many *performance policies* to be conceptually layered over it. These performance policies have no correctness requirements, which allows them significant flexibility to seek high performance.[4] This layered framework is illustrated in Figure 1.3b.

In the remainder of this section, we describe the most important aspects of this framework. In the next section (Section 1.4), we use this flexibility to create a sequence of performance policies that ultimately capture the three desirable attributes that we described in Section 1.2.

[4]Although Token Coherence decouples performance and correctness, Token Coherence is not a speculative execution technique; Token Coherence does not speculatively modify memory state, and it does not require a rollback or recovery mechanism.

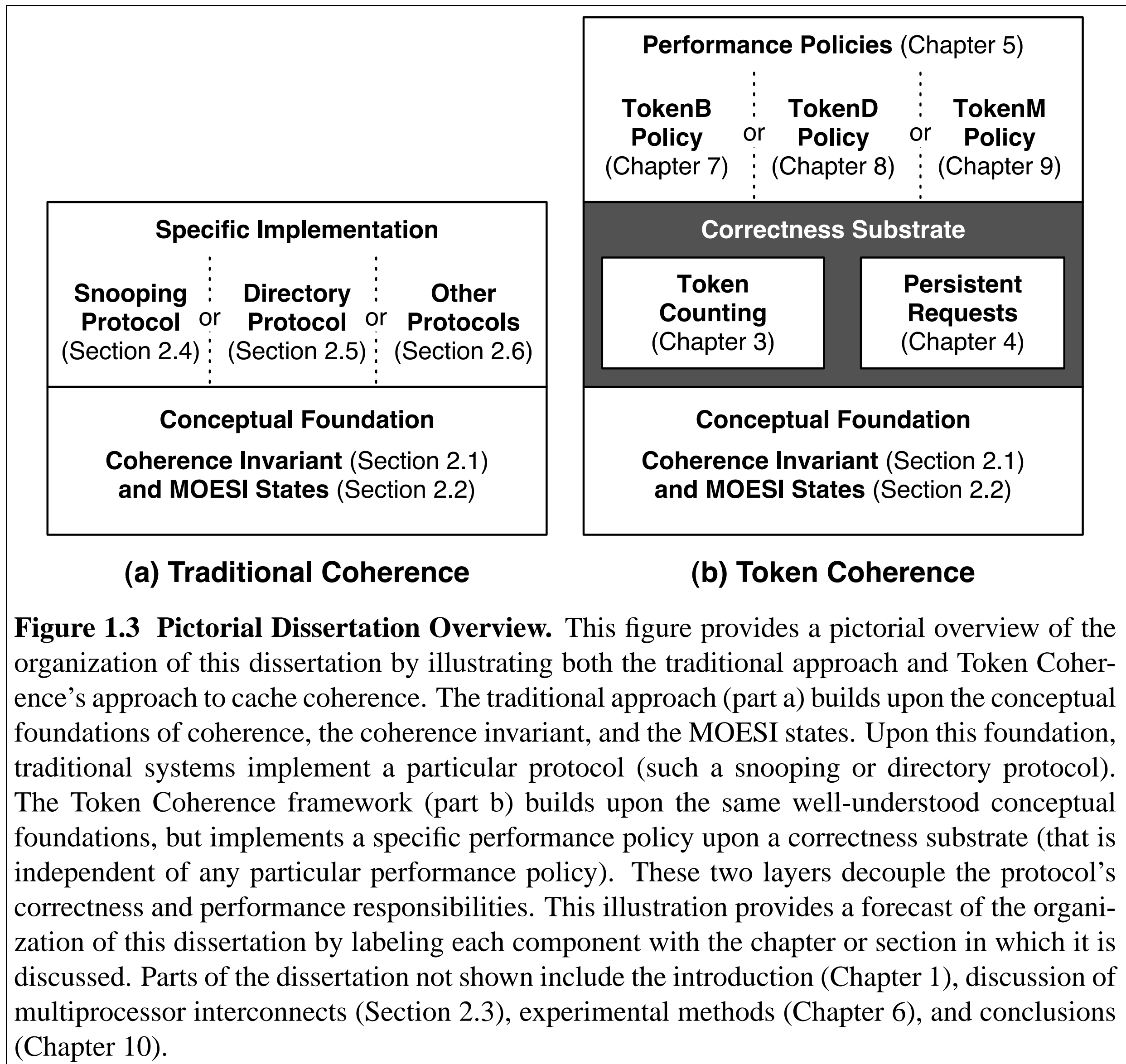


Figure 1.3 Pictorial Dissertation Overview. This figure provides a pictorial overview of the organization of this dissertation by illustrating both the traditional approach and Token Coherence's approach to cache coherence. The traditional approach (part a) builds upon the conceptual foundations of coherence, the coherence invariant, and the MOESI states. Upon this foundation, traditional systems implement a particular protocol (such a snooping or directory protocol). The Token Coherence framework (part b) builds upon the same well-understood conceptual foundations, but implements a specific performance policy upon a correctness substrate (that is independent of any particular performance policy). These two layers decouple the protocol's correctness and performance responsibilities. This illustration provides a forecast of the organization of this dissertation by labeling each component with the chapter or section in which it is discussed. Parts of the dissertation not shown include the introduction (Chapter 1), discussion of multiprocessor interconnects (Section 2.3), experimental methods (Chapter 6), and conclusions (Chapter 10).

1.3.2 Correctness Substrate

The correctness substrate provides a foundation for building correct coherence protocols by separating the correctness aspects of coherence into ensuring safety (do no harm) and preventing starvation (do some good).

Safety. A coherence protocol ensures *safety* if it guarantees that all reads and writes are coherent (*i.e.*, they maintain the single writer or multiple reader coherence invariant). The correctness

substrate ensures safety using token counting. Token counting (1) associates a fixed number of tokens with each logical block of shared memory, and (2) ensures that a processor may read a cache block only when it holds at least one of the block's tokens, and it may write a cache block only when it holds all of the block's tokens (allowing for a single writer or many readers, but not both). Tokens are held with copies of the block in caches and memory and exchanged using coherence messages. We further discuss enforcing safety, token-counting rules, and token overheads in Chapter 3.

Starvation freedom. A coherence protocol is *starvation-free* if all reads and writes eventually complete. The correctness substrate prevents starvation using *persistent requests*. A persistent request is a special type of heavy-weight request that is used in the infrequent situation in which a processor may be starving. This special type of request ensures that—no matter how tokens are moving throughout the system—the requester is guaranteed to eventually receive the tokens and data necessary to complete its request. A processor invokes a persistent request when it detects possible starvation. Persistent requests always succeed in obtaining data and tokens—even when conflicting requests occur—because once activated they persist in forwarding data and tokens until the request is satisfied. Once the request is satisfied, the requester explicitly deactivates the request by sending another round of messages. Because processors should only infrequently resort to persistent requests (*i.e.*, for only a couple percent of cache misses), persistent requests must be correct but not necessarily fast or efficient. We further discuss starvation freedom, persistent requests, and associated overheads in Chapter 4.

1.3.3 Performance Policies

The correctness substrate frees performance policies to seek high performance and bandwidth efficiency without concern for correctness. One way in which performance policies seek high performance is by using *transient requests* as "hints" to direct the correctness substrate to send data and tokens to the requesting processor. A transient request is a simple request that is not guaranteed to succeed (*e.g.*, it may fail to find sufficient tokens because of conflicting concurrent requests), but—in the common case—it often succeeds in obtaining the requested data and tokens.

Because the correctness substrate prevents starvation (via persistent requests) and guarantees safety (via token counting), performance policy bugs and various races may hurt performance, but they cannot affect correctness. We further discuss performance policies, the opportunities they provide, and several possible performance policies in Chapter 5.

1.4 Achieving the Three Desirable Attributes

The flexibility granted by the Token Coherence framework allows the creation of performance policies that can capture all three of the desirable attributes previously described in Section 1.2. This dissertation presents a progression of three performance policies for capturing these three attributes. Figure 1.4 illustrates the attributes captured by each of these policies.

1.4.1 TOKENB

The goal of the *Token-using-Broadcast* (TOKENB) performance policy is to simultaneously capture attributes #1 and #2 (*i.e.*, avoid an ordered interconnect and provide low-latency cache-to-cache misses). These two attributes are the more important of the three attributes, and targeting both of these attributes results in a low-latency, broadcast-based coherence protocol suitable for implementation with a glueless, point-to-point interconnect. TOKENB is faster than traditional snooping protocols (by not requiring a higher-latency totally-ordered interconnect) and directory protocols (by avoiding indirection for frequent cache-to-cache misses). TOKENB is described and evaluated in Chapter 7.

1.4.2 TOKEND

TOKENB broadcasts all requests, and thus it uses substantially more bandwidth than a directory protocol. In an effort to achieve bandwidth efficiency (attribute #3), we first create the *Token-based-Directory* (TOKEND) performance policy. TOKEND uses the Token Coherence framework to emulate a directory protocol, resulting in a protocol that has the same (desirable) bandwidth and (undesirable) latency properties as a traditional directory protocol. The TOKEND performance

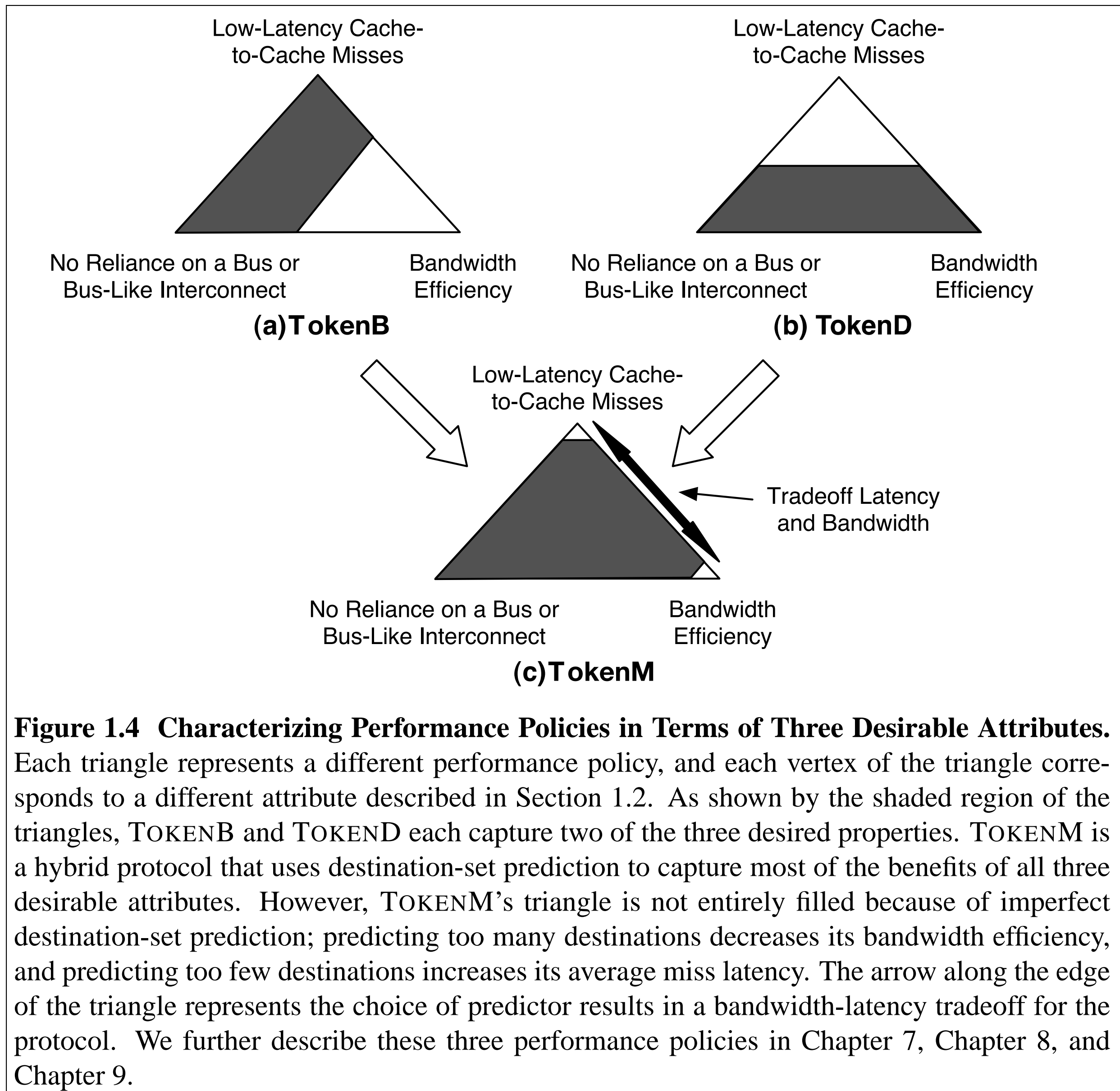


Figure 1.4 Characterizing Performance Policies in Terms of Three Desirable Attributes. Each triangle represents a different performance policy, and each vertex of the triangle corresponds to a different attribute described in Section 1.2. As shown by the shaded region of the triangles, TOKENB and TOKEND each capture two of the three desired properties. TOKENM is a hybrid protocol that uses destination-set prediction to capture most of the benefits of all three desirable attributes. However, TOKENM's triangle is not entirely filled because of imperfect destination-set prediction; predicting too many destinations decreases its bandwidth efficiency, and predicting too few destinations increases its average miss latency. The arrow along the edge of the triangle represents the choice of predictor results in a bandwidth-latency tradeoff for the protocol. We further describe these three performance policies in Chapter 7, Chapter 8, and Chapter 9.

policy illustrates the flexibility of Token Coherence, and it provides a foundation for creating the hybrid performance policy TOKENM. TOKEND is described and evaluated in Chapter 8.

1.4.3 TOKENM

Token-using-Multicast (TOKENM) is a hybrid performance policy that uses *destination-set prediction* to multicast requests to a subset of processors that likely need to observe the requests.

Destination-set prediction has been applied to more traditional protocols [3, 4, 20, 79, 114], and TokenM applies this preexisting predictive technique to Token Coherence. The TokenM performance policy captures attribute #2 (it does not require a totally-ordered interconnect), and allows the system designer to trade off between attributes #1 and #3 (low-latency cache-to-cache misses and bandwidth efficiency). In the extremes, TokenM acts like either TokenB (when the predictor always predicts broadcast) or TokenD (when the predictor only sends requests to the home memory for forwarding if necessary). However, when the predictor uses past behavior to correctly predict which processors need to see various requests, TokenM can capture all three of our desired attributes. TokenM is described and evaluated in Chapter 8.

1.5 Differences from Previously Published Versions of this Work

A preliminary subset of this work was published by Martin *et al.* [80, 81]. This dissertation extends these earlier documents by refining the token-counting rules, adding a clean/dirty owner token, describing the overheads associated with token counting, introducing another approach for implementing persistent requests, and evaluating three distinct performance policies. In contrast, the earlier publications explored only one performance policy. This dissertation also builds upon another work by Martin *et al.* [79] that explores destination-set prediction in coherence protocols. We use these destination-set predictors as part of constructing TokenM, a predictive hybrid protocol based upon Token Coherence that captures the best aspects of snooping and directory protocols. As many readers of this dissertation will be familiar with one or more of these previous publications, we further describe the most important differences between these earlier works and this dissertation in Appendix A.

1.6 Contributions

This section describes our view of this dissertation's most important contributions.

- **Proposes a coherence framework that decouples correctness and performance.** This dissertation proposes decoupling coherence protocols into a correctness substrate and a per-

formance protocol. This decoupling provides significant flexibility for seeking high performance to enable protocols that overcome the limitations of previous approaches to coherence.

- **Introduces simple token-counting rules for enforcing safety.** The most important contribution of this dissertation is the observation that simple token counting rules can provide safety in all cases. By requiring that a processor must hold at least one of a block's tokens to read it and all of a block's tokens to write it, the correctness substrate guarantees safety without unnecessarily constraining or complicating the protocol.

- **Develops a persistent request mechanism for preventing starvation.** Although token counting enforces safety, it does nothing to guarantee that processors will receive the tokens they need to complete their reads and writes; thus, the third contribution of this dissertation is using a dedicated mechanism to prevent starvation. Such a mechanism is designed to be invoked only rarely, and thus can have higher overhead and/or latency than the mechanism used in the common case.

- **Develops and evaluates three performance policies.** We develop and quantitatively evaluate a sequence of three performance protocols. TOKENB, our broadcast-based performance policy, outperforms a traditional snooping protocol by avoiding the overhead of an ordering interconnect. TOKEND, our directory-like performance policy, has performance characteristics similar to a traditional directory protocol. TOKENM is a predictive protocol that borrows from both TOKENB and TOKEND to capture most of TOKENB's low-latency cache-to-cache misses and TOKEND's bandwidth efficiency. These three performance policies provide an attractive set of alternatives to traditional coherence protocols.

1.7 Dissertation Structure

This dissertation is divided into two parts. The first part of the dissertation describes the general Token Coherence framework. It begins with this introductory and motivational chapter (Chapter 1)

and continues with a chapter that describes the fundamentals of cache coherence protocols (including interconnect networks, snooping protocols, and directory protocols), and how these concepts and approaches relate to Token Coherence (Chapter 2). We then describe the correctness substrate's use of token counting to ensure safety (Chapter 3) and its use of persistent requests to prevent starvation (Chapter 4). The first part of the dissertation ends by presenting an overview of performance policies, describing the performance policies we explore later in the dissertation, and discussing other potentially-attractive performance policies relegated to future work (Chapter 5).

The second part of this dissertation develops and evaluates three specific proof-of-concept performance protocols. It begins by describing our evaluation methods and characterizing our workloads (Chapter 6). It then describes and evaluates each of three performance policies: TOKENB (Chapter 7), TOKEND (Chapter 8), and TOKENM (Chapter 9). The dissertation ends with a chapter that summarizes Token Coherence, reflects on the current state of cache coherence protocols (especially in light of Token Coherence), discusses the future of multiprocessors, and identifies other advantages of token coherence—such as simplicity—that are not evaluated in this dissertation (Chapter 10).

Chapter 2

Understanding and Extending Traditional Coherence Protocols

This chapter describes the background and terminology for understanding the Token Coherence framework for creating cache coherence protocols. Although this chapter reviews some basics of cache coherence, it is not an introduction to cache coherence. It is instead intended to provide a consistent terminology, to define the scope of the protocols we discuss in this dissertation, and to provide insight into how current protocols relate to the Token Coherence framework. We refer the reader to the established textbooks on this topic for further background and introductory material (*e.g.*, Chapter 6 of Hennessy and Patterson [53] and Chapters 5–8 of Culler and Singh [32]). Since invalidation-based coherence has been used in favor of update-based coherence protocols in most recent systems (*e.g.*, [15, 21, 23, 25, 26, 58, 72, 91, 119]), this dissertation only considers invalidation-based cache coherence protocols.

This chapter first discusses the problem of keeping the contents of caches coherent in shared-memory multiprocessors (Section 2.1). Next, we describe the basic strategy and coherence states used by invalidation-based cache coherence protocols to address this problem, and foreshadow how these basic states relate to Token Coherence (Section 2.2). After describing the abstract operation of coherence protocols, we discuss various alternatives for implementing multiprocessor interconnection networks (Section 2.3). The chapter continues by describing snooping protocols and directory protocols (the two main classes of cache coherence protocols), presenting a concrete design for both approaches, and discussing how these approaches relate to Token Coherence (Section 2.4 and Section 2.5). Next, we discuss AMD's Hammer protocol, one of several recent and non-traditional coherence protocols that have some characteristics of both snooping and di-

rectory protocols (Section 2.6). We conclude this chapter with a brief summary of the chapter (Section 2.7).

2.1 Coherent Caching in Multiprocessor Systems

This dissertation considers multiprocessor systems with multiple memory modules in which each processor has one or more levels of private cache memory. When processors in such a system share the same *physical memory address space*, that system is called an *shared-memory multiprocessor*. These systems manage the shared memory address space by dividing the memory into *blocks*. Managing and caching memory at the block granularity—typically 32 to 128 bytes in size—amortizes overheads and allow caches to capture spatial locality. Each block has a single *home memory module* that holds the architected state for the block when it is not present in any caches. Processors cache copies of recently accessed data to both reduce the average memory access latency and increase the effective bandwidth of the memory system. When a processor desires to cache a copy of a block that it can read, it issues a *read request*. When a processor desires a copy of the block it can both read and write, it issues a *write request*. A *coherence transaction* is the entire process of issuing a request and receiving any responses. Processors issue these requests to satisfy load or store instructions that miss in the cache, as well as for software or hardware non-binding prefetches. Requests and responses travel between system components over an *interconnection network*, also (more simply) called the *interconnect*. A *system component* in this context is either a processor or a memory module. A miss that requires another processor's cache to supply the data (often caused by an access to shared data) is called a *cache-to-cache miss*. In contrast, a miss that the memory fully satisfies is called a *memory-to-cache miss*.

Allowing multiple processors to cache local copies of the same block results in the *cache coherence problem*, a problem solved by the introduction of a cache coherence protocol. The goal of a cache coherence protocol is to interact with the system's processors, caches, and memories to provide a consistent view of the contents of a shared address space. The exact definition of a *consistent view* of memory is defined by a *memory consistency model* [5] specified as part of the

instruction set architecture of the system. The simplest and most intuitive memory consistency model is *sequential consistency* [71]. In this dissertation, we assume sequential consistency for both describing and evaluating coherence protocols.

As part of enforcing a consistency model, invalidation-based cache coherence protocols maintain the global invariant—what we will call the *coherence invariant*. The coherence invariant states that for a block of shared memory either (a) zero or more processors are allowed to read it or (b) exactly one processor is allowed to write and read it. For sequentially consistent systems, this invariant can enforce the existence of a single, well-defined value of each block at all times (the value generated by the most recent write to the block). Only this well-defined value may be read by processors in the system; they are not allowed to read a stale version of the data. The coherence invariant is most simply enforced in physical time, but it may also be enforced in logical time in protocols that exploit ordering between requests (*e.g.*, [20, 25, 41, 82, 114]).

2.2 Base States and Abstract Operation of Cache Coherence Protocols

To enforce the coherence invariant, coherence protocols use protocol states to track read and write permissions of blocks present in processor caches. This section describes the well-established MOESI states [116] that provide a set of common states for reasoning about cache coherence protocols, and it foreshadows how Token Coherence uses similar concepts to enforce cache coherence. This section also discusses a progression of conceptualized protocols based on the MOESI states. The section begins by discussing a simple MSI three-state protocol (Section 2.2.1), and it continues by discussing the additions of the OWNED state (Section 2.2.2), the EXCLUSIVE state (Section 2.2.3), an optimization for migratory sharing (Section 2.2.4), and an upgrade request (Section 2.2.5).

2.2.1 The MODIFIED, SHARED, and INVALID States

We first consider the MSI subset of the MOESI states. A processor with a block in the INVALID or I state signifies that it may neither read nor write the block. When a block is not found in a cache, it is implicitly in the INVALID state in that cache. The SHARED or S state signifies that

Table 2.1 MSI State Transitions

State	Processor Action: Load	Processor Action: Store	Processor Action: Eviction	Incoming: Read Req.	Incoming: Write Req.
MODIFIED	hit	hit	writeback → INVALID	send data & writeback → SHARED	send data → INVALID
SHARED	hit	write request → MODIFIED	silent evict → INVALID	(none)	(none) → INVALID
INVALID	read request → SHARED	write request → MODIFIED	(none)	(none)	(none)

a processor may read the block, but may not write it. A processor in the MODIFIED or M state may both read and write the block. These three states (INVALID, SHARED, and MODIFIED) are used to directly enforce the coherence invariant by (a) only allowing a single processor to be in the MODIFIED state at a given time,[1] and (b) disallowing other processors to be in the SHARED state while any processor is in the MODIFIED state.[2] The basic operation of a processor in an abstract MSI coherence protocol is shown in Table 2.1.

When a processor requests a new block, it often must *evict* a block currently in the cache (also known as replacing or victimizing). The effort required to evict a block depends upon the coherence state of the block and upon the specific protocol. For example, most protocols require a data *writeback* to memory when evicting blocks in the MODIFIED state and allow for a silent eviction of cache blocks in the SHARED state. A protocol is said to support *silent evictions* if it allows a processor to evict blocks in SHARED without sending a message. Protocols that do not support silent eviction require that a *eviction notification message* (but not the entire data block) must be sent to the home memory upon eviction of blocks in the SHARED state (*e.g.*, [13, 46, 123]).

[1] As previously stated, some protocols—ones that exploit ordering of requests in the system—enforce coherence invariants not in physical time, but in logical time. Since Token Coherence and many of the other protocols we consider in this dissertation enforce these invariants in physical time, the discussion in this section assumes such protocols.

[2] Throughout this dissertation, we will often shorten a phrase such as "a processor with a block in its cache in the SHARED state" to simply "a processor in SHARED."

Table 2.2 MOSI State Transitions

State	Processor Action: Load	Processor Action: Store	Processor Action: Eviction	Incoming: Read Req.	Incoming: Write Req.
MODIFIED	hit	hit	writeback → INVALID	send data → OWNED	send data → INVALID
OWNED	hit	write request → MODIFIED	writeback → INVALID	send data	send data → INVALID
SHARED	hit	write request → MODIFIED	silent evict → INVALID	(none)	(none) → INVALID
INVALID	read request → SHARED	write request → MODIFIED	(none)	(none)	(none)

As described later (in Section 3.1.1), Token Coherence enforces the coherence invariant by assigning a fixed number of tokens to each block, in which holding all the tokens for a block corresponds to the MODIFIED state, holding one or more tokens for the block corresponds to the SHARED state, and holding no tokens for a block corresponds to the INVALID state. Encoding these states using token counts directly disallows a processor from reading a block (which requires only a single token) while another processor is writing the same block (which requires *all* tokens). Unlike most traditional protocols, Token Coherence must maintain a fixed number of tokens, and thus Token Coherence requires that a processor issue either a writeback or a notification of eviction operation whenever evicting a block for which it holds any tokens.

2.2.2 The OWNED State

The optional OWNED or O state in a processor's cache allows read-only access to the block (much like SHARED), but also signifies that the value in main memory is incoherent or *stale*. Thus the processor in OWNED must update the memory before evicting a block. As with the MODIFIED state, only a single processor is allowed to be in the OWNED state at one time. Unlike MODIFIED, however, other processors are allowed to be in the SHARED state when one processor is in the OWNED state. The basic operation of a processor in an abstract MOSI coherence protocol is shown in Table 2.2.

The addition of the OWNED state has two primary advantages.

- First, the OWNED state can reduce system traffic by not requiring a processor to update memory when it transitions from MODIFIED to SHARED during a read request. In a protocol without the OWNED state, the responder would transition from MODIFIED to SHARED, both providing data to the requester *and updating memory* (as shown in Table 2.1). With the addition of the OWNED state, the MODIFIED processor transitions to OWNED, and need not send a message to the memory at that time (as shown in Table 2.2). If another processor issues a write request for the block before it is evicted from the OWNED processor's cache, memory traffic is reduced.

- Second, in some protocols (*e.g.*, systems based on IBM's NorthStar/Pulsar processors [21, 22, 67]) a processor can respond more quickly by providing data from its SRAM cache than the home memory controller can respond from its DRAM. To facilitate this optimization, a processor in OWNED is often given the additional responsibility of responding to requests for data, providing a convenient mechanism for selecting a single responder. In contrast, supplying data from an OWNED copy can be significantly slower than a response from memory in other protocols (*e.g.*, most directory protocols). These systems often do not implement the OWNED state (favoring lower latency at the cost of additional traffic).

As described later (in Section 3.1.4), Token Coherence incorporates the benefits of an OWNED state by distinguishing a single token as the *owner token*. The owner token is always transfered with valid data, and the processor that holds the owner token is responsible for updating memory upon eviction.

2.2.3 The EXCLUSIVE State

The final MOESI state is the EXCLUSIVE or *E* state. The EXCLUSIVE state is much like the MODIFIED state, except the EXCLUSIVE state implies the contents of memory match the contents of the EXCLUSIVE block. By distinguishing this *clean* EXCLUSIVE state from its corresponding

Table 2.3 MOESI State Transitions

State	Processor Action: Load	Processor Action: Store	Processor Action: Eviction	Incoming: Read Req.	Incoming: Write Req.
MODIFIED	hit	hit	writeback → INVALID	send data → OWNED	send data → INVALID
EXCLUSIVE	hit	hit → MODIFIED	silent evict → INVALID	send data → SHARED †	send data → INVALID
OWNED	hit	write request → MODIFIED	writeback → INVALID	send data	send data → INVALID
SHARED	hit	write request → MODIFIED	silent evict → INVALID	(none)	(none) → INVALID
INVALID	read request (response: shared) → SHARED *– or –* read request (response: clean) → EXCLUSIVE	write request → MODIFIED	(none)	(none)	(none)

† When a processor in EXCLUSIVE receives an incoming read request, some protocols will transition to SHARED (as shown here). In some protocols this transition will require notifying the memory. Other protocols will transition to OWNED or possibly a special clean-OWNED state.

dirty MODIFIED state, a block in EXCLUSIVE can be evicted without updating the block at the home memory.[3] When no other processor is caching the block, the memory responds to a read request with a clean-data response. The requesting processor transitions to EXCLUSIVE, granting it read/write permission to the block without the added burden of updating memory. The block can be later be quickly written without an external coherence request by silently transitioning from EXCLUSIVE to MODIFIED (requiring a writeback upon subsequent eviction). The basic operation of a processor in an abstract MOESI coherence protocol is shown in Table 2.3. The EXCLUSIVE state can also be implemented without the OWNED state, resulting in a MESI protocol (not shown).

As described later (in Section 3.1.5), Token Coherence can incorporate the benefits of the EXCLUSIVE state by allowing the owner token to be either *clean* or *dirty*. When the owner token is

[3] Depending on the specifics of the protocol, evicting a block in the EXCLUSIVE state can be either a silent eviction or a require an eviction notification.

in the clean state, a processor can evict the block by sending only an eviction notification, avoiding a full data block writeback.

2.2.4 Optimizing for Migratory Sharing

Although the standard MOESI coherence protocols are efficient for some multiprocessor access patterns, researchers have proposed modifying the operation of these protocols to optimize for migratory sharing patterns [31, 115]. Migratory sharing patterns are common in many multiprocessor workloads, and they result from data blocks that are read and written by many processors over time, but by only one processor at a time [122]. For example, migratory sharing occurs when shared data is protected by lock-based synchronization. In standard MOESI coherence protocols, the resulting read-then-write sequences generate a read miss followed by a write miss.

The protocols we describe and evaluate in this dissertation target this read-then-write pattern of sharing by making a minor modification to the basic MOESI protocol transitions. Although somewhat different from the originally proposed migratory sharing proposals [31, 115], this enhancement successfully targets these same migratory sharing patterns. When a processor is in MODIFIED, it has the choice of acting like a standard MOESI protocol by responding to an external read request with a *shared-data response* and transitioning to OWNED. However, in our variant, the processor also has the option of sending a *migratory-data response* and transitioning to INVALID. When the requesting processor receives the migratory-data response, it transitions immediately to the MODIFIEDMIGRATORY state. This special state signifies that the block is dirty, the processor can read and write it, and the block has not yet been written *by this processor*. A processor in the MODIFIEDMIGRATORY state silently transitions to MODIFIED when it writes the block.

If a workload contained only these read-then-write patterns, the policy of always responding with migratory data would perform well; however, this policy substantially penalizes other sharing patterns (*e.g.*, widely shared data). To find a balance, we employ the heuristic of only sending migratory data when the responding processor is in the MODIFIED state. In contrast, a processor in MODIFIEDMIGRATORY behaves as a standard MOESI protocol (responding to read requests by

Table 2.4 MOESI State Transitions Optimized for Migratory Sharing

State	Processor Action: Load	Processor Action: Store	Processor Action: Eviction	Incoming: Read Req.	Incoming: Write Req.
MODIFIED	hit	hit	writeback → INVALID	send migratory data → INVALID	send data → INVALID
MODIFIED-MIGRATORY	hit	hit → MODIFIED	writeback → INVALID	send data → OWNED	send data → INVALID
EXCLUSIVE	hit	hit → MODIFIED	silent evict → INVALID	send data → SHARED	send data → INVALID
OWNED	hit	write request → MODIFIED	writeback → INVALID	send data	send data → INVALID
SHARED	hit	write request → MODIFIED	silent evict → INVALID	(none)	(none) → INVALID
INVALID	read request (response: shared) → SHARED *– or –* read request (response: clean) → EXCLUSIVE *– or –* read request (response: migratory) → MODIFIED-MIGRATORY	write request → MODIFIED	(none)	(none)	(none)

giving away a SHARED copy and retaining an OWNED copy of the block). Because a processor in MODIFIEDMIGRATORY transitions to MODIFIED when it writes the block, a processor will send a migratory-data response only if it has written the block since the processor received it. The operation of this optimized system is shown in Table 2.4.

2.2.5 Upgrade Requests

In addition to issuing read requests and write requests, some systems implement the optional enhancement of allowing processors to issue an *upgrade request* to transition from SHARED to MODIFIED without redundantly transferring the data block. The decision to implement upgrade

requests in a protocol depends upon the possible advantages, the complexity of implementation, and the frequency of SHARED to MODIFIED transitions.

Upgrade requests have two possible advantages. First, an upgrade request reduces traffic by avoiding a data transfer when the requester already has a SHARED copy of the block in its cache. (The OWNED to MODIFIED transition already avoids the data transfer, *i.e.*, the OWNED processor knows it does not need to send data to itself.) Second, upgrade requests can have lower latency on some split-transaction bus-based multiprocessor designs by entirely avoiding the data response phase of the coherence transaction. In other types of systems (*e.g.*, directory protocols) upgrades have little or no latency benefit, but are still often implemented to reduce traffic.

Although upgrade requests may be advantageous, adding upgrades to a protocol can introduce substantial complexity. In our abstract model of coherence protocols, allowing upgrade requests appears simple. However, concrete protocols (some of which are discussed later in this chapter) are significantly more complicated due to the non-atomic nature of coherence transactions. For example, although a processor may have a SHARED block when it initiates an upgrade request, it may no longer have a SHARED copy of the block when the transaction would normally complete. This window of vulnerability can be closed in various protocol-specific ways, but it does complicate the cache coherence protocol.

Perhaps the most important consideration when deciding whether to implement upgrade requests is their frequency. Although upgrade requests do occur, they are much less frequent when employing the migratory sharing optimization we described in Section 2.2.4. Because this optimization eliminates such a significant fraction of SHARED to MODIFIED transitions, the additional complexity of upgrade requests may be too high a cost for a small reduction in traffic (and latency in some protocols).

We decided not to implement upgrade requests in any of the protocols we describe and evaluate in this dissertation. Since all of the protocols use the migratory sharing optimization, upgrade requests are infrequent. Thus, we believe the extra complexity is too high a cost for this infrequent benefit. Although we do not evaluate protocols with upgrade requests, for completeness we describe how Token Coherence could support the upgrade request in Section 3.1.6.

2.3 Interconnection Networks

The goal of a multiprocessor interconnect is to provide low-cost, low-latency, high-bandwidth, and reliable message delivery between system components (processors and memory modules). The relative cost, latency and bandwidth of an interconnect depends on many factors, including topology, routing, and ordering properties. This section (1) briefly highlights the design considerations that are relevant to our discussion of Token Coherence and (2) describes three concrete interconnects that we use in our later evaluations. A more complete and detailed discussion of interconnects can be found in a book dedicated to this subject by Duato *et al.* [36].

2.3.1 Types of Interconnects: Buses, Indirect, Direct

We consider three main types of interconnects in this dissertation.

- **Bus-based interconnects.** A *bus-based interconnect* connects many system components to the same set of physical wires. A component sends a message by (1) arbitrating for the bus (to avoid having multiple processors driving the bus at the same time), and (2) driving the message on the bus (allowing all components on the bus to observe the message). Since all components can observe or "snoop" transactions on the bus, such interconnects support broadcast with little additional cost. Unfortunately, electrical implementation issues limit the performance of shared-wire buses [35, 59]. Instead, designers are looking to high-speed point-to-point links to build both indirect and direct interconnects.

- **Indirect interconnects.** An *indirect interconnect* uses dedicated switch chips to connect many system components. The simplest indirect interconnect is a single crossbar switch chip with multiple components attached to it. More sophisticated indirect interconnects use a hierarchy of switches or many stages of switches to form a multi-stage interconnection network (MIN). These interconnects are often used to provide uniform-latency and high-bandwidth any-to-any communication. Single-switch crossbars and some hierarchical indirect interconnects use a centralized root switch. Such interconnects are desirable because

they provide ordering properties required for snooping protocols, as discussed later in this section.

- **Direct interconnects.** In contrast to indirect interconnects, *direct interconnects* avoid distinct switch chips and instead directly connect nodes that contain a switch and one or more system components (processors and memory controllers). Popular topologies include a grid-like arrangement of nodes, naturally creating an interconnect with non-uniform latency and bandwidth between various system components. Such interconnects may not scale available bandwidth linearly with the number of nodes, but the interconnect cost per node generally does not increase with larger systems.

The main advantage of direct interconnects over indirect interconnects is that direct interconnects do not require dedicated switch chips [36]. Eliminating these dedicated switch chips can reduce cost (fewer system components) and perhaps decrease latency (fewer chip crossings). The main disadvantage of direct interconnects is that they do not provide the bus-like properties required for some coherence protocols. Because comparing direct and indirect interconnects in abstract terms is difficult, we discuss concrete examples of both direct and indirect interconnects in Section 2.3.4.

2.3.2 Interconnect Routing

Switched interconnects (*i.e.*, direct and indirect interconnects) use various techniques to quickly route messages from source to destination (while avoiding congestion and deadlock). Interconnects use virtual channels and virtual networks which reserve buffers to prevent routing and protocol deadlock, respectively. An interconnect designer must choose between store-and-forward (buffer entire message before forwarding it), virtual-cut-through (pipeline by sending out the message before it is completely buffered), and worm-hole routing (only buffer part of the message, using back-pressure to stop the transmission mid-message when congestion occurs). Interconnects can use simple fixed routes to send messages from source to destination, or they may use adaptive

routing to avoid congestion by dynamically adjusting the paths of messages to use less-congested routes.

Multicast routing. To support protocols that send messages to multiple destinations, an interconnect can send a separate point-to-point message to each destination. Alternatively, the interconnect can initiate a single message that fans out across the interconnect [36]. For example, sending a point-to-point message on a two-dimensional torus crosses on average $\frac{1}{2}\sqrt{n}$ links, where n is the number of nodes in the interconnect. Using fan-out multicast routing to send a broadcast to all nodes uses only $n-1$ links [36]. Thus, supporting fan-out multicast routing can significantly reduce interconnect traffic—from $\Theta(n\sqrt{n})$ to $\Theta(n)$—at the cost of additional interconnect complexity and extra control information in the messages. As all of the protocols we explore in this dissertation gain some benefit from such multicast routing, all of the interconnects we use in our evaluation and analysis assume multicast routing.

2.3.3 Ordering Properties: Unordered, Point-to-Point, Totally-Ordered

Interconnects must provide the desired ordering properties upon which the system's coherence protocol depends.

- **Unordered.** Many high-performance interconnects guarantee no ordering properties whatsoever, but these interconnects are only appropriate for protocols that do not require any ordering (*e.g.*, most directory protocols). For example, adaptive routing may cause the interconnect to reorder messages because messages between the same source/destination pair may travel along different routes.

- **Point-to-point.** An interconnect provides *point-to-point ordering* when all messages sent between a pair of components arrive in the order in which they were sent. Interconnects that do not use adaptive routing can often provide point-to-point ordering with little additional effort (*e.g.*, ensuring that all messages sent between a pair of processors travel along the same path). Some protocols exploit point-to-point ordering of one or more virtual networks to more efficiently handle certain protocol corner cases. By requiring point-to-point ordering

on only some of the virtual networks, the interconnect can still employ adaptive routing on the other virtual networks.

- **Totally-ordered.** An interconnect provides a *total order of messages* if all messages are delivered to all destinations in some order. A *total order* requires an ordering among all the messages (even those from different sources or sent to different destinations). For example, if any processor receives message A before message B, then no processor receives message B before A. Many protocols (*e.g.*, snooping protocols) require an interconnect that provides a total order of requests. This property must hold for all requests, for all blocks, and for all messages from all sources. Unfortunately, establishing a total order of requests can add complexity, increase cost, and increase latency. For example, totally-ordered interconnects commonly use some centralized root switch or arbitration mechanism, and such mechanisms are not a good match for direct interconnects. We further describe the importance (and costs) of totally-ordered interconnects when discussing snooping (in Section 2.4) and when describing two concrete example interconnects (next).

2.3.4 Our Interconnect Implementations: TREE and TORUS

To provide a concrete comparison of our protocols on different interconnects, we implemented two conventional interconnects for use in our evaluations. The first interconnect, TREE, is an indirect interconnect that provides a total order of requests (sufficient for supporting snooping). The second interconnect is TORUS, a direct interconnect that does *not* provide a total order (and thus is not appropriate for use in a conventional snooping protocol). Both of these interconnects use bandwidth-efficient multicast routing (described in Section 2.3.2) when delivering messages to multiple destinations. Each interconnect provides sufficient virtual channels to avoid routing deadlock and provides several virtual networks (which coherence protocols use to avoid deadlock). Like many recent systems [25, 72, 91, 119], we assume that these interconnects provide reliable message delivery.

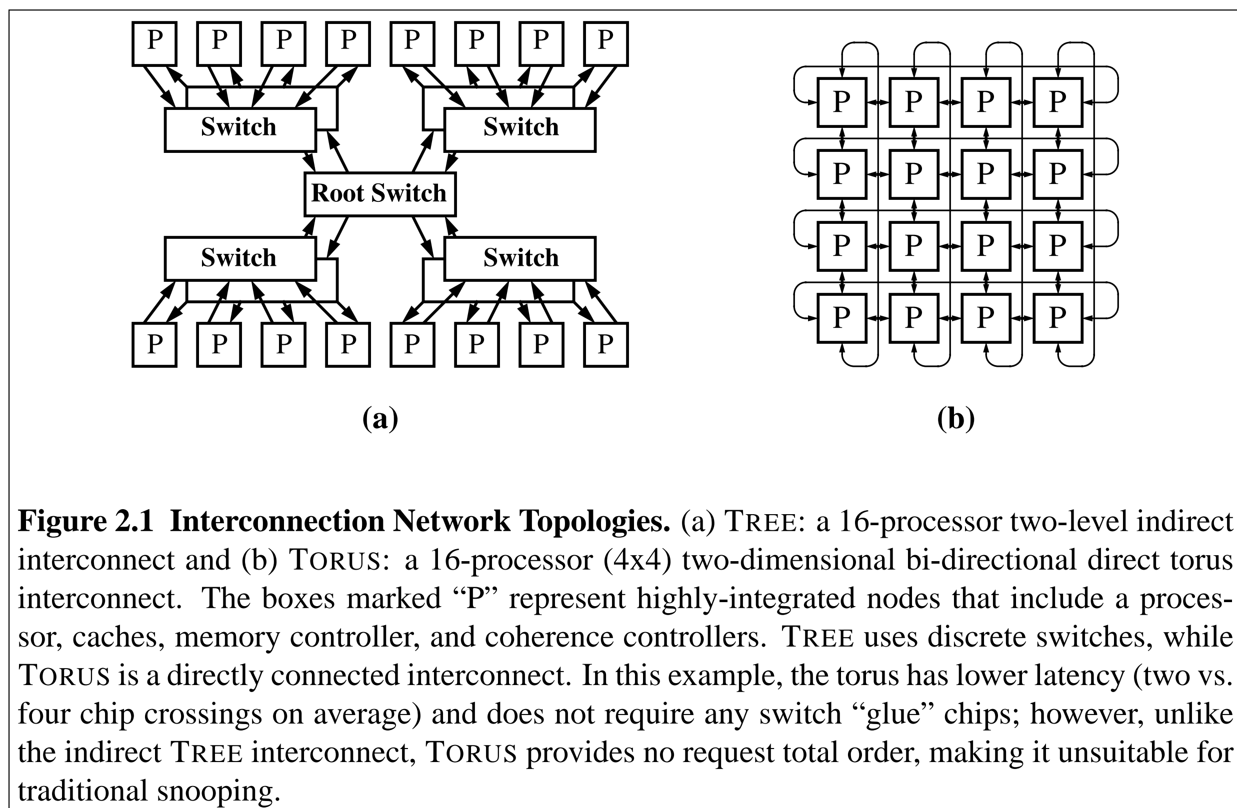


Figure 2.1 Interconnection Network Topologies. (a) TREE: a 16-processor two-level indirect interconnect and (b) TORUS: a 16-processor (4x4) two-dimensional bi-directional direct torus interconnect. The boxes marked “P” represent highly-integrated nodes that include a processor, caches, memory controller, and coherence controllers. TREE uses discrete switches, while TORUS is a directly connected interconnect. In this example, the torus has lower latency (two vs. four chip crossings on average) and does not require any switch “glue” chips; however, unlike the indirect TREE interconnect, TORUS provides no request total order, making it unsuitable for traditional snooping.

TREE interconnect. The TREE interconnect uses a two-level hierarchy of switches to form a pipelined broadcast tree with a fan-out of four, resulting in a message latency of four link crossings. This indirect interconnect provides the total order required for traditional snooping by using a single switch at the root. To reduce the number of pins per switch, a 16-processor system using this topology has nine switches (four incoming switches, four outgoing switches, and a single root switch). This interconnect is illustrated in Figure 2.1a.

TORUS interconnect. The TORUS interconnect is a directly-connected interconnect that uses a two-dimensional, bidirectional torus like that used in the Alpha 21364 [91]. A torus has reasonable latency and bisection bandwidth, especially for small to mid-sized systems. For 16-processor systems, this interconnect has an average message latency of two link crossings. This interconnect is illustrated in Figure 2.1b.

2.4 Snooping Protocols

This section describes *snooping protocols*, the most commonly used approach to building shared-memory multiprocessors. The key characteristic that distinguishes snooping protocols from other coherence protocols is their reliance on a "bus" or "virtual bus" interconnect. Early multiprocessors used a shared-wire, multi-drop bus to connect all processors and memory modules in the system. Snooping protocols exploit such a bus-based interconnect by relying on two properties of a bus: (1) all requests that appear on the bus are visible to all components connected to the bus (processors and memory modules), and (2) all requests are visible to all components in the same total order (the order in which they gained access to the bus) [32]. In essence, a bus provides low-cost atomic broadcast of requests.

2.4.1 Snooping Protocol Background and Evolution

Some snooping protocols exploit the atomic nature of a bus by directly implementing the abstract MOESI protocols previously described. In these systems, processors begin coherence transactions by arbitrating for the shared bus. Once granted access to the bus, the processor puts its request on the bus, and each of the other processors listens to or *snoops* the bus (hence the name snooping protocol). The snooping processors transition their state and may respond with data (as specified in the abstract protocol operation). The memory determines if it should respond by either storing state for each block in the memory (the approach used by the Synapse N+1 [38] as described by Archibald and Baer [14]) or by observing the *snoop responses* generated by the processors.[4] Only after the requesting processor receives its data response (completing its coherence transaction) is another processor allowed to initiate a request.

To increase effective system bandwidth, snooping protocols have introduced many evolutionary enhancements to these atomic-transaction system designs. Split-transaction designs pipeline requests for more efficient bus usage by allowing the requesting processor to release the bus while waiting for its response. Systems also use multiple address-interleaved buses and separate data-

[4]The state in memory or snoop response also determines if a requesting processor should transition to SHARED or EXCLUSIVE on a read request.

response interconnects (buses or unordered point-to-point interconnects) to multiply available bandwidth. Even more aggressive systems avoid the electrical limitations of shared-wire buses entirely and implement a *virtual bus* using point-to-point links, dedicated switch chips, and distributed arbitration. However, these virtual-bus systems still rely on totally-ordered broadcasts for issuing requests. Although each of these enhancements add significant complexity, many snooping systems use these techniques to create high-bandwidth systems with dozens of processors (*e.g.*, Sun's UltraEnterprise servers [25, 26, 27, 113]).

2.4.2 Advantages of Snooping Protocols

The primary current advantage of snoop-based multiprocessors is the low average miss latency, especially for cache-to-cache misses. Since a request is sent directly to all the other processors and memory modules in the system, the responder quickly knows it should send a response. As discussed in Chapter 1, low cache-to-cache miss latency is especially important for workloads with significant amounts of data sharing. If cache-to-cache misses have lower latency than fetching data from memory (*i.e.*, a memory-to-cache miss), replying with data from processor caches whenever possible can reduce the average miss latency. Finally, the tightly-coupled nature of these systems often results in low-latency memory access as well.

In the past, snooping has had two additional advantages, but these advantages are less important in both current and future systems. First, shared-wire buses were cost-effective interconnects for many systems and bus-based coherence provided a complexity-effective approach to implementing cache coherence. Unfortunately, few future high-performance systems will use shared-wire buses due to the difficulty of scaling the bandwidth of shared-wire buses. Second, bus-based snooping protocols were relatively simple. This once-important advantage is now much less pronounced; contemporary snooping protocols using virtual buses are often as complicated or more complicated than alternative approaches to coherence. For example, one potential source of subtle complexity in aggressive snooping protocols is the need to reason about the protocol operation in logical (ordered) time, rather than in physical time. We touch on additional tangential disadvantages of relying on a totally-ordered interconnect in Section 10.3.3.

2.4.3 Disadvantages of Snooping Protocols

The first main disadvantage of snooping is that—even though system designers have moved beyond shared-wire buses—snooping designers are still constrained in their choice of interconnect to those interconnects that can provide virtual-bus behavior (*i.e.*, a total order of requests). These virtual-bus interconnects may be more expensive (*e.g.*, by requiring switch chips), may have lower bandwidth (*e.g.*, due to a bottleneck at the root), or may have higher latency (since all requests need to reach the root).[5] In contrast, an unordered interconnect (such as a directly connected grid or torus of processors) perhaps has more attractive latency, bandwidth, and cost attributes.

The second main disadvantage is that snooping protocols are still by nature broadcast-based protocols; *i.e.*, protocols whose bandwidth requirements increase with the number of processors. Even after removing the bottleneck of a shared-wire bus or virtual bus, this broadcast requirement limits system scalability. To overcome this limitation, recent proposals [20, 79, 114] attempt to reduce the bandwidth requirements of snooping by using *destination-set prediction* (also known as predictive multicast) instead of broadcasting all requests. Although these proposals reduce request traffic, they suffer from snooping's other disadvantage: they rely on a totally-ordered interconnect.

2.4.4 Comparing Snooping Protocols with Token Coherence

Token Coherence captures snooping's primary advantage (low-latency cache-to-cache misses) by supporting direct request-response misses. However, Token Coherence aims to avoid snooping's disadvantages by (1) using token counting to remove the need for a totally-ordered interconnect (explored in Chapter 7), and (2) using destination-set prediction to improve its bandwidth efficiency (explored in Chapter 9).

[5] A recent proposal [82] attempts to overcome these disadvantages by using timestamps to reorder messages on an arbitrary unordered interconnect to reestablish a total order of requests; however, this proposal adds significant complexity to the interconnect.

2.4.5 Our Snooping Protocol Implementation: SNOOPING

To provide a concrete comparison to Token Coherence, we implemented a traditional, but aggressive, MOESI snooping protocol optimized for migratory sharing (described in Section 2.2.4). This dissertation will use the notation SNOOPING to refer to this specific protocol implementation. We based our implementation on a modern snooping protocol [25], and we added additional non-stable states to relax synchronous timing requirements. This change allows the protocol to use an unsynchronized[6]—but totally ordered—interconnect (*e.g.*, the TREE interconnect in Figure 2.1a).

To issue a request, a processor injects the request message into the interconnect, and the requester waits to observe its own request return to it on the interconnect. This method of issuing requests avoids an explicit global-arbitration mechanism by allowing the interconnect to decide the exact total order that requests will be delivered (*e.g.*, a indirect interconnect could order requests at its root). Due to the total order of requests, all processors will observe the requests in the same order, allowing them each to make a globally consistent decision (much like a shared-wire bus). Once a processor has observed its own request, it logically has the permissions and responsibilities associated with its new coherence state, even though in most cases the data response will have not yet arrived. This window of vulnerability leads to many complex race cases.

The total order of requests is also necessary for avoiding explicit acknowledgment messages from each processor. Instead of using explicit acknowledgment messages, when a requesting processor observes its write request in the total order, it knows that all other processors have logically invalidated any copies. However, since the delivery of coherence requests is allowed to be unsynchronized, this invalidation guarantee only holds in logical time—not physical time. As long as all processors enforce coherence permissions in logical time and are careful not to reorder various types of messages, this approach to cache coherence can provide a sequentially consistent view of the memory system to the software.

---

[6]We use the term "unsynchronized" to imply that a request will not need to arrive at all destinations in the same system clock cycle and that this system clock does not need to be tightly synchronized. We do not use this term to imply that the implementation of the system will use any sort of asynchronous logic.

Rather than using snooping response combining (as used in many snooping systems), SNOOPING avoids the complexity and latency of snoop response combining by maintaining two bits per block in memory. The first bit determines if the memory is responsible for responding to requests for the block (*i.e.*, no other processor is in OWNED, EXCLUSIVE, or MODIFIED). The second bit determines if the memory can respond to a read request for the block with an *exclusive-data* response (which enables the recipient to transition to EXCLUSIVE). Avoiding snoop response combining is especially attractive in this protocol because of its non-synchronous nature. The memory controller can encode these two bits using the memory's error correction (ECC) bits, a technique that we discuss in detail in Section 3.2.4.

To provide more specific implementation details, we have made protocol tables and the specification for SNOOPING available on-line [78].

2.5 Directory Protocols

Directory protocols aim to avoid the scalability and interconnection limitations of snooping protocols. Directory protocols actually predate snooping protocols, with Censier and Feautrier [24] and Tang [117] performing early work on directory protocols in the late 1970s. Systems that use these protocols—also know as distributed shared memory (DSM) or cache-coherent non-uniform memory access (CC-NUMA) systems—are preferred when scalability (in the number of processors) is a first-order design constraint. Although these protocols are significantly more scalable than snooping protocols, they often sacrifice fast cache-to-cache misses in exchange for this scalability. Examples of systems that use directory protocols include Stanford's DASH [74, 75] and FLASH [68], MIT's Alewife [7], SGI's Origin [72], the AlphaServer GS320 [41] and GS1280 [33], Sequent's NUMA-Q [76], Cray's X1 [2], and Piranha [18].

2.5.1 Directory Protocol Background

One goal of directory-based coherence is to avoid broadcasting requests by only communicating with those processors that might actually be caching data. To avoid broadcast, a processor issues a request by sending it only to the home memory module for the block. The home memory

contains a *directory* that encodes information about the state, and a superset of processors may be caching each of the blocks in that memory module (hence the name directory protocol). When the home memory receives a request, it uses the directory information to respond directly with the data and/or forward the request to other processors. For example, if the home memory receives a write request from processor P_0 and the directory state signifies that no processors currently hold copies of the data, the memory responds with the data block and updates the directory state to reflect that processor P_0 is now in MODIFIED. Later, when the memory module receives a write request from processor P_1 for the same block, it forwards the request to processor P_0. Processor P_0 responds to the forwarded request by supplying the data to processor P_1. The simplest encoding for this information is a bit vector (one bit per processor) for the sharers and a processor identifier ($\log_2 n$ bits) for the owner. Alternatively, many researchers have proposed approximate encodings to reduce directory state overheads (*e.g.*, [8, 13, 45, 52, 97]). These approximate encodings may specify a superset of sharers to invalidate, but must still include all processors that might be sharing the block. Alternatively, some directory schemes use entries at the memory and caches to form a linked list of processors sharing the block [46].

In addition to tracking sharers and/or owners of a block, the directory also plays a critical role by providing a per-block ordering point to handle conflicting requests or eliminate various protocol races. A *protocol race* can occur anytime multiple messages for the same block are active in the system at the same time. Since the directory observes all requests for a given block, the order in which requests are processed by the directory unambiguously determines the order in which these requests will occur in the system. Many directory protocols use various *busy states* (also know as *pending* or *blocking states*) to delay subsequent requests to the same block by queuing or *negatively acknowledging* (nacking) requests at the directory while a previous request for the same block is still active in the system. Only when the first request has completed are subsequent requests allowed to proceed past the directory. A simple directory protocol might enter a busy state anytime a request reaches the directory; a more optimized protocol enters busy states less frequently. The directory responds or forwards the request as appropriate, and it only clears the busy state when the requester sends the directory an acknowledgment message giving the "all

clear" signal. In contrast, some directory protocols can avoid all busy states, especially protocols that rely on point-to-point ordering in the interconnect.

Another important aspect of directory-based cache coherence is the use of explicit *invalidation acknowledgment* messages to allow requesters to detect completion of write requests. Unlike snooping protocols that use a total order of requests (for all blocks) to enable implicit acknowledgments, most directory protocols eschew a totally-ordered interconnect, and thus these protocols must rely on explicit invalidation acknowledgments.[7] When a requester issues a write request, the directory forwards the request to any potential sharers and/or the owner. When each of these processors receive the forwarded requests they must send an explicit message acknowledging they invalidated the block. Having a per-block ordering point (*i.e.*, the directory) is not sufficient to avoid explicit acknowledgments because for implementing a consistency model the requester must know when its request has been ordered with all other accesses in the system (not just those for the same block).

These three aspects of directory protocols—tracking sharers/owner in a directory, using the directory as a per-block ordering point, and explicit acknowledgments—directly result in both the advantages and disadvantages of directory protocols (described next).

2.5.2 Advantages of Directory Protocols

The two main advantages of directory protocols are their better scalability than snooping protocols and avoidance of snooping's virtual bus interconnect. The dramatically improved scalability of directory protocols is perhaps its most discussed and studied advantage. By only contacting those processors that might have copies of a cache block (or a small number of additional processors when using an approximate directory implementation), the traffic in the system grows linearly with the number of processors. In contrast, the endpoint traffic of broadcasts used in snooping protocols grows quadratically. Combined with a scalable interconnect (one whose bandwidth grows linearly with the number of processors), a directory protocol allows a system to scale to hundreds

[7] The AlphaServer GS320 is a notable exception; it uses a totally ordered interconnect to avoid explicit acknowledgments [41].

or thousands of processors. At large system sizes, two scalability bottlenecks arise. First, the amount of directory state required becomes a major consideration. Second, no interconnect of reasonable cost is truly scalable. Both of these problems have been studied extensively, and actual systems that support hundreds of processors exist (*e.g.*, the SGI Origin 2000 [72]).

The second—and perhaps more important—advantage of directory protocols is the ability to exploit arbitrary point-to-point interconnects. In contrast, snooping protocols are restricted to systems with virtual bus interconnects. Arbitrary point-to-point interconnects (*e.g.*, the TORUS interconnect illustrated in Figure 2.1b) are often high-bandwidth, low-latency, and able to more easily exploit increased levels of integration by including the switch logic on the main processor chip.

2.5.3 Disadvantages of Directory Protocols

Directory protocols have two primary disadvantages. First, the extra interconnect traversal and directory access is on the critical path of cache-to-cache misses. Memory-to-cache misses do not incur a penalty because the memory lookup is normally performed in parallel with the directory lookup. In many systems, the directory lookup latency is similar to that of main memory DRAM, and thus placing this lookup on the critical path of cache-to-cache misses significantly increases cache-to-cache miss latency. While the directory latency can be reduced by using fast SRAM to hold or cache directory information, the extra latency due to the additional interconnect traversal is more difficult to mitigate. These two latencies often combine to dramatically increase cache-to-cache miss latency. With the prevalence of cache-to-cache misses in many important commercial workloads, these higher-latency cache-to-cache misses can significantly impact system performance.

The second—and perhaps less important—disadvantage of directory protocols involves the storage and manipulation of directory state. This disadvantage was more pronounced on earlier systems that used dedicated directory storage (SRAM or DRAM) which added to the overall system cost. However, several recent directory protocols have used the main system DRAM and reinterpretation of bits used for error correction codes (ECC) to store directory state without additional storage capacity overhead (*e.g.*, the S3mp [96], Alpha 21364 [91], UltraSparc III [58],

and Piranha [18, 39]). Storing these bits in main memory does, however, increase the memory traffic by increasing the number of memory reads and writes [39]. Since Token Coherence can use this same technique to hold token counts in main memory, we further discuss this technique, its effectiveness, and its overheads in Section 3.2.4.

2.5.4 Comparing Directory Protocols with Token Coherence

Token Coherence avoids some of the disadvantages of directory protocols while gaining some of their advantages. Token Coherence and directory protocols share the advantage of being able to exploit an unordered interconnect. They also share the disadvantage of needing to hold per-block state in the memory module (the directory state or token count). Like a directory protocol, Token Coherence requires some state to be held by the memory. However, the amount of state per block required by Token Coherence is smaller; the memory is only required to hold a token count (approximately $\Theta(\log_2 n)$, where n is the number of processors in the system). In contrast, directory protocols often require more bits because they encode (exactly or approximately) which processors hold the block. Although Token Coherence requires only a token count in memory, we later describe how Token Coherence performance policies can exploit additional in-memory state for use as hints for finding tokens in a bandwidth-efficient manner (Chapter 8).

The primary difference between Token Coherence and directory protocols is that Token Coherence is a flexible framework for creating a range of coherence protocols, each with a unique latency/bandwidth tradeoff. For example, in Chapter 7 we describe a Token Coherence protocol that—much like a snooping protocol—has both low-latency cache-to-cache misses (by avoiding indirection to the home memory) and poor scalability (by broadcasting). In Chapter 8 we describe a different Token Coherence protocol that captures the scalability and higher-latency cache-to-cache misses of a directory protocol. Finally, in Chapter 9 we show that prediction can be used to achieve most of the bandwidth-efficiency (and thus scalability) of directory protocols with most of the latency advantages of snooping protocols. Previous research by Acacio *et al.* [3, 4] has explored reducing the latency of standard directory protocols by using prediction. Token Coherence

exploits similar opportunities for prediction (Chapter 9), but does so within the more general Token Coherence framework.

2.5.5 Our Directory Protocol Implementation: DIRECTORY

To provide a concrete comparison with Token Coherence, we implemented a directory protocol that we will refer to as DIRECTORY. This protocol is a standard full-map directory protocol inspired by the Origin 2000 [72] and Alpha 21364 [91]. The base system stores the directory state in the main memory DRAM [39, 58, 91, 96], but we also evaluate systems with perfect directory caches by simulating a zero cycle directory access latency.

We designed DIRECTORY to be a low-latency directory protocol, and thus whenever facing a choice between reducing message count or reducing latency, we choose to reduce latency. The protocol requires no ordering in the interconnect (not even point-to-point ordering) and does not use negative acknowledgments or retries, but it does queue requests at the directory controller using busy states in some cases. The directory controller queues messages on a per-block basis, allowing messages for non-busy blocks to proceed. The directory uses per-block fair queuing to prevent starvation.

The directory enters a busy state each time it receives a request. The busy state prevents all subsequent operations on the block if the operation results in a change of ownership; however, multiple read requests to the same block are allowed to proceed in parallel using a special busy state that counts the number of parallel requests (and only unblocks when all of the parallel requests have completed). When any operation completes, the initiating processor sends a *completion message* to the directory to (1) remove the busy state and (2) inform the directory whether the recipient received a clean or migratory data response (so the directory knows if the responding processor invoked the migratory sharing optimization). These additional completion messages create interconnect traffic and increase controller occupancy; however, they also allow the protocol to support all the MOESI states and optimize for migratory sharing.

DIRECTORY supports silent eviction of SHARED blocks, but MODIFIED, OWNED, and EXCLUSIVE blocks require a three-phase eviction process. The process begins when the processor

sends a message to the directory asking permission to evict the block. The directory responds to this message by sending the processor an acknowledgment, and the the directory transitions to a waiting-for-writeback busy state. When the processor receives the acknowledgment, it sends a writeback message with the data (for blocks in MODIFIED and OWNED) or a data-less eviction message (for blocks in EXCLUSIVE). When the directory receives this message, it updates memory and transitions to a non-busy state, completing the process.

Unlike many directory protocols (*e.g.*, [72, 91]) which are MESI protocols, our DIRECTORY protocol uses the MOESI states. As explained in Section 2.2.2, the OWNED state can reduce traffic, but more importantly it can also lower latency by allowing the system to source data from the OWNED processor rather than fetching it from memory. Fetching data from the OWNER processor is faster than fetching data from memory when the combination of accessing the directory and one additional interconnect traversal is faster than a DRAM memory lookup. For example, small systems with a fast interconnect and fast SRAM directory may have faster cache-to-cache misses than memory-to-cache misses. Another advantage of using an MOESI protocol is that it allows all of our protocols to use the same base states, making them easier to compare quantitatively. In the next section, we describe our third and final coherence protocol we use to compare and contrast with token coherence.

To provide more specific implementation details, we have made protocol tables and the specification for DIRECTORY available on-line [78].

2.6 A Non-Traditional Protocol: AMD's Hammer

Several recent designs have used non-traditional coherence protocols, *i.e.*, protocols that are not easily classified as either snooping or directory protocols (*e.g.*, Intel's E8870 Scalability Port [15], IBM's Power4 [119] and xSeries Summit [23] systems, and AMD's Hammer [9, 65, 121]). These systems have a small or moderate number of processors, use a tightly-coupled point-to-point interconnect (but not a virtual bus), and send all requests to all processors. Unfortunately, these systems are currently not well described in the academic literature. In this section we present our

understanding of the AMD Hammer protocol based on the little publicly-available documentation [9, 65, 121]. Even if this description does not accurately reflect the operation of AMD's actual system, the approach we describe is a correctly functioning coherence protocol. After describing the protocol, we then extended it to make it more comparable with our other protocols, describe the advantages and disadvantages of this protocol, and compare it with Token Coherence.

2.6.1 The Hammer Protocol

Systems built from AMD's Hammer chip (also know as the Opteron or Athlon 64 [9, 65, 121]) uses a protocol not previously described in the academic literature. The Hammer protocol targets small systems (in which broadcast is acceptable) with unordered interconnects (in which traditional snooping is not possible), while avoiding directory state overhead and directory access latency. The protocol is interesting because it is being used in an actual system, and because it has a unique blend of attributes of traditional directory and snooping protocols.

A requester first sends its request to the home memory module (much like a directory protocol), and the home memory module allocates a transaction entry to place the block into a busy state. However, unlike a directory protocol, the memory contains no directory state, but instead immediately forwards the request (these forwarded requests are called probes) to all the processors in the system (the broadcast nature and lack of state in memory is reminiscent of a snooping protocol). In parallel with the sending of the forwarded requests, the memory module fetches data from the DRAM and sends it to the requester. When each processor in the system receives the forwarded request, it sends an explicit response or acknowledgment to the requester (like a directory protocol), and when the requester has collected all the responses (one from each processor and another from the memory) it sends a message to the memory module to free the block from the busy state and deallocate the transaction entry. By avoiding a directory lookup, this protocol has lower latency for cache-to-cache misses than a standard directory protocol, but it still requires indirection through the home node. Perhaps the best way to reason about this protocol is as a directory protocol with no directory state (or Dir_0B, using the terminology from Agarwal *et al.* [8]).

This approach has many advantages and disadvantages compared with a traditional directory or snooping protocol. The advantages are that (1) it does not rely on a totally-ordered interconnect and (2) it does not add the latency of a directory lookup to the critical path of cache-to-caches misses (the memory controller forwards the request as soon as it verifies that no other requests are outstanding for the block). The disadvantages are that (1) all requests must first be sent to the home memory (adding an extra interconnect traversal to the critical path of cache-to-cache misses), (2) all requests are forwarded to all processors (creating the same traffic issues as in snooping protocols), (3) all processors must respond with an acknowledgment to the forwarded request messages (adding even more traffic than in snooping protocols, because snooping protocols do not use explicit acknowledgments), (4) when multiple sharers are invalidated the requester must wait for acknowledgments to arrive *from all processors* before it has write permission to the block (in contrast, a directory protocol only needs acknowledgments from those processors to which it forwarded the request, and thus it can avoid waiting for responses from distant non-sharers), and (5) the system sends two data responses for each cache-to-cache miss (one response from the memory—which is ignored—and another from the processor that owns the block). Due to duplicate responses, broadcasting of forwarded request messages, and many explicit acknowledgments, this protocol is most suitable for small systems.

2.6.2 Our Implementation: HAMMEROPT

HAMMEROPT is a reverse-engineered *approximation* of AMD's Hammer protocol [9, 65, 121]. We actually enhanced the basic protocol by (1) adding our migratory sharing optimization and (2) eliminating the redundant data responses for cache-to-cache transfers. To eliminate redundant data responses, we added the same two bits of per-block state at the memory controller that we used in SNOOPING (described in Section 2.4.5). In essence, HAMMEROPT removes one of Hammer's advantages (no per-block state in memory) in exchange for removing one of Hammer's disadvantages (redundant data responses). To reduce the traffic created by broadcasting the forwarded request messages, HAMMEROPT uses multicast routing to provide bandwidth-efficient broadcast.

To provide more specific implementation details, we have made protocol tables and the specification for HAMMEROPT available on-line [78].

2.7 Protocol Background Summary

In this chapter we briefly discussed many aspects of coherence, general approaches to coherence, and how these relate to Token Coherence. This chapter also described specific interconnects (TREE and TORUS) and coherence protocols (DIRECTORY, SNOOPING, and HAMMEROPT) that we use later in this dissertation for evaluating Token Coherence. The next three chapters describe the core of Token Coherence by presenting token counting (Chapter 3), persistent requests (Chapter 4), and the concept of a performance policy (Chapter 5).

Chapter 3

Safety via Token Counting

Token Coherence uses a *coherence substrate* to both ensure safety and avoid starvation. This chapter describes only the coherence substrate's use of token counting to provide safety (*i.e.*, ensuring data are read and written in a coherent fashion); Chapter 4 describes the substrate's use of persistent requests to provide starvation freedom (*i.e.*, guaranteeing that all reads and writes eventually succeed). The complete specification of the Token Coherence substrate is available on-line [78].

In Token Coherence, token counting enforces safety in all cases. The system associates a fixed number of tokens with each block of shared memory.[1] A processor is only allowed to read a cache block when it holds at least one token or write a cache block when holding all tokens. This token-counting approach directly enforces the "single-writer or many-reader" coherence invariant described earlier in Section 2.1. One of the primary benefits of token counting is that it allows Token Coherence to ensure safety without relying on request ordering established by the home memory or a totally-ordered interconnect. This chapter discusses both the rules that govern token counting (Section 3.1) and token storage and manipulation overheads (Section 3.2).

3.1 Token Counting Rules

This section describes token counting by initially presenting a simplified version of token counting—one that requires data to always accompany tokens as they travel throughout the system (Section 3.1.1)—and discusses the invariants it maintains (Section 3.1.2) and how this simple

[1] The system associates tokens with each block of the physical memory (not with virtual block address)

scheme enforces a memory consistency model (Section 3.1.3). The section then continues by presenting two important traffic-reduction refinements that reduce the number of cases that require tokens to be sent with data (Section 3.1.4 and Section 3.1.5). The chapter continues by describing support for other type of requests—*e.g.*, upgrades, cache allocate requests, and input/output (Section 3.1.6). The chapter concludes with a brief discussion of reliability (Section 3.1.7) and opportunities enabled by token counting (Section 3.1.8).

3.1.1 Simplified Token Counting Rules

During system initialization, the system assigns each block a fixed number of tokens, T.[2] The number of tokens for each block (T) is generally at least as large as the number of processors. Tokens are tracked per block and can be held in processor caches, memory modules, coherence messages (in-flight or buffered), and input/output devices. A *coherence message* is any message sent as part of the coherence protocol. We collectively refer to those devices that can hold tokens as *system components*. Initially, the block's home memory module holds all tokens for a block. Tokens and data are allowed to move between system components as long as the substrate maintains these four rules:

- **Rule #1 (Conservation of Tokens):** Once the system is initialized, tokens may not be created or destroyed.
- **Rule #2 (Write Rule):** A system component can write a block only if it holds all T tokens for that block.
- **Rule #3 (Read Rule):** A system component can read a block only if it holds at least one token for that block.
- **Rule #4 (Data Transfer Rule):** If a coherence message contains one or more tokens, it must contain data.

[2] In most implementations, each block will have the same number of tokens. However, token counting continues to work if blocks have a different numbers of tokens as long as all system components know how many tokens are associated with each block.

Rule #1 ensures that the substrate never creates or destroys tokens and enforces the invariant that at all times each block in the system has T tokens. Rules #2 and #3 ensure that a processor will not write the block while another processor is reading it. Adding rule #4 ensures that processors holding tokens always have a valid copy of the data block. In more familiar terms, token possession maps directly to traditional coherence states (described in Section 2.2): holding all T tokens is MODIFIED; one to $T - 1$ tokens is SHARED; and zero tokens is INVALID.

While our rules restrict the data and token content of coherence messages, the rules do not restrict when or to which component the substrate can send coherence messages. For example, to evict a block (and thus tokens) from a cache, the processor simply sends all its tokens and data to the memory.[3] Unlike most coherence protocols, token coherence does not allow silent evictions (*i.e.*, evicting the block without sending any messages). In contrast, many traditional protocols allow silent evictions of *clean* (*i.e.*, SHARED and EXCLUSIVE) blocks. Both token coherence and traditional protocols require writeback messages that contain data when replacing *dirty* (*i.e.*, OWNED and MODIFIED) blocks.

The rules also allow for transferring data without the guarantee that the receiving processor's cache has sufficient space to hold the new data. To handle these "stray" data messages, a processor that receives a message carrying tokens and/or data can choose to either accept it (*e.g.*, if there is space in the cache) or redirect it to the home memory (using another virtual network to avoid deadlock as described in Section 4.2.1).

3.1.2 Invariants for Simplified Token Counting

By implementing the token-counting rules directly at each system component, these local rules lead to several global system invariants (some of which are described below). The substrate maintains these global invariants by induction; the invariants hold for the initial system state, and the locally-enforced rules ensure that all movements of data and tokens preserve the global invariants.

[3] Section 3.1.4 introduces a refinement that enables a processor to evict a clean block without sending the data block back to memory; instead, the processor need send only a small eviction-notification message to the memory.

Thus, in Token Coherence safety is guaranteed without reasoning about the interactions among non-stable protocol states, request indirection, message ordering in the interconnect, or system hierarchy. Examples of global invariants include:

- "The number of tokens for each block in the system is invariant" (because no component creates or destroys tokens and the interconnect provides reliable message delivery[4]).
- "No component may read the block while another component may write the block" (because the writer will be holding all tokens, no other component will have any tokens, and thus may not read the block).
- "At a given point in time, there is only one valid value for every physical address in the shared global address space" (because all tokens must be gathered to write the block and a copy of new contents of the data block is required when subsequently transferring tokens). Although this invariant holds in the memory system, a processor may relax this constraint (*e.g.*, when implementing a weaker memory consistency model or speculating on locks).
- A corollary to the previous invariant: "at a given time, all components that can read a block will observe the same value" (because the contents of the data block are immutable between write epochs, and a writer must collect all the tokens before it can write the block).

Some preliminary research using model checking techniques suggests that these and other invariants can be shown to hold using model checking. Alternatively, these invariants could be shown to hold using other formal methods, possibly based upon "Lamport clocks" [30, 100, 114] or via several other approaches [29, 42, 54, 101, 104, 112]. Further research in this effort is on-going and has been relegated to future work.

[4] We discuss reliability issues in Section 3.1.7

3.1.3 Memory Consistency and Token Coherence

The processors and coherence protocol interact to enforce a memory consistency model [5]—the definition of correctness for multiprocessor systems. Token Coherence plays a similar role as that of a traditional invalidation-based directory protocol. Token Coherence and traditional directory protocols both provide the same guarantee: each block can have either a single writer or multiple readers (but not both at the same time). For example, the MIPS R10000 processors [126] in the Origin 2000 [72] use this property to provide sequential consistency, even without a global ordering point.[5] The Origin protocol uses explicit invalidation acknowledgments to provide the above property. Token Coherence provides the same property by explicitly tracking tokens for each block. Although this property concerns only a *single block* and consistency involves the ordering of reads and writes to *many blocks*, this property is sufficient to allow the processor to enforce sequential consistency or any weaker consistency model.

3.1.4 The Owner Token and Revised Token Counting Rules

An issue with the rules we presented in Section 3.1.1 is that data must always travel with tokens. Requiring tokens to always travel with data is bandwidth inefficient because it leads to the redundant transfer of the block's data (*e.g.*, when a request gathers tokens from many processors or when many evictions of read-only blocks are redundantly sent to the memory).

To avoid these bandwidth inefficiencies, this section presents a modification to the substrate that allows tokens to be transferred in small data-less messages (effectively behaving as either eviction notification messages or invalidation acknowledgment messages in a directory protocol). To enable this enhancement, the substrate distinguishes a unique per-block *owner token*, adds a per-block *valid-data bit* (distinct from the traditional valid-tag bit), and maintains the following five rules (changes in *italics*):

[5]The Origin protocol uses a directory to serialize some requests *for the same block*; however, since memory consistency involves the ordering relationship between different memory locations [5], using a distributed directory is not sufficient to implement a memory consistency model.

- **Rule #1′ (Conservation of Tokens):** Once the system is initialized, tokens may not be created or destroyed. *One token for each block is the owner token.*

- **Rule #2′ (Write Rule):** A system component can write a block only if it holds all T tokens for that block *and has valid data*.

- **Rule #3′ (Read Rule):** A system component can read a block only if it holds at least one token for that block *and has valid data*.

- **Rule #4′ (Data Transfer Rule):** If a coherence message contains *the owner token*, it must contain data.

- **Rule #5′ (Valid-Data Bit Rule):** *A system component sets its valid-data bit for a block when a coherence message arrives with data and at least one token. A component clears the valid-data bit when it no longer holds any tokens.*

Rule #1′ distinguishes one token as the owner token. Rules #2′ and #3′ continue to ensure that a processor will not write the block while another processor is reading it. Rule #4′ allows coherence messages with non-owner tokens to omit data, but it still requires that messages with the owner token contain data (to prevent all processors from simultaneously discarding data). Possession of the owner token but not all other tokens maps to the familiar MOESI state OWNED. Although the owner token provides the traditional advantages of the OWNED state (described in Section 2.2.2), in Token Coherence the owner token further reduces traffic by allowing the transfer of non-owner tokens without including data.

System components (*e.g.*, processors and the home memory) maintain a per-block valid-data bit (in addition to the traditional valid-tag bit in the cache). This valid-data bit allows components to receive and hold tokens for a block without valid data while also preventing them from responding with, reading, or writing stale data. Rules #2′ and #3′ explicitly require the valid-data bit be set before a processor can read or write data for a block. Rule #5′ ensures (1) the valid-data bit will only be set when a component has received data and (2) the valid-data bit is only set while that component is holding tokens for that block.

These revised rules provide three primary advantages. First, evictions of non-owner read-only copies require only a small, data-less coherence message to carry the tokens back to the memory. Second, a request invalidating many read-only copies does not generate a data response for each copy it invalidates. Third, we later use the owner token to conveniently select a single data responder for requests.

3.1.5 Rules for Supporting the EXCLUSIVE State

Using the rules described in Section 3.1.4, Token Coherence can only approximate the EXCLUSIVE state by allowing the memory to send data and all the tokens for a block when a processor requests to read the block.[6] This approach captures the primary latency benefit of the EXCLUSIVE state (*i.e.*, it allows the processor to later write the block without the latency of issuing another request), but a processor evicting a block with all tokens would still be required to send a dirty-writeback message *even when the data was not written by the processor*. In contrast, most traditional protocols (*e.g.*, [72, 91]) can evict blocks in EXCLUSIVE either silently or with a data-less eviction notification.

To allow token coherence to benefit from the EXCLUSIVE state without this extra writeback traffic penalty, we further refine the token counting rules to distinguish between the EXCLUSIVE and MODIFIED states by allowing the owner token to be either *clean* (signifying the value held in memory matches the currently valid value for the block) or *dirty* (signifying the value held in memory is stale)—changes to the rules are marked in *italics*:

- **Rule #1″ (Conservation of Tokens):** Once the system is initialized, tokens may not be created or destroyed. One token for each block is the owner token. *The owner token may be either clean or dirty.*

- **Rule #2″ (Write Rule):** A component can write a block only if it holds all T tokens for that block and has valid data. *After writing the block, the writer sets the owner token to dirty.*

[6] Transient requests are described later in Section 5.2.

- **Rule #3″ (Read Rule):** A component can read a block only if it holds at least one token for that block and has valid data.
- **Rule #4″ (Data Transfer Rule):** If a coherence message contains *a dirty* owner token, it must contain data.
- **Rule #5″ (Valid-Data Bit Rule):** A component sets its valid-data bit for a block when a message arrives with data and at least one token. A component clears the valid-data bit when it no longer holds any tokens. *The home memory sets the valid-data bit whenever it receives a clean owner token, even if the message does not contain data.*
- **Rule #6″ (Clean Rule):** *Whenever the memory receives the owner token, the memory sets the owner token to clean.*

These six rules are designed to enforce the global invariant that if the owner token is clean, the value in memory will match the current single value of the block (Section 3.1.2 introduced this concept of a "single valid value"). Thus, as long as the owner token is clean, the system knows the memory still holds the correct value. Once a processor dirties the owner token by writing the block, the system can no longer rely on the contents of memory, and the system's behavior (in essence) reverts to using the rules described in Section 3.1.4. When a processor sends the dirty owner token and data back to memory, the memory will set the owner token to clean (once again allowing for the owner token to be transfered without data). Holding valid data and all the tokens with a dirty owner token or clean owner token corresponds to the MODIFIED and EXCLUSIVE state, respectively. Holding valid data and less than all the tokens with a clean or dirty owner token corresponds to the OWNED and SHARED state, respectively.[7] As before, holding valid data with at least one token without the owner token corresponds to the SHARED state.

Rule #1″ allows the owner token to be dirty or clean. Rule #6″ instructs the memory to set the owner token to clean whenever it receives the owner token (*i.e.*, if the owner token is at the

[7]Holding a clean owner token (but not all tokens) is considered the SHARED state (not OWNED) since evicting a clean owner token does not require a dirty-writeback. However, a performance policy may dictate that a processor with a clean owner token will respond to requests for the block.

memory, it can only be clean). Since the memory initially holds all tokens, all owner tokens are initially clean. Rule #6″ allows the memory to transform the owner token from dirty to clean by receiving a message with the dirty owner token, writing the data into the memory (data must always accompany the dirty owner token), and setting the owner token to clean. Rule #2″ now specifies that the owner token becomes dirty whenever the block is written. Rule #3″ is unchanged from rule #3′. Rule #4″ only requires data be sent when the owner token is dirty, further reducing the number of cases in which data must be sent with tokens.

In Rule #5″, the valid-data bit acts as before except the home memory module sets the valid-data bit whenever it receives a clean owner token, even if the message does not contain data. The way in which processors set and clear the valid bit is unchanged. In combination with rule #4″, this change allows a processor holding all tokens for a clean block (*i.e.*, the owner token is still clean) to evict a block by returning the tokens to memory without the contents of the data block. When a clean owner token arrives at memory, the memory knows the block could not have been written since the memory last held tokens (otherwise the owner token would be dirty), and thus the content of memory is actually still valid.

One unfortunate side effect of these modified rules is that they do not prevent a processor from holding all tokens without valid data (possible only when the owner token is clean), which prevents the processor from reading, writing, or responding with the block. A processor can prevent this undesirable situation from persisting by sending its tokens back to the memory to reassociate the owner token with valid data. As this is a technically a starvation-avoidance issue and not an issue with safety, we discuss this further in Section 4.2.3.

With these new rules, when the memory holds all tokens for a block, the memory can send a message to a processor that contains a copy of the block, all the non-owner tokens, and a clean owner token (giving the receiver an EXCLUSIVE copy of the block). If the receiving processor only reads the block before it evicts the block, the processor needs to send only a small eviction notification message. If the processor writes the block, it can do so quickly, without the latency of collecting additional tokens. After a processor has written the block, any subsequent eviction will generate a dirty writeback—the same behavior as in a traditional coherence protocol.

Even with the above refinement of the rules, Token Coherence still requires eviction notifications when evicting clean blocks, unlike many traditional protocols that allow silent eviction. These non-silent evictions do create additional interconnect and memory traffic overheads. The magnitude of these overheads and additional ways of reducing them are further discussed in Section 3.2.

3.1.6 Supporting Upgrade Requests, Special-Purpose Requests, and I/O

Token Coherence also supports other types of common coherence operations.

Upgrade requests. As currently described, the token counting rules do not support upgrade requests. Although these rules can be modified to included upgrade request (see below), we believe Token Coherence has little to gain from upgrade requests (as mentioned in Section 2.2.5). In Token Coherence, transitions from OWNED to MODIFIED already avoid transferring data in most cases, and we find that SHARED to MODIFIED transitions are rare, especially when using the migratory sharing optimization (described in Section 2.2.4).

The token counting rules can be modified to support upgrade requests by allowing the dirty owner token to be transfered without data. The cost of this flexibility is requiring such a transfer to be explicitly acknowledged to ensure the requesting processor still held a valid copy of the block when it received the owner token. If the recipient of the owner token does not have a valid copy of the block, the recipient must return the owner token to its sender (*e.g.*, by returning the owner token in a negative acknowledgment message). While waiting for the acknowledgment message to arrive, the responding processor must continue to hold a valid copy of the data (to ensure at least one system component maintains a valid copy of the block).

Special-purpose requests. Token Coherence can also support operations generated by special memory instructions. One important example is the special instructions used to reduce traffic by not fetching blocks that will be entirely overwritten (*e.g.*, the destination in a memory copy or the `bzero()` routine). Examples include Alpha's "Write Hint 64" (`WH64`), PowerPC's "Data Cache Block Clear to Zero" (`dcbz`) and "Data Cache Block Allocate" (`dcba`), and SPARC's block store instructions. The exact semantics of these instructions vary between instruction set architectures,

but an efficient implementation of these instructions generally requires the coherence protocol to allocate a writable copy of a cache block (often with undefined contents) into a processor's cache without generating a data transfer from memory.

In Token Coherence, a memory module can respond to such a coherence operation by sending all the tokens and clean owner token for a block—but not the content of the data block—to the initiator. The recipient processor will thus have permission to write the block, but the data-valid bit will not be set. When the processor retires the instruction that writes or clears the cache block, it simply sets the data-valid bit and changes the owner token from clean to dirty. This approach allows Token Coherence to avoid the interconnect and memory traffic that would result from a normal write request. If the home memory module does not have all the tokens, the system must reinterpret or reissue the request as a traditional write request operation.

I/O in Token Coherence. Token Coherence supports DMA (direct memory access) I/O (input/output) by allowing I/O devices (or their corresponding DMA controllers) to hold and manipulate tokens, allowing them to coherently read and write the shared address space. I/O devices can also use the techniques described above for obtaining a write-only copy of a block without a data transfer. Alternatively, I/O devices can rely on a memory controller, processor, or processor's cache controller to perform read and writes on its behalf. Memory mapped (uncacheable) I/O access is not affected by the choice of coherence protocol, and thus its implementation is unaffected by Token Coherence.

3.1.7 Reliability of Token Coherence

Like most other coherence protocols, Token Coherence depends on reliable message delivery and fault-free logic to behave correctly. The system may provide these reliability guarantees directly (*e.g.*, with highly-reliable links) or indirectly (*e.g.*, using error detection and message retransmission to recover from corrupted messages).

Although explicitly counting tokens may seem more fragile or more susceptible to errors due to lost messages or corrupted state, it isn't any more error prone than other coherence protocols.

For example, a lost invalidation message in a snooping protocol or a lost acknowledgment message in a directory protocol with both cause either inconsistency or prevent forward progress.

In some ways, Token Coherence may be more resilient to faults that cause lost messages. For example, the system may drop any message that is part of the performance policy (discussed in Chapter 5) without affecting system correctness. Any token messages that are dropped will prevent processors from writing the block, and will thus likely cause deadlock or livelock. However, the system will act in a failsafe manner by not allowing processors to read or write data incoherently. A deeper discussion of the reliability issues of both traditional coherence protocols and Token Coherence is beyond the scope of this work.

3.1.8 Opportunities Enabled by Token Counting

By not restricting when and to which processors tokens must be sent, token counting grants significant freedom while still ensuring safety. We exploit this freedom to create many performance policies (Chapter 5). Unfortunately, the cost of this freedom is the need for a separate starvation-prevention mechanism (Chapter 4). Before discussing these two aspects of Token Coherence, we first discuss several overheads that arise when implementing token counting.

3.2 Token Storage and Manipulation Overheads

This section discusses several real-world implementation overheads of Token Coherence; as such, this section is not strictly necessary for an understanding of the conceptual aspects of Token Coherence. Thus, the reader may continue reading with Chapter 4 without a conceptual disconnect. However, this section's material is helpful for understanding (1) some of the implementation issues of Token Coherence and (2) the evaluation of Token Coherence in later chapters.

By using tokens to explicitly track coherence permissions, Token Coherence incurs additional overheads not found in traditional coherence protocols. Fortunately, these overheads are either minor or can be mitigated using a variety of methods. This section describes, approximates, and discusses how to reduce the impact of each of the four main sources of token-counting overhead: holding tokens in caches, transferring tokens in messages, sending additional messages due to non-

silent evictions, and holding tokens in memory. These overheads are generally small because the size of the token state is only a small fraction of the size of a cache block.

3.2.1 Token Storage in Caches

Although Token Coherence requires extra state in caches, the state is small compared to the size of a datablock, and thus the overhead is low. Token Coherence requires that a processor maintain—for each block in its cache(s)—a token count, an owner-token bit, a clean/dirty owner bit, and a valid-data bit. The most straightforward approach for encoding this state is to add these bits to the cache tags, much like other coherence protocols add bits to the cache tags to encode the MOESI state of the block or a $\lceil \log_2 n \rceil$-bit pointer used in cache-based directory schemes (*e.g.*, SCI [46], or Sequent's NUMA-Q [76]). Since Token Coherence only counts tokens (*i.e.*, it does not need to track which processors hold tokens), this information can be stored in $3 + \lceil \log_2 T \rceil$ bits (valid-data bit, owner-token bit, clean/dirty bit, and non-owner token count). Encoding a maximum of 64 tokens—enough for a 32- or 64-processor system—requires only 9 bits. These extra bits result in less than a 2% increase in the total number of bits in the cache (tags and data) for 64-byte cache blocks (for 128-byte blocks the overhead is reduced to less than 1%). Thus, this overhead appears to be a reasonable cost for the benefits that token coherence provides.

To ensure these additional bits do not increase the cache access latency, these bits can be removed from the critical path of cache access. First, to reduce the number of bits that the processor must examine on a cache hit, the cache tags could introduce a few additional bits for an MOSEI-like encoding of the permissions derived from the token count. Second, to further reduce the token bits from impacting the critical path, the tokens could be stored in a separate array that parallels the tag array.

3.2.2 Transferring Tokens in Messages

Messages that transfer tokens between system components must encode token-counting information. This information is similar to that information stored in caches: a token count, an owner-

token bit, and a clean/dirty owner bit. Tokens travel in two types of messages: data messages and non-data messages.

- The overhead for data messages is low for the same reason that the overheads of holding tokens in the cache is low: tokens are much smaller than data blocks. As a result, data messages increases in size by less then 2%.

- Token Coherence allows for transfer of messages without a dirty owner token to omit data. As these messages are significantly smaller than messages that carry data, the overheads of encoding tokens is also larger. These data-less messages are used (1) as eviction notifications when evicting non-dirty blocks or (2) as invalidation acknowledgment messages (much like a directory protocol). For our system assumptions, these data-less messages are approximately eight bytes in size. One byte of token-related state results in a 13% overhead.

  Fortunately, these data-less messages are almost always paired with a previous data message. In most cases, a processor that issues a data-less token message received those tokens in a previous message *that contained data*. This occurs when data-less token messages are sent as invalidation acknowledgments or as eviction notification messages. When these non-data messages are paired with data messages, the amortized overhead is approximately 4% (or twice the overhead of just a single data message).

  Although data-less eviction messages are common, data-less messages used as invalidation acknowledgments are infrequent because of (1) the small number readers between writes [31, 44, 75, 79, 115], and (2) the migratory sharing optimization we employ (described in Section 2.2.4). The other main source of data-less token messages is Token Coherence's non-silent evictions; we discuss this overhead separately in Section 3.2.3.

The overhead due to the increase in message size is not modeled in our simulations. In our simulations, all message sizes are multiples of our 4-byte link width. Since a 4-byte message is too small (due to a large physical address space) and an 8-byte message can encode all required information, the size of all of our non-data messages in all our protocols is 8 bytes. Thus, in

our simulations Token Coherence messages are no larger than messages used by the other protocols. However, our experimental results do provide a breakdown of distribution of message sizes, allowing for an estimate of this overhead.

3.2.3 Non-Silent Evictions Overheads

Although the increase in message size is small, Token Coherence also increases the number of messages sent by requiring non-silent evictions of clean blocks. Similar to our analysis in the previous section, these 8-byte eviction messages only occur when a data message was previously sent to the processor. The worst-case behavior for eviction notifications is a processor reading many blocks without writing them. This pattern would generate an outgoing 8-byte eviction notification for each incoming 72-byte data block (64 bytes of data plus an 8-byte header), resulting in an 11% increase in traffic (in bytes). Of course most real workloads also write to data, and the resulting dirty writebacks reduce the relative overhead. For example, a workload that dirtied half of the blocks it accessed would incur only a 3% traffic overhead (two 72-byte data responses and a 72-byte dirty writeback versus these same messages plus an 8-byte eviction notification). This worst-case estimate does not include any request traffic, which further reduces the overhead as a percent of bytes of interconnect traffic.

All eviction-notification messages are explicitly modeled in our simulations (described later in Chapter 6), and our experiments quantify this overhead by explicitly tracking the additional traffic caused by non-silent evictions. Our results indicate that this overhead is small (for our workloads and assumptions). These non-silent evictions also result in an increase the number of messages processed by the memory controller and increase the frequency with which the memory must update its token count; we discuss this overhead next (as we discussion the overheads of holding tokens in memory).

3.2.4 Token Storage in Memory

The final—and perhaps most serious—overhead of token counting is holding and manipulating the tokens for each block in memory. Although the relative overhead is the same as in caches (less

than 2% for our assumptions), the sheer number of memory block creates a potentially serious issue. For example, with 9 bits of token state per 64-byte block, a memory module with 4GBs of DRAM would need an additional 72MBs for token storage. Not only is the amount of state large, but the system must provide high-bandwidth access to the state because it is both read and written for each update of the token count (potentially each coherence request).

Fortunately, this problem is not unique to Token Coherence, but it is the same problem encountered by directory protocols (for holding the directory state that tracks the sharers and/or owner for each block). In terms of the amount of state, the magnitude of the problem is actually smaller for Token Coherence than for some directory protocols because the amount state required for Token Coherence is only a token count, and not a full encoding of which processors are sharing the block. In terms of required bandwidth, Token Coherence requires more frequent manipulations of the information due to non-silent evictions.

The next few pages describe four standard alternatives used by directory protocols to mitigate this problem and discuss their application to Token Coherence. In the discussion, the term *per-block state* will generically refer to both the token-counting state (in Token Coherence) and the sharers/owner directory information (in a directory protocol).

Approach#1: Dedicated storage. The most straightforward approach to holding per-block state is to use a large dedicated RAM (either SRAM or DRAM) at each memory module to store the state (as adopted by many directory protocol systems in the 1990s [72, 74, 75, 76]). This approach is simple, but it adds to the total system cost. Alternatively, the per-block state can be stored in memory by reserving a range of physical memory to be used for per-block state (making this range unavailable to the software). This approach does have the advantage of not increasing the number of discrete system components, but it also reduces the amount of memory available to software, and the operating system needs to recognize which addresses are unavailable to it.

Approach#2: Redefining the ECC bits. Encoding per-block state in the normal system DRAM by redefining the memory's ECC[8] (error correction code) bits is an approach recently

[8]Chen and Hsiao [28] and Peterson and Weldon [99] provide an overview of various error correcting codes.

adopted by several systems (*e.g.*, the S3mp [96], Alpha 21364 [91], UltraSparc III [58], and Piranha [39]). In contemporary ECC DRAM, 72 bits are used to store 64 bits of data, and the memory controller uses the remaining 8 bits to implement a SECDED (single error correct, double error detect) code. If the memory controller is changed to calculate ECC more coarsely—*e.g.*, on 128/144-bit blocks—not all 16 bits that are available for ECC are required to implement a SECDED code. For example, Sun's UltraSPARC III systems use this technique to gain seven "unused" bits per 128/144-bit block (for a total of 28 bits per 64-byte coherence block) to hold the directory state for its coherence protocol [58].

This ECC-based approach has one main advantage (it is relative inexpensive), a few minor disadvantages (it requires ECC DRAM, complicates the ECC circuitry, and weakens the ECC code), and one serious disadvantage (it generates extra memory DRAM traffic). The main advantage of this approach is that it requires no additional state and only requires a localized and well-understood change to the memory controller. One possible disadvantage is that currently not all systems use ECC memory (which is more expensive than non-ECC memory). However, almost all servers currently use ECC memory, and most future systems will use ECC memory due to the increasing transient bit fault rates as DRAM becomes more dense [60, 87, 88]. To support this coarser ECC, the ECC calculation logic must concurrently consider a larger number of bits, which increases the size and complexity of the ECC circuitry. This technique also weakens the ECC code (*e.g.*, bit flips in each of two consecutive 64-bit blocks would have been correctable using standard ECC but might not be correctable using coarser ECC). This disadvantage is not too serious in current systems due to (1) the presumed independence of transient faults, (2) the careful physical layout of DRAM cells, and (3) memory controllers that proactively scan (or "scrub") the memory looking for and correcting bit errors. Systems that use stronger codes than SECDED (*e.g.*, correcting four consecutive bits to protect data against a "chipkill" fault of a 4-bit DRAM bank [61]) many not free up as many bits when using larger ECC blocks, but these systems should still be able to use this general technique to encode per-block state.

The most serious disadvantage of this approach is the extra memory (DRAM) traffic it introduces. Instead of just performing a read or write of main memory DRAM, updating the per-block

state in memory transforms many memory reads or writes into read-modify-write actions. Gharachorloo *et al.* [39] studied this issue in detail for a directory protocol and identified situations in which memory controllers can reduce the amount of extra traffic by limiting the number of read-modify-write operations.[9] Unfortunately, the extra memory traffic can be especially prevalent due to the non-silent evictions required by Token Coherence. An alternative approach (described next) uses an on-chip cache of per-block state to reduce both the access latency and the bandwidth demands placed on the memory.

Approach#3: Directory/token cache. A SRAM *token cache* or *directory cache* behaves much like a traditional cache, except that (1) it is logically part of the memory controller, and (2) it caches only per-block state for blocks in the local memory. By keeping the most recently used per-block state in on-chip SRAM, such a cache provides the memory controller with high-bandwidth, low-latency access to the per-block state. When the desired information is not found in the cache, the memory controller retrieves the per-block information either stored in dedicated storage or encoded in the memory's ECC bits. The memory controller can either access this cache and dedicated storage in parallel (lower latency, but higher traffic) or in series (lower traffic, but higher latency), or it can adaptively or predictively choose between these options for each request.

The size of the directory cache depends on the goal of the directory cache. As a small number of commonly requested "hot blocks" account for a larger fraction of the misses in many workloads [79], even a small directory cache (hundreds of entries) should exhibit a significant hit rate. Alternatively, a sufficiently large cache of per-block state—large enough to capture all the blocks from its memory module that are cached by processors—can approach the hit rate of a infinite cache (at the cost of a non-trivial amount of die area). For example, if the number of memory modules is the same as the number of processors (as they are for some highly integrated systems like AMD's Hammer [9, 121] and Alpha 21364 [47, 91]), the average number of blocks cached per memory module is the same as the number of blocks in each processor's cache. If each processor had a 4-way set-associative 2MB second-level cache and the system had a 38-bit physical address space,

[9] For example, in Token Coherence a dirty writeback message with *all* tokens knows the memory holds no tokens, and thus the memory can avoid reading the token count before updating the per-block state.

a token cache would require a 19-bit tag and 9-bits of token state (64 tokens and the associated valid/owner/dirty bits). A cache with (1) the same number of entries as the second level cache and (2) a 28-bit overhead per block results in a near-perfect cache that is approximately 5% of the size of the second level cache.

A cache of per-block state magnifies bandwidth and reduces latency of per-block state access. However, it has a couple of disadvantages: (1) it adds complexity to the system (yet another widget to design and verify) and (2) it consumes a modest amount of on-chip resources. Although such caches are not a new idea, they have not yet achieved widespread adoption in real systems or even in academic studies.

Approach#4: Inclusive directory/token caching. The directory/token caching approach assumes that when per-block state is evicted from the cache it can be written back to the main directory or token storage (*i.e.*, dedicated storage or encoded in the ECC bits). An alternative approach is to enforce *inclusion* between the per-block cache and systems caches. Enforcing inclusion allows a cache of per-block state that does not rely upon a larger in-memory per-block state storage. In this approach, when an entry is not found in the cache it is known to be in the default or uncached state (*e.g.*, all tokens in memory for Token Coherence or that no processors are caching the block for a directory protocol). The memory controllers enforce inclusion using *recall messages*. We first explain the operation of inclusive directory cache; second, we'll extend the approach to include Token Coherence.

In a standard directory protocol that uses an inclusive directory cache, a directory controller desires to evict an entry from its directory cache, it sends a recall message to any processor that might be caching the block (some limited-pointer directory schemes such as Dir_1NB [8] use a similar mechanism). Each processor acknowledges the recall message by responding with either an acknowledgment message (for those processors in the, SHARED, and EXCLUSIVE states) or a data writeback message (for those processors in MODIFIED and OWNED). When the memory module has received the required responses it knows that no processor in the system could be caching the block, allowing it to deallocate the entry (freeing the cache frame to hold another block's directory state). To avoid affecting the critical path of a miss, a memory controller can

reserve a few extra entries to allow the recall operation to proceed in parallel with handling the request (much as a processor's cache handles evictions in parallel with misses).

Extending this approach to Token Coherence is straightforward. The memory controller using the same recall messages for evicting entries from the cache of per-block state. Processors respond to these recall messages by sending any tokens they have back to the memory (using the processor's existing eviction mechanism); processors holding no tokens do not respond. When the memory receives all a block's tokens it can safely deallocate the block's entry. In essence, the memory issues a request to gather all tokens (much like a processor's might issue a write request). To ensure that the memory eventually receives all the tokens, it may need to invoke the correctness substrate's starvation prevention mechanism (Chapter 4).

Cache frames may also be freed when the entry reverts to the default setting (*e.g.*, when a processor writes back a MODIFIED copy of a block or the last sharer/token holder sends an eviction notification message to the directory).[10] This proactive deallocation of entries allows protocols that support eviction notification—required in Token Coherence, but optional in many directory protocols—to significantly reduce the number of recall operations. Without eviction notifications a block that was read and silently replaced will occupy an entry until the memory controller recalls the block or another processor later writes and evicts the block. The combination of a system that (1) uses eviction notifications, (2) has a sufficiently large cache of per-block state, and (3) is running a workload that exhibits a normal distribution of access to blocks from each memory module should generate a negligible number of recall messages.

Of course, like the other approaches, this technique has advantages and disadvantages. The main advantage of this approach is that it eliminates all need to store per-block state in DRAM, avoiding any additional off-chip DRAM traffic. However, this approach has three significant disadvantages:

- First, implementing recall messages introduces extra protocol complexity—especially when considering deadlock issues. To ensure that recall operations will not create circular depen-

[10]This early deallocation of cache frame entries can also be used to increase the effectiveness of caches that do not use recall operations.

dencies among protocol messages, adding recalls to a directory protocol requires both (1) additional virtual networks (*e.g.*, additional virtual networks for recall messages and recall responses) and (2) careful examination of transient states that block or queue messages.

Extending Token Coherence to support recall operations may be easier than some other coherence protocols due to the flexible nature of the correctness substrate. For example, the memory controller could use a modified version of Token Coherence's existing persistent request mechanism (described in Chapter 4) to implement the recall mechanism. Even with the reuse of this mechanism, supporting the recall operation still adds unwanted subtlety to the coherence protocol.

- Second, the cache of per-block state must be large enough (contain a similar number of blocks as the second-level processor caches) to prevent excessive recall messages from prematurely evict blocks from processor caches. As described previously, a straight-forward implementation of such a cache results in a 5% increase in the number of bits as compared with the number of bits in the second level cache.[11]

- Third, memory access patterns that exhibit a non-uniform distribution of accesses among the many memory controllers may "thrash" a single memory controllers cache of per-block state, greatly limiting the effective size of processor caches in pathological cases (*i.e.*, all processor are accessing blocks from a single memory module).

Summary for token storage in memory. While none of the four approaches are ideal, using a substantial token cache backed up by storage in main memory's ECC has moderate cost and should generate little additional memory traffic. However, the possible methods for encoding per-block state covers a large design space. In addition, this design choice effects not only Token Coherence, but also directory protocols and snooping protocols that exploits per-block state. Because (1)

[11] A designer can reduce the number of bits in a token cache by encoding only certain token counts (such as all tokens or no tokens) or by using sector-cache techniques (also know as sub-blocking). For example, using a 4-subblocks per cache tag reduces size overhead by almost half. However, sector caches complicate cache evictions and are susceptible to access patterns that lack spatial locality, reducing the effective size of the cache.

much of this design space is tangential to the core ideas of Token Coherence and (2) we want to avoid skewing our experimental results based on this implementation choice, our experiments use a simplistic model of dedicated per-block storage with unbounded directory bandwidth for all of the protocols we evaluate.

3.2.5 Overhead Summary

Overall, the overheads of maintaining tokens counts are modest. To support a moderate number of tokens (*e.g.*, 64 tokens) fewer than ten bits are required per 64-bytes or 128-byte data block, resulting in a approximately a 1-2% increase in the number of bits in the processor caches. The same small overhead is added to the size of coherence message that carry tokens and data; data-less token messages have higher overheads, but are often paired with larger data messages. Non-silent evictions cause additional messages, but the additional interconnect traffic (in terms of bytes) only increases by 3% for a workload with equal numbers of clean and dirty evictions. The memory controller can use techniques developed for directory protocols to both cache its token and encode its tokens in the DRAM's ECC bits, providing high-bandwidth access to the memory's per-block state with reasonable cost.

Chapter 4

Starvation Freedom via Persistent Requests

The previous chapter described the correctness substrate's use of token counting to ensure safety; however, safety is not sufficient for correct operation. In addition to ensuring safety (do no harm), the correctness substrate must also prevent starvation (do some good). A starvation-free system must ensure that every read and write attempt eventually succeeds.

This chapter extends the correctness substrate by adding a mechanism to explicitly prevent starvation. Since the performance policy (described in Chapter 5) should enable processors to quickly complete their requests in the common case, this new mechanism should be rarely invoked (*i.e.*, only a couple percent of misses should need to invoke persistent requests). A processor only invokes the persistent request mechanism when it suspects it may be starving (*e.g.*, it has failed to complete a cache miss within a timeout period). This approach frees the performance policy to focus on making the common case fast, allowing the starvation-prevention mechanism to handle the rare cases that can lead to starvation.

The flexibility provided by Token Coherence is one of its key advantages, but this same flexibility is directly responsible for complicating starvation prevention. To prevent starvation the correctness substrate must ensure that all attempts to read or write a block will eventually succeed. This guarantee must hold in all cases not explicitly disallowed by the token counting rules. For example, tokens can be delayed arbitrarily in transit, tokens can "ping-pong" back and forth between processors, or many processors may wish to access the same block at the same time. In many coherence protocols once a request is *ordered*, it is guaranteed to succeed. These traditional protocols rely on a total order of requests provided by the interconnect (snooping protocols) or a

per-block ordering point at the home memory (directory protocols). These methods of ordering requests introduce penalties that Token Coherence seeks to avoid. As token counting does not use a point-of-ordering of requests, Token Coherence must adopt a different approach to preventing starvation.

Although many methods of preventing starvation are possible in this context, this dissertation focuses on using *persistent requests*. A processor issues a persistent request when it detects it may be starving (by noticing a lack of recent progress). The substrate arbitrates among the outstanding persistent requests to determine the current *active request* for each block. The substrate sends the persistent requests to all system components. These components must both remember all active persistent requests and redirect their tokens—those tokens currently present and those to be received in the future—to the requesting processor until the requester has *deactivated* its persistent request. The initiator deactivates its requests when it has received sufficient tokens to perform a memory operation (*e.g.*, a load or store instruction). Starvation-free arbitration and the persistent nature of these requests enable the substrate to eventually find and redirect all the required tokens, even in pathological cases.

To provide several reasonable design options, this chapter presents descriptions of multiple approaches for preventing starvation. We hope that the description of these techniques will inspire other researchers to develop alternative approaches for preventing starvation in Token Coherence. The chapter begins by introducing a simplified persistent request mechanism that uses a centralized arbiter (Section 4.1) and by outlining how such an approach prevents starvation (Section 4.2). The chapter continues by describing an approach with multiple arbiters to increase throughput (Section 4.3) and an enhancement to improve the handling of read requests (Section 4.4). The penultimate section of this chapter describes a distributed arbitration technique for persistent requests that has better behavior in worst-case situations (Section 4.5). The final section briefly discusses the possibility of improving the scalability of these approaches (Section 4.6).

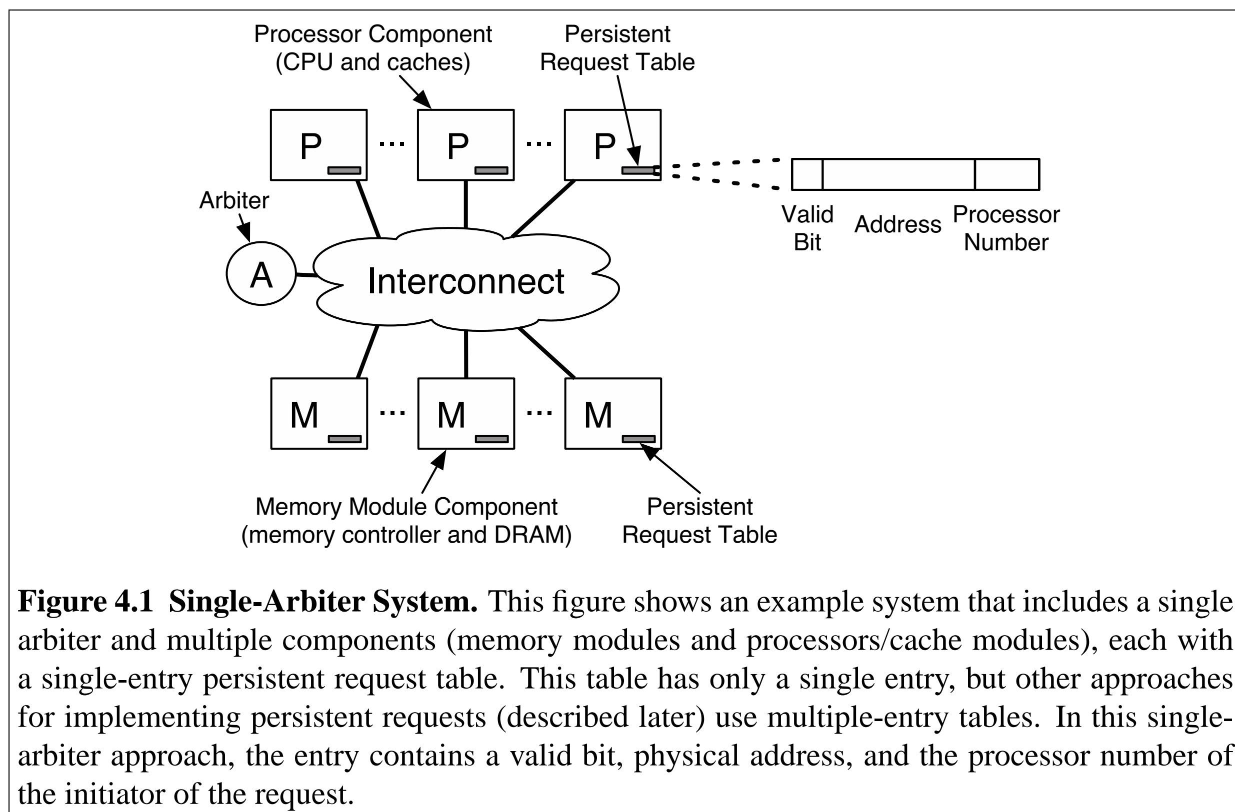


Figure 4.1 Single-Arbiter System. This figure shows an example system that includes a single arbiter and multiple components (memory modules and processors/cache modules), each with a single-entry persistent request table. This table has only a single entry, but other approaches for implementing persistent requests (described later) use multiple-entry tables. In this single-arbiter approach, the entry contains a valid bit, physical address, and the processor number of the initiator of the request.

4.1 Centralized-Arbitration Persistent Requests

This section begins the discussion of persistent requests by describing a correct, but simplified, implementation of persistent requests using a single centralized *arbiter*. The substrate directs persistent requests to the arbiter, queuing multiple requests in a dedicated virtual network[1] or at the arbiter itself. The arbiter state machine *activates* a single request by informing all processors and the block's home memory module. These components each remember the active persistent request using a hardware *persistent request table*. Figure 4.1 shows a system with a single arbiter and the contents of the persistent request table found at each component. In this centralized-arbitration approach, each persistent request table has a single entry with three fields (a valid bit, an address, and a processor number). We introduce the notion of a persistent request table in preparation for

[1]The virtual network must provide starvation-free message delivery.

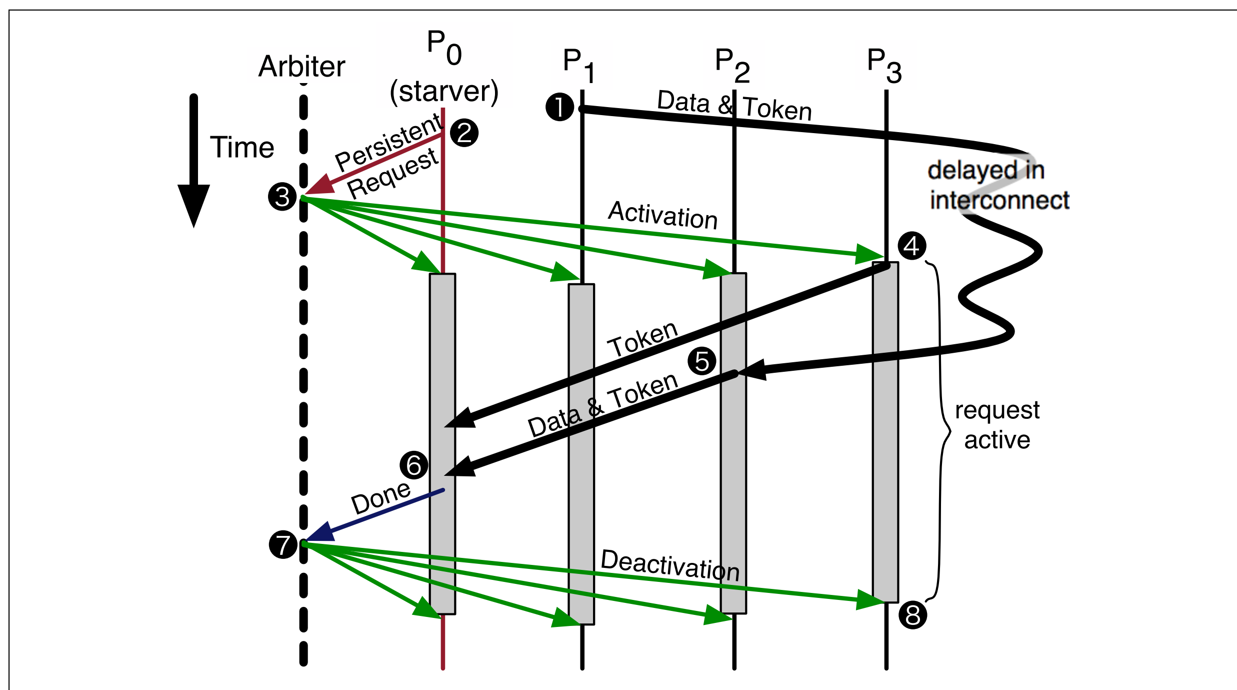


Figure 4.2 Arbiter-based Persistent Request Example. This figure illustrates the operation of the arbiter-based persistent request mechanism. The dotted vertical line represents the arbiter and the other lines each represent a processor; to simplify the example memory modules are not explicitly shown, but they act similarly. Time flows from top to bottom, and each angled line represents a coherence message. At time ❶, P_1 sends a message with data and a token to P_2, but the interconnect significantly delays the message. Later, at time ❷, P_0 detects it may be starving and sends a persistent request to the arbiter. The arbiter receives the message (at time ❸) and sends activation messages to all processors (and the home memory, not shown). At time ❹ each processor receives the persistent request activation and records the persistent request in its persistent request table. After receiving the activation message, each processor will send to P_0 any tokens and data it currently has (as P_3 does) or that arrive later (as P_2 does at time ❺). When P_0 has received data and both system tokens (at time ❻), P_0 performs a memory operation and sends a completion message to inform the arbiter to deactivate the request (which it does at time ❼). When each processor receives a deactivation message (at time ❽), it deactivates the request by clearing its persistent request table (reverting the system back to normal behavior).

other approaches to implementing persistent requests that use multiple-entry tables (described later in this chapter).

While a persistent request is active for a block, each component must forward all the block's tokens (and data, if it has the owner token and valid data) to the requester. The component will also

forward tokens (and data) that arrive later, because the request persists until the requester explicitly deactivates it. The initiator of the active persistent request[2] must not give away any of its tokens, except when it holds all tokens without valid data. As described in Section 3.1.5, a processor can hold all tokens without valid data when a clean owner token is transfered without data. When this situation occurs, the processor sends the tokens to the memory to reassociate the tokens with valid data (via Rule #5″ in Section 3.1.5). Once the requester has (1) received all tokens, (2) received valid data, and (3) observed the activation of its own persistent request, it sends a message to the arbiter to *deactivate* its request. The arbiter deactivates the request by informing all nodes, which delete the entry from their tables. Figure 4.2 presents an example of arbiter-based persistent request activation and deactivation.

As described, this approach requires that the system prevent the reordering of activation and deactivation messages. Such reordering would allow starvation situations in which different processors would permanently disagree which processor is the currently active processor. The system can prevent reordering of these messages using point-to-point ordering in the interconnect or by using rounds of explicit acknowledgments; these two alternatives and other approaches are further discussed later in Section 4.7.

4.2 Showing That Persistent Requests Can Prevent Starvation

To show that this mechanism prevents starvation, we argue that (1) the system will eventually deliver all messages, (2) each starving processor will receive all tokens, (3) the starving processor will receive valid data (with all tokens), and (4) the mechanism deactivates the request in all cases (to enable the next persistent request activation).

4.2.1 Deadlock-Free Message Delivery

First, because of a interconnect that is reliable, deadlock-free, and livelock-free, the system eventually delivers all messages. Such an interconnect may require employing well-understood

[2] The initiating processor knows its request is active because it also receives the activation message.

techniques [36] such as additional virtual networks (to avoid protocol deadlocks), each with multiple virtual channels (to avoid routing deadlocks).

To prevent deadlocks, this implementation of persistent requests conservatively uses six virtual networks: (1) persistent requests, (2) activations, (3) "to-processor" data/token messages, (4) "to-memory" data/token messages, (5) persistent request completions, and (6) deactivations. To avoid deadlock, the system uses these virtual networks only in increasing order (*i.e.*, a message on a higher virtual network cannot be blocked waiting for an equal or lower virtual network):

- Persistent requests (network #1) elicit only activation messages.
- An activation message (network #2) may elicit either no response or a "to-processor" data response.
- A message on the "to-processor" data/token virtual network (network #3) can elicit no response, a completion message, or a redirection of the data/tokens. If a persistent request is active for the block, the processor either (1) redirects the message to the initiator of the persistent request (if the "to-processor" virtual network is not blocked), or (2) redirects the message to the memory on the "to-memory" virtual network (otherwise). The processor also redirects to the memory any message for which its cache does not have an entry allocated (using the "to-memory" virtual network).
- Memory *always* sinks "to-memory" data messages (network #4) without blocking, even when a persistent request is active for the block. It does this by (1) writing the datablock to the memory (it has capacity to hold all blocks) and (2) periodically scanning the persistent request table for any blocks for which the memory holds tokens. Any tokens found during this scan are forwarded as specified by the persistent request mechanism.
- Completion messages (network #5) elicit only deactivation messages.
- Finally, deactivations (network #6) can always be sunk because they never generate messages.

Thus, six virtual networks are sufficient, but perhaps not necessary. Although using six virtual networks simplified the discussion, fewer virtual networks may be sufficient (*e.g.*, if processors can defer sending data in response to persistent request activations).

4.2.2 Receiving All Tokens

Second, this implementation guarantees that each persistent request will eventually be invoked, be activated, and collect all tokens. A potentially starving processor will time out and invoke a persistent request. Assuming that (1) all previously issued persistent requests will eventually complete (assumed in an inductive fashion) and (2) the arbiter uses a starvation-free queuing policy (*e.g.*, first-in-first-out), all persistent requests will eventually reach the head of the arbiter's queue and the arbiter will activate it (*i.e.*, all persistent requests in the arbiter's queue ahead of this request must eventually deactivate to guarantee the request will eventually reach the front of the queue, causing the arbiter to activate it). Once the activation message has reached all components, each component will agree to forward all tokens it holds to the single starving processor (and they are not allowed to send tokens to any other processors). This property ensures that all tokens will eventually be redirected to the starving processor (since components will eventually redirect all in-flight tokens).

4.2.3 Receiving Valid Data

Third, a starving processor must obtain valid data (in addition to sufficient tokens). A starving processor can hold all tokens without valid data only when holding a clean owner token (a dirty owner token requires data with its transfer). When this situation occurs, the starving processor rectifies the situation by sending its tokens to the home memory module. Once the tokens and data are reunited at the memory, the owner token will always travel with data (at least while the persistent request is active), ensuring that the starving processor will receive data the next time it receives the owner token. Once a processor has sufficient tokens and valid data, it performs at least one non-speculative memory operation before deactivating the request.

4.2.4 Persistent Request Deactivation Requirement

Fourth, in addition to completing its own request, a processor must also deactivate its request and allow the arbiter to activate the next persistent request. The arbiter will forward the deactivation message to all processors and the home memory. This deactivation guarantees that each component will eventually reset all the state in the persistent request table for this request. After sending the deactivation, the arbiter can then activate the next request.[3]

4.2.5 Summary

Thus, this persistent request mechanism prevents starvation by using an arbiter to order persistent requests, and since it only affects system behavior during periods of possible starvation, it prevents starvation without unduly restricting the flexibility of Token Coherence.

4.3 Banked-Arbitration Persistent Request

A single centralized persistent request arbiter may become a bottleneck in large systems. Fortunately, a system can use multiple arbiters to scale persistent request throughput at the cost of larger persistent request tables. The previous approach did not explicitly require a single arbiter. As long as only one request is active per block and each component can remember all active persistent requests, a similar approach—one that uses multiple arbiters—will also prevent starvation. In this banked-arbitration approach, each arbiter is responsible for a fixed portion of the global address space, and the persistent request table at each component must contain one entry per arbiter. Each entry in this new multi-entry table encodes the same information as the previous single-entry table (a valid bit, a physical address, and a processor). Figure 4.3 illustrates a system with multiple arbiters and multi-entry persistent request tables.

Although the amount of state in each persistent request table grows linearly with the number of arbiters, in practice the table is still modest in size. For example, a system with an arbiter co-located

[3] As later described in Section 4.7, if the interconnect does not provide point-to-point ordering of activations and deactivation, a round of explicit acknowledgments is required before the next persistent request may be activated.

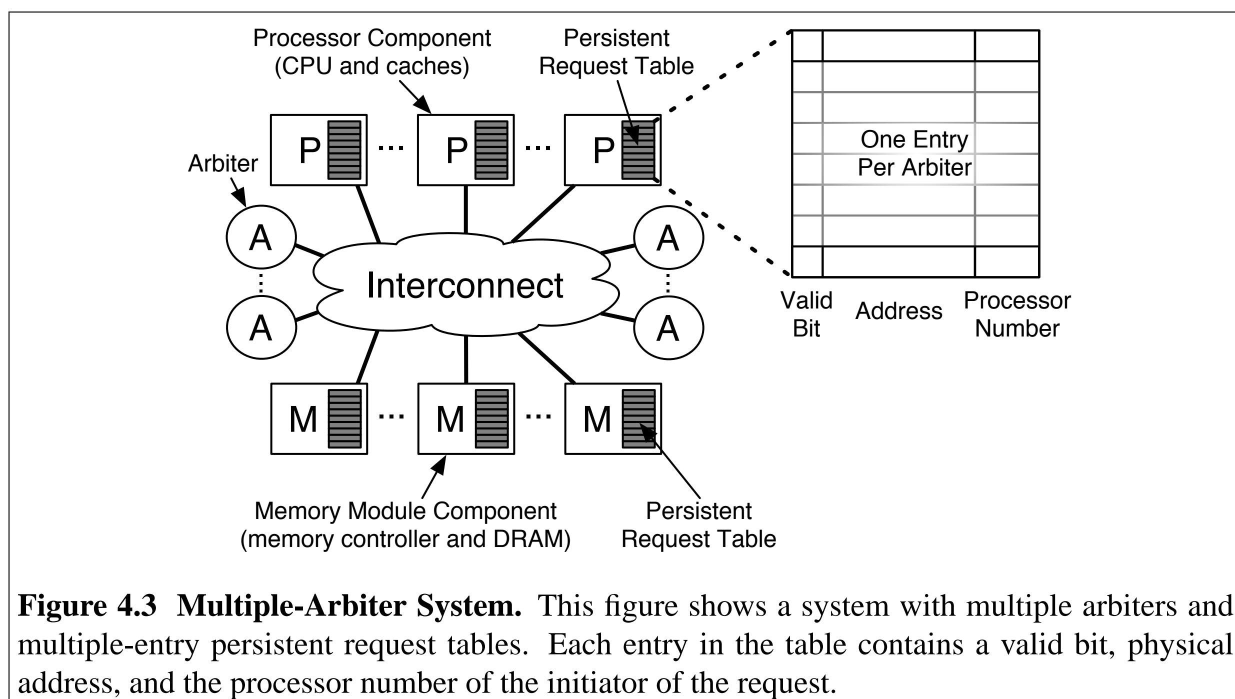


Figure 4.3 Multiple-Arbiter System. This figure shows a system with multiple arbiters and multiple-entry persistent request tables. Each entry in the table contains a valid bit, physical address, and the processor number of the initiator of the request.

within each of 64 processor/memory nodes requires only a 512-byte table per node.[4] Supporting 1024 nodes requires an 8KB table per node (0.3% of a 2MB on-chip second-level cache). The access bandwidth of this structure is also modest; the component accesses the table only when receiving persistent requests or messages with tokens (*i.e.*, not transient requests). In addition, since the table is indexed directly (*i.e.*, a table lookup does not require an associative search) using the memory controller number (assuming one arbiter per memory controller), a straightforward implementation of this structure should provide adequate bandwidth.

4.4 Introducing Persistent Read Requests

As described, the correctness substrate handles read and write persistent requests identically. While sufficient to prevent starvation, this approach has the undesirable side effect of stealing tokens away from all sharers to satisfy a persistent read request. This effect can result in a situation in which multiple processors trying to read a block devolve into continually issuing persistent

[4]This table may be shared among the node's many caches and memory controllers, or each structure may have its own table.

requests. To prevent this situation, a starving processor may use a *persistent read request* when it desires only a read-only copy of the block. In our implementation, a recipient of a persistent read request keeps at most one non-owner token, but the recipient must give up the rest of its tokens. In a system with at least as many tokens per block as processors, this approach guarantees both that (1) the starving processor will receive the owner token and (2) no processor will hoard many tokens at the expense of the others. Multiple processors trying to read the same block will eventually each hold one token (and valid data), avoiding the problem of continual persistent read requests. As in persistent write requests, a processor holding the (clean) owner token without valid data sends the owner token back to the memory. Because of the active persistent request, the memory will reassociate the owner token with valid data and promptly return it to the initiator of the persistent request.

4.5 Distributed-Arbitration Persistent Requests

The previously described approach to implementing persistent requests requires that processors first send persistent requests to an arbiter. This section describes a distributed-arbitration technique that reduces the latency of persistent requests by avoiding any (per-block) arbiter and allows direct communication of persistent requests between processors. This approach targets better worst-case behavior for highly-contended blocks (perhaps caused by performance-critical and highly-contended synchronization).

Unlike the previous approach that limits the number of active requests per arbiter, this approach relies on limiting the number of outstanding persistent requests (*e.g.*, only one) per processor. By limiting the number of persistent requests *per processor* to one, the processor number becomes the index for the persistent request table and the table has as many entries as the maximum number of processors in the system (as opposed to the maximum number of arbiters in the previous approach). Figure 4.4 shows a system that uses distributed arbitration.

When a processor detects possible starvation, it invokes a persistent request (with the constraint that each processor is allowed only a single outstanding persistent request at a time). To invoke

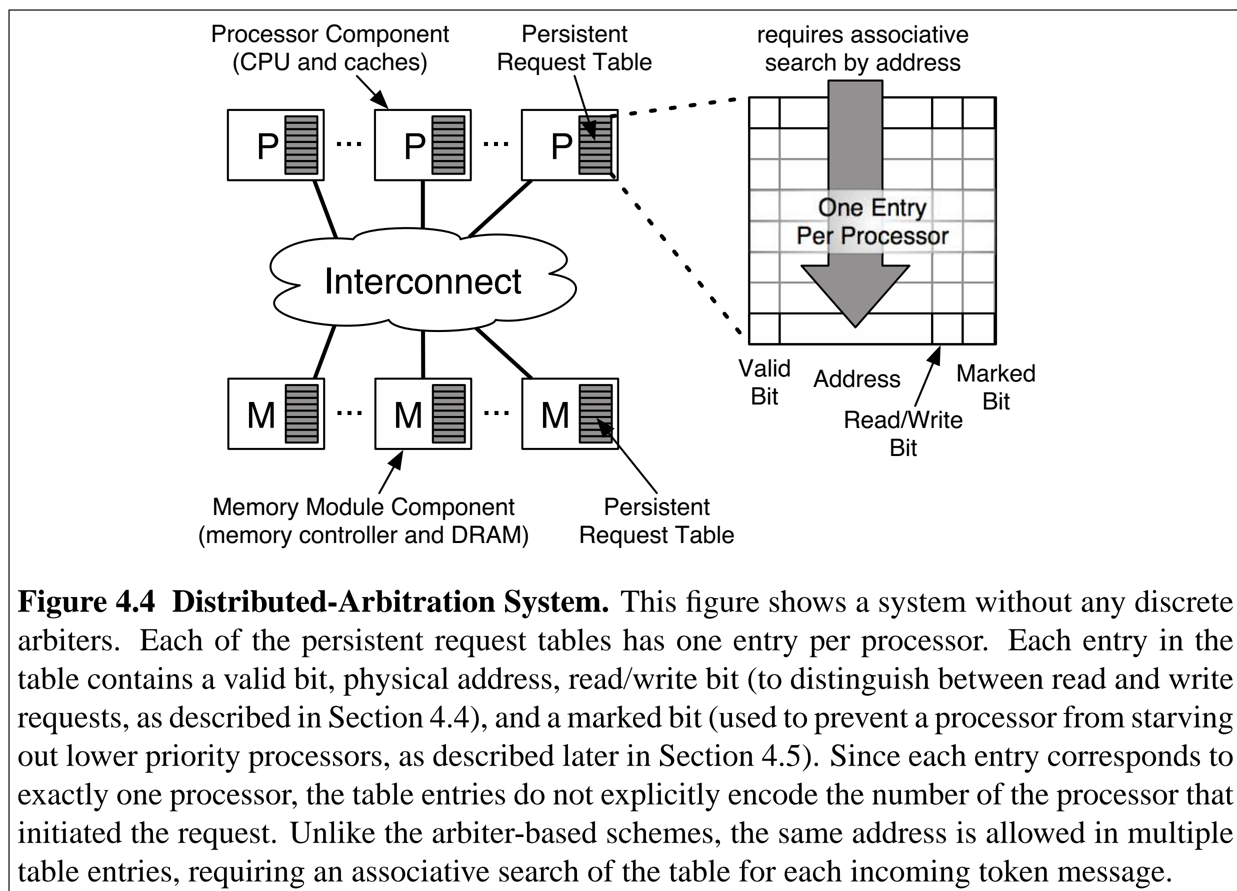


Figure 4.4 Distributed-Arbitration System. This figure shows a system without any discrete arbiters. Each of the persistent request tables has one entry per processor. Each entry in the table contains a valid bit, physical address, read/write bit (to distinguish between read and write requests, as described in Section 4.4), and a marked bit (used to prevent a processor from starving out lower priority processors, as described later in Section 4.5). Since each entry corresponds to exactly one processor, the table entries do not explicitly encode the number of the processor that initiated the request. Unlike the arbiter-based schemes, the same address is allowed in multiple table entries, requiring an associative search of the table for each incoming token message.

the request, the processor sends a persistent request directly to all processors and the home memory module. Each of these components records the requests in its local persistent request table. This persistent request table is indexed by processor number (since each processor is allowed only a single outstanding persistent request, the table needs as many entries as the maximum number of processors in the system), and each entry has an address, a read/write request type bit, a valid bit, and a marked bit (used later). Since this table can now have multiple entries that contain the same address, the entry for the processor with the lowest number is appointed the active persistent request. Similar to the previous approaches, all components send tokens (and data) to the initiator of the active persistent request. When a processor completes its request, it sends a deactivation message directly to all processors, which clears the corresponding entry in the table. This deactivation implicitly activates the next persistent request (the request with the next lowest processor number). Like the previous approach, the system must not reorder activation and deactivation mes-

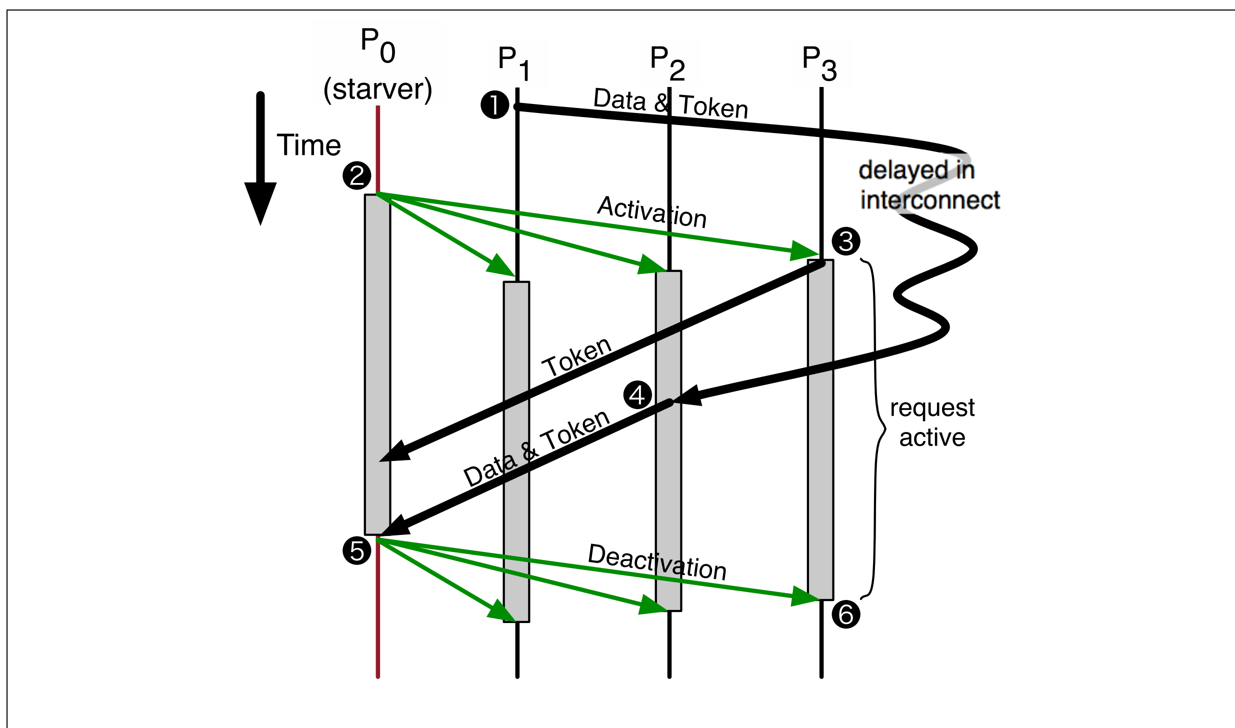


Figure 4.5 Distributed Persistent Request Example. This figure illustrates the operation of the distributed-arbitration persistent request mechanism using the same example as Figure 4.2. Similarly to Figure 4.2, the vertical lines represent processors, time flows from top to bottom, and each angled line represents a coherence message. At time ❶, P_1 sends a message with data and a token to P_2, but the interconnect significantly delays the message. Later, at time ❷, P_0 detects it may be starving and sends a persistent write request directly to all other processors (and the home memory, not shown). At time ❸ each processor receives the persistent request activation and updates its persistent request table. After receiving the activation message, each processor will send to P_0 any tokens and data it current has (as P_3 does) or that arrive later (as P_2 does at time ❹). When P_0 has received data and both tokens (at time ❺), P_0 performs a memory operation and sends deactivation messages to the system components. When each processor receives a deactivation message (at time ❻), it deactivates the request by clearing its persistent request table (reverting the system back to normal behavior).

sages (Section 4.7 describes several implementation alternatives for enforcing this requirement). Figure 4.5 illustrates the operation of this distributed-arbitration approach.

While this approach prioritizes to which processor the system should next send tokens, it does not (yet) prevent starvation. To prevent higher-priority processors from starving out lower-priority processors, this approach uses a simple mechanism inspired by techniques used to enhance mul-

tiprocessor bus arbitration mechanisms [118, 120]. Our mechanism prevents a higher-priority processor from making a new persistent request for the block until all the lower priority processors have completed their persistent requests. When a processor completes a persistent request, it sets a *marked* bit on each of the entries for that block in its local persistent request table, and a processor is not allowed to invoke a persistent request for any block with a marked entry in its table. The processor will eventually receive deactivation messages for all the marked entries, allowing the processor to once again issue a persistent request for that block. This approach prevents starvation by bounding the number of higher-priority persistent requests that processors can invoke before the substrate satisfies lower-priority persistent requests.

This decentralized-arbitration approach has advantages and disadvantages that prevent a clear choice of the best persistent request mechanism. Lower latency (due to direct communication) and lack of arbiter (and associated queuing) are the main advantages of the decentralized approach over arbiter-based approaches. The distributed-arbitration approach has three main disadvantages. First, the persistent request table requires associative matching and priority encoding logic, increasing the hardware complexity, size, and access latency of the structure (especially for a large number of processors). Second, the fixed-priority scheme might create load imbalance when using certain types of non-fair synchronization primitives. Third, this approach is somewhat more subtle (and possibly more error prone) than the arbiter-based approaches.

4.6 Improved Scalability of Persistent Requests

Although persistent requests are intended to be used infrequently, persistent requests are one limit of the scalability of Token Coherence. Although scalability is *not* a primary design goal of Token Coherence (because most machines have a small to moderate number of processors), this section briefly discusses the two main impediments to the scalability of these implementations of persistent requests.

First, as currently described, the system broadcasts persistent requests to all processors. To relax the broadcast-always nature of persistent requests, a starving processor (or the arbiter in

arbiter-based approaches) can initially send a persistent request only to a predicted subset of processors that are actively manipulating the block. Only when the request is not satisfied within another timeout window does the mechanism need to resort to broadcasting persistent requests throughout the system.

Second, each persistent request table requires either one entry per arbiter (in arbiter-based schemes) or one entry per processor (in decentralized-arbitration schemes). This $\Theta(n^2)$ amount of state limits the scalability of the proposed persistent request schemes. Fortunately the constant factor of the equation is small (*e.g.*, eight bytes per entry), allowing tables that support hundreds or thousands of processors with modest hardware cost. The arbiter-based scheme may be more appropriate for such large systems, because (1) it avoids associative search of the persistent request tables, and (2) it can allow for fewer than one arbiter per processor. For example, consider a chip multiprocessor (CMP) that consists of 16 processors and a single on-chip arbiter. A 1024-processor system of 64 of these CMP nodes would require persistent request tables with only 64 entries (one entry for each arbiter).

4.7 Preventing Reordering of Persistent Request Messages

Both the arbiter-based and distributed persistent request mechanisms assume that activation and deactivation messages are never reordered. This section describes the problem that reordering of these messages causes and presents several implementation approaches to avoiding the problem.

4.7.1 Problems Caused by Reordered Activations and Deactivations

Reordering of activation and deactivation messages can create confusion among the recipients of the messages. For example, consider the following situation: the system activates a request by sending it to all processors, and the processor with the tokens sends all its tokens to the initiator of the request, but the message sent to another processor is delayed in the interconnect (*e.g.*, due to congestion). While the interconnect is delaying that message, the initiator deactivates its persistent request (even before the persistent request has arrived at all its recipients). Furthermore, additional requests might be activated and deactivated before the first activation arrives at its destinations.

Thus, if the interconnect is allowed to reorder these messages, a processor could receive the activations and deactivations in the wrong order, with disastrous consequences. This reordering could cause a processor to erroneously conclude that an active request has been deactivated, a deactivated request is still active, or that the oldest request is the most recent and the most recent request is the oldest. Although these situations may be rare in practice, they can lead to starvation (in both the arbiter-based and distributed-arbitration approaches). To prevent this undesirable situation, a system designer has at least four choices, described in the next four sections.

4.7.2 Solution#1: Point-to-point Ordering

As we assumed in the initial discussion of persistent requests, the interconnect can prevent such troublesome reordering by providing point-to-point ordering. Point-to-point ordering concerns delivery ordering only between pairs of end points, and thus many interconnects naturally provide this property. [5] However, some interconnects may not provide point-to-point ordering because they exploit (1) adaptive routing or (2) message retransmission:

- Interconnects often use adaptive routing to reduce queuing delays caused by localized interconnect congestion. Unfortunately, adaptively routed interconnects allow messages to take different paths between endpoints, and thus such systems do not naturally provide point-to-point ordering. A reasonable compromise for capturing most of the benefits of adaptive routing without compromising point-to-point ordering for persistent requests is to selectively disable adaptive routing for those virtual networks that carry activation and deactivation messages.

- Interconnects that seek increased reliability using link-level or end-to-end retransmission of messages may also violate point-to-point ordering when faults that require retransmission occur. Although some such retransmission mechanisms will fail to maintain ordering, systems can adopt well-known techniques used in networking (*e.g.*, sliding window retransmis-

[5] As discussed earlier (in Section 2.3), point-to-point ordering is a much weaker form of ordering in the interconnect than the totally-ordered interconnect property required for snooping protocols (and directory protocols that exploit implicit acknowledgments [41]).

sion) to reestablish the appearance of point-to-point ordering. For example, the networking protocol TCP provides end-to-end message ordering over a unreliable substrate.

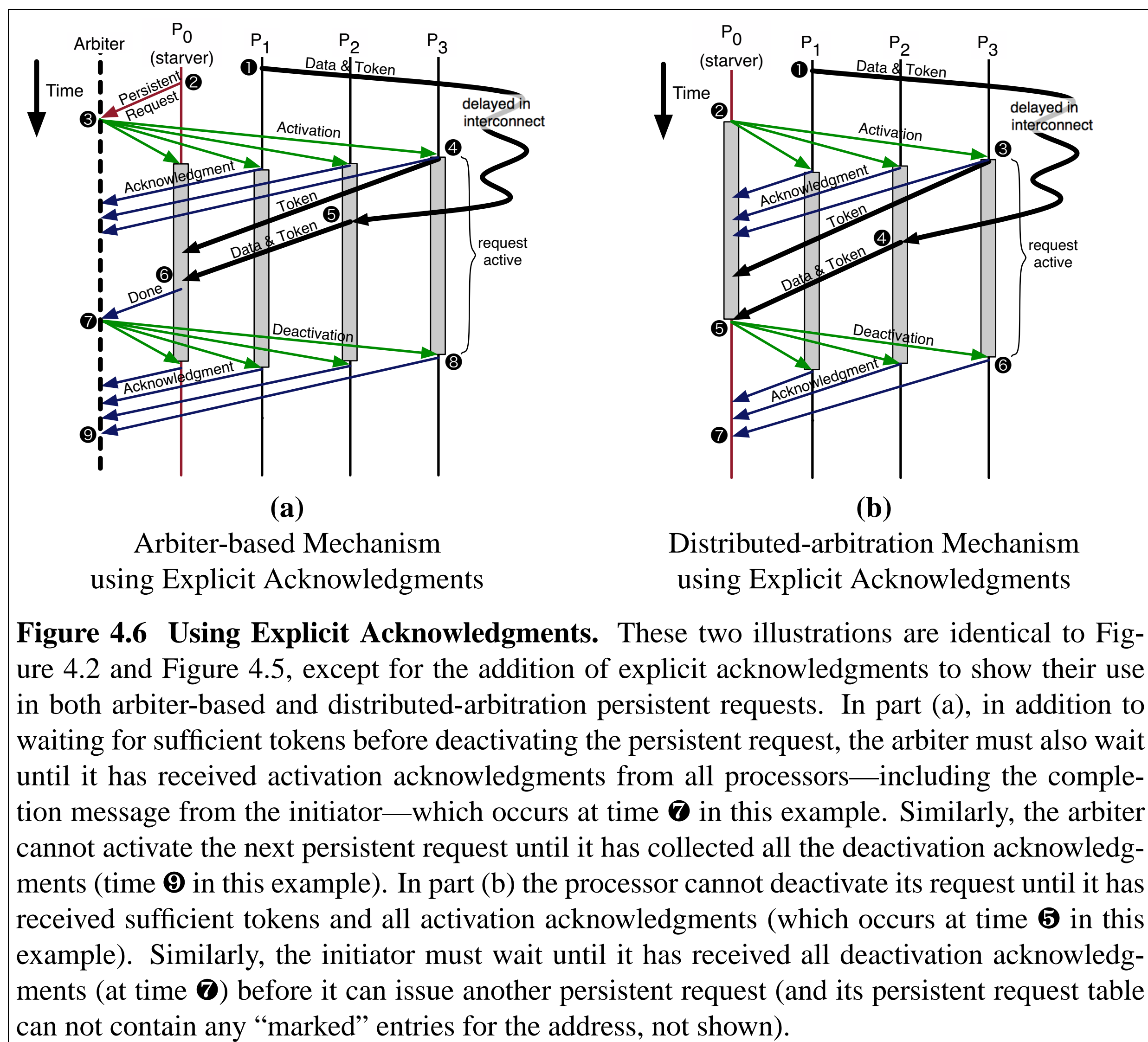
Relying on point-to-point ordering has advantages and disadvantages. Its main disadvantage is that it requires the interconnect to provide point-to-point ordering on one or more virtual networks (by disabling adaptive routing and/or using a retransmission protocol that reestablishes point-to-point order). Its main advantages are that (1) it is conceptually straightforward at the coherence protocol layer, (2) it does not complicate the persistent request mechanism, and (3) it avoids introducing additional messages that increase system traffic.

4.7.3 Solution#2: Explicit Acknowledgments

Instead of relying on the interconnect, a persistent request implementation can use explicit acknowledgment messages to guarantee that each arbiter or processor (in the arbiter-based and distributed approaches, respectively) has outstanding messages for only one active persistent request at a time. Before a component can issue another activation or deactivation, each previous recipient must send an acknowledgment confirming that it received the message. As illustrated in Figure 4.6, this approach adds two rounds of explicit acknowledgments for each persistent request.

This approach has the advantage of not relying on any specific interconnect ordering properties, but it has the disadvantages of (1) adding complexity to the persistent request mechanism, (2) requiring additional virtual networks for acknowledgment messages, and (3) increasing message traffic.

Unfortunately, the relative cost (in terms of bytes of traffic on the interconnect) of the acknowledgments is substantially worse for interconnects that support efficient broadcasts. For example, a broadcast on a two-dimensional toroidal network that uses fan-out routing will cross $\Theta(n)$ interconnect links (in which n is the number of processors). However, when each of these processors sends an acknowledgment response, the responses (in aggregate) cross $\Theta(n\sqrt{n})$ links. Due to the larger order of growth, for large systems the traffic caused by these acknowledgments will greatly exceed the traffic generated by the persistent request activation and deactivation messages. This



(a)
Arbiter-based Mechanism
using Explicit Acknowledgments

(b)
Distributed-arbitration Mechanism
using Explicit Acknowledgments

Figure 4.6 Using Explicit Acknowledgments. These two illustrations are identical to Figure 4.2 and Figure 4.5, except for the addition of explicit acknowledgments to show their use in both arbiter-based and distributed-arbitration persistent requests. In part (a), in addition to waiting for sufficient tokens before deactivating the persistent request, the arbiter must also wait until it has received activation acknowledgments from all processors—including the completion message from the initiator—which occurs at time ❼ in this example. Similarly, the arbiter cannot activate the next persistent request until it has collected all the deactivation acknowledgments (time ❾ in this example). In part (b) the processor cannot deactivate its request until it has received sufficient tokens and all activation acknowledgments (which occurs at time ❺ in this example). Similarly, the initiator must wait until it has received all deactivation acknowledgments (at time ❼) before it can issue another persistent request (and its persistent request table can not contain any “marked” entries for the address, not shown).

greater cost of sending acknowledgments is known as *ack implosion*, and its extreme effect is illustrated by the traffic use of the HAMMEROPT protocol (as discussed later in Section 7.2).

4.7.4 Solution#3: Acknowledgment Aggregation

To reduce the negative consequences caused by acknowledgment implosion, the persistent request mechanism can aggregate (or batch) multiple acknowledgments in a single message. For example, instead of acknowledging both the activation and deactivation, a recipient can reduce the acknowledgment overhead by half by sending acknowledgments only after it has received both the

activation and deactivation. If the deactivation arrives before the activation, the recipient must recognize the situation and remember it has already received the deactivation for a future activation.

To further reduce the traffic caused by acknowledgments, the substrate can give each activation and deactivation a source-specific sequence number (each activator has an incrementing counter), and recipients use these sequence numbers to ignore any delayed activation or deactivation messages that arrive out of sequence order. Each recipient must annotate the persistent request table to track (1) the highest sequence number it has received for the entry and (2) the number of messages it has received since it last sent an acknowledgment. The recipient uses this extra information to both ignore stale requests and send an acknowledgment when the count reaches a predetermined number (*e.g.*, 15 for a 4-bit sequence number space). When the activator reaches this limit, it must wait for all acknowledgments before it can continue to activate persistent requests (starting again with the sequence number zero).

Aggregating acknowledgments has the advantage of reducing the acknowledgment overhead by a potentially large constant factor (the traffic is reduced by a factor equal to the number of sequence numbers). The main disadvantage is the extra logic and state used to encode the sequence numbers in the messages and persistent request table (and the associated increase in design complexity).

4.7.5 Solution#4: Large Sequence Numbers

The previous solution introduced the use of sequence numbers to mitigate acknowledgment overhead by reducing the frequency of acknowledgments. Alternatively, by using a large enough sequence number such that—in practice—messages always have unambiguous sequence numbers, the system can simply eliminate persistent request acknowledgments. For example, if an arbiter uses a 4-byte counter and activates a persistent request every 100ns, the sequence number space would wrap around only every 3.4 minutes; if a message took that long to deliver, the system software would have declared an error long before the time limit was exceeded. To correctly handle sequence number wrap-around, the system components use an algorithm (based upon algorithms for handling finite sequence numbers in retransmission schemes) to determine if they should ignore the incoming message. For completeness, the algorithm is given in Figure 4.7. This approach of

```
function ignore(last_seq, incoming_seq):
    diff = incoming_seq - last_seq
    if diff > MAX_SEQ/2:
        return diff <= MAX_SEQ
    elif diff <= -MAX_SEQ/2:
        return diff <= -MAX_SEQ
    else:
        return diff <= 0
```

Figure 4.7 Algorithm for determining when a recipient should ignore an incoming message. If the `ignore()` function returns *true*, the recipient ignores the incoming message. In the pseudo-code above, `last_seq` is the sequence number of the last message that was not ignored, `incoming_seq` is the sequence number of the incoming message, and `MAX_SEQ` is the maximum sequence number allowed in a message.

using a sufficiently large sequence number space and handling sequence number wrap-around is inspired by similar mechanisms used in wide-area networking protocols (*e.g.*, TCP).

The main advantage of this approach is the removal of all acknowledgment traffic without relying on the interconnect, while preserving the simplicity of the persistent request mechanism. Its two main disadvantages are that (1) it adds a large sequence number (*e.g.*, four bytes) to the persistent request table and to each activation and deactivation message, and (2) additional logic is needed to determine which messages the recipients should ignore.

4.7.6 Summary of Solutions

Any of these four approaches can address the problem of reordered activation and deactivation messages. Although each approach has its benefits and liabilities, relying on interconnect point-to-point ordering for a single virtual network is a pragmatic approach due to its bandwidth efficiency

and simple nature. For these two reasons, the quantitative evaluations later in this dissertation adopt this approach.

4.8 Persistent Request Summary

This chapter has introduced persistent requests as one approach for preventing starvation in Token Coherence. The chapter also presented several approaches for implementing persistent requests and additional options for preventing the reordering of activation and deactivation messages. Due to the large design space for implementing persistent requests, we selected a single proof-of-concept design point for use in our quantitative evaluations. We selected the distributed-arbitration scheme (Section 4.5) and use point-to-point ordering in the interconnect to avoid reordering activation and deactivation messages (Section 4.7.2). Evaluation of the various persistent request alternatives and development of other approaches are relegated to future work.

Chapter 5

Performance Policies Overview

The previous two chapters describe what is required for Token Coherence to provide correct operation in all cases; in contrast, this chapter describes alternatives for making Token Coherence fast in the common case. The system's *performance policy* is the set of specific policies the system uses to instruct the correctness substrate to move tokens and data throughout the system. When it is not overridden by a persistent request, the performance policy decides when and to which processors the system should send various coherence messages. Token Coherence allows for many possible performance policies, and since its correctness substrate guarantees safety and prevents starvation, performance policy designers can innovate without fear of corner-case correctness errors. The remainder of this dissertation describes and evaluates various performance policies and shows that Token Coherence enables the creation of coherence protocols that exhibit lower latency and are more flexible than traditional protocols.

This chapter first discusses performance policy obligations (Section 5.1) and opportunities (Section 5.2). It then provides a brief overview of the performance policies explored in this dissertation (Section 5.3) and discusses other possible performance policies that could be explored in future work (Section 5.4).

5.1 Obligations

Performance policies have no obligations, because (1) the correctness substrate ensures correctness (safety via token counting and starvation prevention via persistent requests), and (2) processors are responsible for asking the correctness substrate to invoke the persistent request mecha-

nism. Thus, even a null or random performance policy would not be incorrect (but might perform poorly). Therefore, performance policies may aggressively seek performance without concern for corner-case errors.

5.2 Opportunities via Transient Requests

Due to a lack of obligations, a performance policy has significant freedom to dictate system behavior; as long as it is not overridden by a persistent request, the performance policy both (1) determines how the substrate moves coherence messages throughout the system and (2) introduces its own *hint* messages to communicate between system components. When a performance policy uses these hint messages to emulate request messages commonly found in traditional coherence protocols, these hints are called *transient requests* (to clearly distinguish them from the substrate's persistent requests).

One way in which performance policies seek high performance is by specifying an approach for using these transient requests. The system sends fast, unordered transient requests to one or more components, and the components respond with all, some, or none of their tokens and possibly data (all specified by the particular performance policy). Transient requests often succeed in collecting data and sufficient tokens (one token to read the block, and all tokens to write it), but they may also fail to obtain sufficient tokens due to races, insufficient recipients, or being overridden by the correctness substrate. When a transient request fails to obtain sufficient tokens, the performance policy may reissue the transient request or do nothing (because the memory operation will eventually complete after the processor detects lack of progress and issues a persistent request). A good performance policy will use transient requests to quickly satisfy most cache misses.

5.3 Performance Policy Forecast

This dissertation uses the flexibility of Token Coherence and transient requests to create a progression of three performance polices. The end result of this progression is a coherence protocol with lower latency than traditional snooping or directory protocols that also uses significantly

less bandwidth than snooping protocols. The three performance policies in this progression are TOKENB (a low-latency broadcast-based protocol), TOKEND (a bandwidth-efficient directory-like protocol), and TOKENM (a hybrid that uses predictive multicasting to create a low-latency and bandwidth-efficient protocol).

5.3.1 TOKENB

The goal of TOKENB is to provide the lowest possible latency for requests by (1) always broadcasting transient requests (to directly find data anywhere in the system), and (2) avoiding the performance bottlenecks of totally-ordered interconnects (since transient requests are simply hints, they have no ordering requirements). In essence, TOKENB is a snooping protocol without the difficulties caused by the totally-ordered interconnect required by traditional snooping systems. To handle occasional situations in which a transient request fails to collect sufficient tokens, TOKENB reissues the transient request after a timeout period, ultimately relying on the correctness substrate's persistent request mechanism to prevent starvation.

5.3.2 TOKEND

The goal of TOKEND is to provide a bandwidth-efficient performance policy to both (1) show that Token Coherence can achieve similar bandwidth efficiency as a directory protocol and (2) provide a bandwidth-efficient base for the hybrid protocol TOKENM. TOKEND achieves this goal by, in essence, emulating a directory protocol by using transient requests. Transient requests are first sent to the home memory module, where a per-block "soft state" directory determines to which processor, if any, it should forward the request. This policy results in a token-based coherence protocol that—to the first order—has the both the desirable traffic characteristics and the undesirable directory-indirection latency penalty of directory protocols.

5.3.3 TOKENM

The goal of TOKENM is to combine the low-latency of TOKENB with the bandwidth efficiency of TOKEND to create a hybrid with more attractive bandwidth and latency characteristics than any

traditional cache-coherence protocol (*i.e.*, capturing most of the latency benefits of TokenB while using only slightly more bandwidth than a directory protocol or TokenD). TokenM achieves this goal by multicasting transient requests to a predicted *destination set* of processors based on the observation of past system behavior [3, 4, 20, 79]. A perfect destination-set prediction includes only those processors that need to respond to a request, excluding uninvolved processors to reduce traffic. A system with a perfect destination-set predictor would capture the low-latency of TokenB with the bandwidth-efficiency of TokenD. This dissertation does not develop any new destination-set predictors, but instead uses some of the predictors described by Martin *et al.* [79] as a proof of concept that Token Coherence can capture the benefit from such a predictive scheme.

Previously proposed protocols that use destination-set prediction [3, 4, 20, 79] require complicated protocols or protocol extensions. In contrast, Token Coherence exploits destination-set prediction simply by changing the performance policy to multicast transient requests. In essence, Token Coherence provides a simpler implementation of these proposals, while eliminating the totally-ordered interconnect required by some proposals [20, 79] and complex races in other proposals [3, 4, 20, 79].

5.4 Other Possible Performance Policies

Although this dissertation describes and evaluates three performance policies in detail, many other performance policies are possible. We leave full exploration of further applications of Token Coherence to future work, but we first briefly describe some of the promising avenues for enhancement enabled by Token Coherence.

5.4.1 Bandwidth Adaptive Protocols

TokenM is only one way of building a directory/snooping hybrid using Token Coherence. Another promising approach to building a hybrid is to use bandwidth adaptive techniques that dynamically adjust to interconnect utilization [84]. For example, a performance policy could broadcast requests (like TokenB) when interconnect utilization is low, and it could first send transient requests to the home memory (like TokenD) when the interconnect utilization is high. In addi-

tion, predictive techniques (like those used in TOKENM) can be incorporated into such a scheme by sending to a predicted destination-set when the interconnect has moderate utilization. By dynamically adjusting the amount of request bandwidth available, such adaptive techniques create robust protocols that adapt to both the workload and number of processors in the system. This adaptive approach achieves similar performance or—for some configurations—significantly better performance than the base protocols [84].

5.4.2 Predictive Push

In addition to using transient requests to reduce the latency of misses, Token Coherence also provides an opportunity to reduce the *number* of cache misses by predictively pushing data between system components. For example, a performance policy will avoid a cache miss if it can predictively send data and tokens to a processor before the processor accesses the block. This predictive transfer of data can be triggered by a coherence protocol predictor [1, 66, 102], by software (*e.g.*, the KSR1's "poststore" [111] and DASH's "deliver" [75]), or by allowing the memory to push data into processor caches. Similar mechanisms have been used to efficiently implement software-controller message passing and coherent block transfer of data [50, 51]. Because Token Coherence allows data and tokens to be transferred between system components without affecting correctness,[1] these schemes are easily implemented correctly as part of a performance policy. While designers can modify other protocols to support predictive push, due to Token Coherence's separation of performance and correctness adding such an optimization to Token Coherence will likely require fewer changes and have a lower risk of introducing design errors.

5.4.3 Multi-Block Request or Prefetch

Since transient requests are only hints, they can be used to request more than one block at a time. For example, if a processor detected a unit-stride sequence of requests, the performance

[1]The substrate avoids deadlock using its second data/token virtual network that gives a processor the choice of redirecting any data it receives to the memory. This pair of data/token virtual networks was described in Section 4.2.1.

policy could use a special transient request to request that the memory send the next several blocks of data to the processor. Alternatively, special software instructions could prefetch a large number of blocks into the cache with low request overhead. This approach could both (1) reduce the request traffic in the interconnect and (2) allow the memory to efficiently stream data directly from DRAM to the requester.

5.4.4 Supporting Hierarchical Systems

Token Coherence may also accelerate hierarchical systems, an increasingly important concern with the rise of chip multiprocessors (CMPs), such as IBM's Power4 [119]. Power4's coherence protocol employs extra protocol states to enable neighboring processors to respond with data, reducing traffic and average miss latency. A Token Coherence performance policy could achieve this more simply by granting extra tokens to requesters, and allowing those processors to respond with data and tokens to neighboring processors. Other hierarchical systems connect smaller snooping-based modules into larger systems (*e.g.*, [15, 23, 75]). Token Coherence may allow for a single protocol to more simply achieve the latency and bandwidth characteristics of these hierarchical systems, without requiring the complexity of two distinct protocols and the bridge logic between them.

5.4.5 Reducing the Frequency of Persistent Requests

One shortcoming of the performance policies discussed in detail in this dissertation is that the only action they take to avoid persistent requests is to reissue a transient request after a timeout period. However, performance policies could use a significantly smarter mechanism to decrease the frequency with which processors invoke persistent requests by detecting and more intelligently handling conflicting requests. For example, the performance policy could observe conflicting transient requests in the system and predictively forward data and tokens to another processor without waiting for a persistent request or reissued transient request. Such an approach might reduce the frequency of persistent requests and reissued transient requests improving latency, traffic, and scalability.

5.5 Roadmap for the Second Part of this Dissertation

This chapter concludes the first part of this dissertation. The previous two chapters described the correctness substrate and this chapter provided an overview of possible performance policies and forecasted the three performance policies we describe and evaluate in the second part of this dissertation. The second part of this dissertation begins by presenting our evaluation methodology (Chapter 6), and it continues with three chapters that fully describe and evaluate each of the three performance policies: TokenB (Chapter 7), TokenD (Chapter 8), and TokenM (Chapter 9). The final chapter of the dissertation presents our conclusions (Chapter 10).

Chapter 6

Experimental Methods and Workload Characterization

We use simulation to provide insight into the effectiveness of Token Coherence by exploring the behavior of several of its performance policies. The goal of this evaluation is to illuminate the relative behavior of various coherence protocols (both traditional protocols and ones based on Token Coherence). Our goal is *not* to either (1) produce absolute execution times or throughput rates for our simulated systems or (2) evaluate these protocols on all possible future system configurations. Instead, we strive for accurate relative comparisons using an approximation of a next-generation multiprocessor system. To achieve this goal we use full-system simulation and model the first-order timing effects to approximate an aggressive multiprocessor system running commercial workloads. We aim to capture the first-order effects, but—like most architectural simulations—we do not attempt to model every aspect of the system in exact detail. This chapter presents our simulation tools (Section 6.1), the specifics of our simulated system (Section 6.2), our commercial workloads (Section 6.3), and a brief workload characterization to provide some insight into the memory-system behavior of our workloads (Section 6.4).

6.1 Simulation Tools

We use the Simics full-system multiprocessor simulator [77], and we extend Simics with a processor and memory hierarchy model to compute execution times [10]. Simics is a system-level architectural simulator developed by Virtutech AB that can run unmodified commercial applications and operating systems. Simics is a functional simulator only, but it provides an interface to support our detailed timing simulations. We use TFsim [86] to model superscalar processor cores

that are dynamically scheduled, exploit speculative execution, and generate multiple outstanding coherence requests. Ruby, our detailed memory hierarchy simulator, models (1) a two-level cache hierarchy, (2) the latency and bandwidth of the interconnect, and (3) timing races and all state transitions (including non-stable states) of the coherence protocols. The next section describes the specific system configuration, coherence protocols, interconnects, and timing parameters used in our simulations. The timing of I/O operations is not modeled, but the operating system device drivers still execute the normally required instructions, and the simulations capture the effects of context switching and interrupts caused by I/O.

We performed limited validation of the memory system using test traces of memory operations. We verify various timing aspects of the system using these traces (*e.g.*, replacement policy, uncontended latencies, behavior under contention, correct implementation of the migratory sharing optimization, and implementation of the EXCLUSIVE state). We also performed non-performance validation using randomized testing with randomized message latencies. This approach stresses protocol corner cases by making these cases more frequent.

6.2 Simulated System

To evaluate Token Coherence, we simulate a multiprocessor server running commercial workloads using multiple interconnects and coherence protocols. Our target system is a 16-processor SPARC v9 system with highly integrated nodes that each include a dynamically-scheduled processor, split first level instruction and data caches, unified second level cache, coherence protocol controllers, and a memory controller for part of the globally shared memory. The system implements sequential consistency using invalidation-based cache coherence and an aggressive, speculative processor implementation [40, 126].

We selected a number of coherence protocols, system interconnects, latencies, bandwidths, cache sizes, and other structure sizes. Table 6.1 lists the system parameters for both the memory system and the processors. We selected these parameters to approximate next-generation systems based on the published parameters of the Alpha 21364/GS1280 systems [33, 47, 91]. We limit

the bandwidth of cache controllers and memory controllers. These limits also indirectly limit the bandwidth that is caused by external requests for the DRAM, cache tag arrays, and cache data arrays. We perform simulations both with unbounded interconnect link bandwidth and interconnects with 4GB/sec links. These two types of simulations allow us to separate the effects of changes in uncontended latency from changes in latency due to interconnect bandwidth constraints. The coherence protocols and interconnection networks are described next.

6.2.1 Coherence Protocols

We compare simulated systems using a few distinct MOESI coherence protocols. All the protocols implement the previously described migratory sharing optimization (Section 2.2.4) which improves the performance of all the protocols. None of the protocols support upgrade requests (Section 2.2.5). Coherence is maintained on aligned 64-byte blocks. All request, acknowledgment, invalidation, and dataless token messages are 8 bytes in size (including the 40+ bit physical address and token count if needed); data messages include this 8-byte header and 64 bytes of data. These message sizes do not include any extra bits used by the interconnect to detect and correct bit errors. We use three base protocols: SNOOPING (an aggressive snooping protocol described in Section 2.4.5), DIRECTORY (a traditional directory protocol described in Section 2.5.5), and HAMMEROPT (an optimized version of a protocol that approximates AMD's Hammer/Opteron protocol described in Section 2.6.2).

6.2.2 System Interconnects

We use the two interconnects described in Section 2.3.4: (1) TREE, an ordered "virtual bus" pipelined broadcast tree that is sufficient for traditional snooping, and (2) TORUS, an unordered two-dimensional torus. Both of these interconnects are illustrated in Figure 2.1 on page 29. We selected these two interconnects because they both use high-speed point-to-point links. We do not consider shared-wire (multi-drop) buses, because designing high-speed buses is increasingly difficult due to electrical issues [35, section 3.4.1]. We selected the link bandwidth of 4 GBytes/second (4-byte wide links at 1 Ghz) and latency of 15ns based on descriptions of current systems (*e.g.*,

Table 6.1 Simulation Parameters

Coherent Memory System Parameters	
cache block size	64 bytes
L1 instruction cache	64KB, 4-way set associative, 1ns latency (2 cycles)
L1 data cache	64KB, 4-way set associative, 1ns latency (2 cycles)
L2 unified cache	4MB, 4-way set associative, 6ns latency (12 cycles) 6ns (12 cycles) to send data to external requester
main memory	4GB, 80ns (160 cycles), includes controller latency
standard directory	Off-chip DRAM, 80ns (160 cycles), includes controller latency
fast directory	On-chip SRAM, 6ns (6 cycles), includes controller latency
interconnect link	high-speed point-to-point, 4GB/second or unbounded bandwidth 15ns latency (30 cycles); includes wire, sync., and routing delay
interconnect interface	4ns (8 cycles) to enter or exit the interconnect
Dynamically-Scheduled Processor Parameters	
clock frequency	2 Ghz
issue/execute width	4 instructions
reorder buffer	128 entries
pipe stages	11 stages
direct branch predictor	1KB YAGS
indirect branch predictor	64 entry
return address stack	64 entry
Calculated Average Miss Latencies	
TREE interconnect hop	4ns interface + 15ns x 4 links + 4ns interface = 68ns (136 cycles)
TORUS interconnect hop	4ns interface + 15ns x 2 links + 4ns interface = 38ns (76 cycles)
memory-to-cache	L2 miss + 2 hops + mem TREE: 6 + (2 x 68) + 80 = 222ns (444 cycles) TORUS: 6 + (2 x 38) + 80 = 162ns (324 cycles)
direct cache-to-cache	L2 miss + 2 hops + cache TREE: 6 + (2 x 68) + 6 = 148ns (296 cycles) TORUS: 6 + (2 x 38) + 6 = 88ns (176 cycles)
indirect cache-to-cache	L2 miss + 3 hops + dir + cache TREE– DRAM dir.: 6 + (3 x 68) + 80 + 6 = 296ns (592 cycles) TREE– SRAM dir.: 6 + (3 x 68) + 6 + 6 = 222ns (444 cycles) TORUS– DRAM dir.: 6 + (3 x 38) + 80 + 6 = 206ns (412 cycles) TORUS– SRAM dir.: 6 + (3 x 38) + 6 + 6 = 132ns (264 cycles)

the Alpha 21364/GS1280 [33, 47, 91] and AMD's Hammer/Opteron [9, 121]). We also perform simulations with unbounded bandwidth to isolate the effect of limited bandwidth on performance. Messages are multiplexed over a single shared interconnect using virtual networks and virtual channels, and messages with more than one destination (*e.g.*, broadcasts) use bandwidth-efficient tree-based multicast routing [36, section 5.5]. Using multicast routing benefits all our protocols; even our directory protocol benefits (by using multicast routing when forwarding requests to more than one processor).

6.3 Workloads and Measurement Methods

This section describes the methods we use to generate meaningful results from simulations of commercial workloads (Section 6.3.1) and describes our specific workloads (Section 6.3.2). In the next section (Section 6.4), we present a brief characterization of these workloads.

6.3.1 Methods for Simulating Commercial Workloads

We use a transaction-counting methodology [10] for determining throughput of our systems running commercial workloads. We initialize the system state using a full-system checkpoint (to provide a well-define starting point) and simulate the execution until the simulated machine has completed a fixed number of transactions (*e.g.*, database transactions or web server requests). We record the number of cycles needed to complete a fixed number of transactions. We use this metric to calculate transactions per cycle (an inverse measure of throughput, the ultimate metric of performance for server workloads). We use the runtime of a fixed number of transactions to conclude that "protocol A is X% faster than protocol B" using the formula $X = (runtime(B)/runtime(A) - 1.0) \cdot 100$.

We avoid using instructions per cycle (IPC) as a metric of performance. Because system timing effects of multiprocessor workloads can change the number of instructions executed, running the simulator for a fixed number of instructions and measuring instructions per cycle (IPC) is *not* guaranteed to reflect the performance of the system [10, 11]. For example, consider a coherence protocol improvement that reduces the time processors spend spinning on locks. Because proces-

sors can rapidly execute instructions while in tight spin loops, reducing idle spinning (usually a good thing) will actually cause instructions per cycle to decrease (usually a bad thing) while the system performance is actually improved (the desired effect).

In addition to reporting runtime, we measure and report the traffic in terms of *endpoint traffic* (in messages per miss) and *interconnect traffic* (in terms of bytes on interconnect links per miss). The endpoint traffic indicates the amount of controller bandwidth required to handle incoming messages. The interconnect traffic indicates the amount of link bandwidth consumed by the messages as they traverse the interconnect. The former metric is mostly independent of the particular interconnect and message size. In contrast, the latter metric is influenced by interconnect topology, the use of bandwidth-efficient multicast routing, and message size.

Due to the computational intensity of detailed architectural timing simulations, we are limited to simulating only a short segment of the workload's entire execution. We use two techniques to partially overcome the problems introduced by such limitations. First, all workloads were warmed up and checkpointed to avoid system cold-start effects, and we ensure that caches are warm by restoring the cache contents captured as part of our checkpoint creation process. Second, to address the variability in commercial workloads, we adopt the approach of simulating each design point multiple times with small, pseudo-random perturbations of request latencies [10, 11]. These perturbations cause alternative operating system scheduling paths in our otherwise deterministic simulations. Running many of these pseudo-randomly perturbed systems creates a distribution of runtimes. We remove all data points further than 1.5 standard deviation from the mean to reduce the effect of the skewed nature of the distribution.[1] Error bars in our runtime results approximate a 95% confidence interval centered around the arithmetic mean of the remaining data points. Each data point is the aggregate of approximately 5 to 15 data points, using a larger number of simulations for those configurations and workloads that exhibit the most variation.

[1] To understand why the distribution is skewed, consider measuring the round-trip time of a car trip between two destinations. The distribution of the times recorded from many trials would form a near-normal distribution. However, a serious traffic jam or car breakdown could result in a trip with several times the mean. With an extremely large number of samples, this would not significantly affect the results. However, if only a handful samples are used, such outliers are best removed when calculating the expected travel time.

6.3.2 Workload Descriptions

Our benchmarks consist of three commercial workloads: an online transaction processing workload (OLTP), a Java middleware workload (SPECjbb), and a static web serving workload (Apache). These workloads execute on a simulated 16-processor SPARC multiprocessor running Solaris 9. The simulated system has 4GBs of main memory.

- **Online Transaction Processing (OLTP): DB2 with a TPCC-like workload.** The TPC-C benchmark models the database activity of a wholesale supplier. Our OLTP workload is based on the TPC-C v3.0 benchmark using IBM's DB2 v7.2 EEE database management system. Our experiments simulate 256 users (16 per processor) without think time. The simulated users query a 5GB database with 25,000 warehouses stored on eight raw disks and a dedicated database log disk. We reduced the number of districts per warehouse, items per warehouse, and customers per district to allow more concurrency provided by a larger number of warehouses. We use 100,000 transactions to warm the system and database buffer pool, 500 transactions to warm simulated hardware caches, and detailed simulations of 50 transactions for our reported results.

- **Java Server Workload: SPECjbb.** SPECjbb2000 is a server-side Java benchmark that models a 3-tier system, but its main focus is on the middleware server business logic. We use Sun's HotSpot 1.4.1-b21 Server JVM configured to use Solaris's native thread implementation. To reduce the frequency of garbage collection, we set the JVM's heap size to 2GB and the new object heap size to 600MB. Our experiments use 24 driver threads (1.5 per processor) and 24 warehouses (with a total data size of approximately 500MB). We use over a million transactions to warm the system, 100,000 transactions to warm simulated hardware caches, and detailed simulations of 2000 transactions for our reported results.

- **Static Web Content Serving: Apache.** Web servers such as Apache are an important enterprise server application. We use Apache 2.0.43 configured to use a hybrid multi-process multi-threaded server model with 64 POSIX threads per server process. Our experiments

use a hierarchical directory structure of 80,000 files (with a total data size of approximately 1.8GB) and a modified version of the Scalable URL Reference Generator (SURGE) [16] to simulate 6400 users (400 per processor) with an average think time of 12ms. We use 800,000 requests to warm the system, 1000 requests to warm simulated hardware caches, and detailed simulations of 100 requests for our reported results.

6.4 Workload Characterization

In this section we briefly characterize the commercial workloads used in this dissertation. We refer interested readers to Alameldeen *et al.* [10] and Martin *et al.* [79] for a more detailed description and characterization of similar versions of these workloads.[2]

6.4.1 Characterization of our Base Coherence Protocols

Table 6.2 and Table 6.3 contain the basic simulation data for our base protocols on the TREE and TORUS interconnects, respectively. We present these basic simulation metrics to both (1) provide a brief characterization of our workloads, and (2) provide sufficient "raw" data to understand our later performance results. The first column of these tables name the system configuration. The second column shows that the number cycles per transaction metric is worse (larger) for slower protocols (unsurprisingly). The third column shows that the number of misses per transaction is relatively stable across different protocols and interconnects (also unsurprising).

In contrast, some of the data in these tables are surprising. The number of instructions per transaction is *not* stable across configurations for two of our three workloads. For example, the difference in number of instructions per transaction for OLTP between SNOOPING and DIRECTORY is over a factor of three. Although we do not know the source of these extra instructions per transaction, such a dramatic increase is consistent with more time spent spinning on contended locks that are obtained more slowly or more time spent in the idle loop.

[2]Alameldeen *et al.* [10] and Martin *et al.* [79] report results using an earlier version of these workloads; the workloads and coherence protocols used in this dissertation have been incrementally improved since that work was published.

configuration	all cycles per transaction	L2 misses per transaction	instructions per transaction	cycles per instruction	misses per thousand instructions	endpoint msgs per miss	interconnect bytes per miss
SPECjbb							
Perfect L2	19,903	NA	54,820	0.36	NA	NA	NA
Snooping	44,941	185	57,884	0.78	3.20	20.63	274.94
Directory - fast	46,840	184	57,944	0.81	3.18	6.01	158.11
Directory - slow	48,921	184	58,105	0.84	3.18	6.01	158.01
Apache							
Perfect L2	176,559	NA	273,711	0.65	NA	NA	NA
Snooping	718,417	4,041	401,851	1.79	10.10	19.32	242.85
Directory - fast	796,974	4,045	401,614	1.98	10.09	5.36	131.36
Directory - slow	913,395	4,201	439,611	2.07	9.49	5.35	131.10
OLTP							
Perfect L2	1,747,629	NA	3,385,566	0.52	NA	NA	NA
Snooping	7,656,243	45,707	7,079,692	1.09	6.40	18.04	218.72
Directory - fast	12,932,199	50,725	20,061,525	0.62	2.39	4.69	111.05
Directory - slow	16,002,054	50,338	26,933,975	0.60	1.88	4.70	111.17

Table 6.2 Non-Token Coherence Protocol Results for the TREE Interconnect. This table contains simulation data for our coherence protocols not based on Token Coherence. All of these simulated configurations use unbounded interconnect link bandwidth. The left-most column is the name of the configuration, including whether the simulation uses a “fast” SRAM directory or “slow” DRAM directory (as applicable). The “perfect L2” configuration simulates an idealized system in which all references hit in the second-level cache. The second column is the number of cycles executed by all processors per transaction (*i.e.*, the total runtime in cycles multiplied by the number of processors). The next three columns (from left to right) are: second-level cache misses per transaction, instructions per transaction, cycles per instruction, and misses per thousand instructions. The remaining two columns are metrics of system traffic: (1) endpoint messages per miss and (2) bytes on the interconnect links per miss. Since traffic per link on the TREE interconnect is non-uniform, we report the sum of the traffic only on the incoming links to the processor on this interconnect. In contrast, the link traffic in bytes per miss on the TORUS interconnect (shown in Table 6.3) is the sum of the traffic on all links.

configuration	all cycles per transaction	L2 misses per transaction	instructions per transaction	cycles per instruction	misses per thousand instructions	endpoint msgs per miss	interconnect bytes per miss
SPECjbb							
Perfect L2	19,903	NA	54,820	0.36	NA	NA	NA
Directory - fast	39,307	182	57,201	0.69	3.20	6.01	319.32
Directory - slow	41,102	182	57,270	0.72	3.18	6.01	318.37
Apache							
Perfect L2	176,559	NA	273,711	0.65	NA	NA	NA
Directory - fast	609,117	4,017	367,030	1.66	10.89	5.36	263.93
Directory - slow	702,715	4,063	390,853	1.78	10.31	5.35	263.71
OLTP							
Perfect L2	1,747,629	NA	3,385,566	0.52	NA	NA	NA
Directory - fast	7,301,596	47,189	7,000,286	1.00	6.40	4.64	227.65
Directory - slow	9,463,958	48,304	11,557,401	0.85	4.32	4.65	227.30

Table 6.3 Non-Token Coherence Protocol Results for the TORUS Interconnect. The metrics presented in this table were described in Table 6.2 on page 101. SNOOPING results are not included because TORUS does not provide the necessary ordering properties for SNOOPING.

As a consequence of the varying number of instructions per transaction, normalizing by instruction count can be misleading. As suggested by these tables, the number of cycles per instruction (CPI) is not always indicative of actual runtime or throughput. For example, consider OLTP on the TREE interconnect: DIRECTORY with the fast directory has better (smaller) CPI than SNOOPING (the fifth column of Table 6.2). Studying only the CPI, one might conclude SNOOPING is slower than DIRECTORY. However, the cycles per transaction column shows that SNOOPING is significantly faster than DIRECTORY (by almost a factor of two). The misses per thousand instructions column is similarly skewed by the non-stable number of misses per transaction. As we mentioned earlier, because of these effects, we use cycles per transaction as our only metric of performance.

In addition to our base protocols (SNOOPING and DIRECTORY), these tables also include the performance of a perfect second-level cache. In these idealized simulations, all references hit in the second-level cache. These tables show a significant difference in the number of cycles per

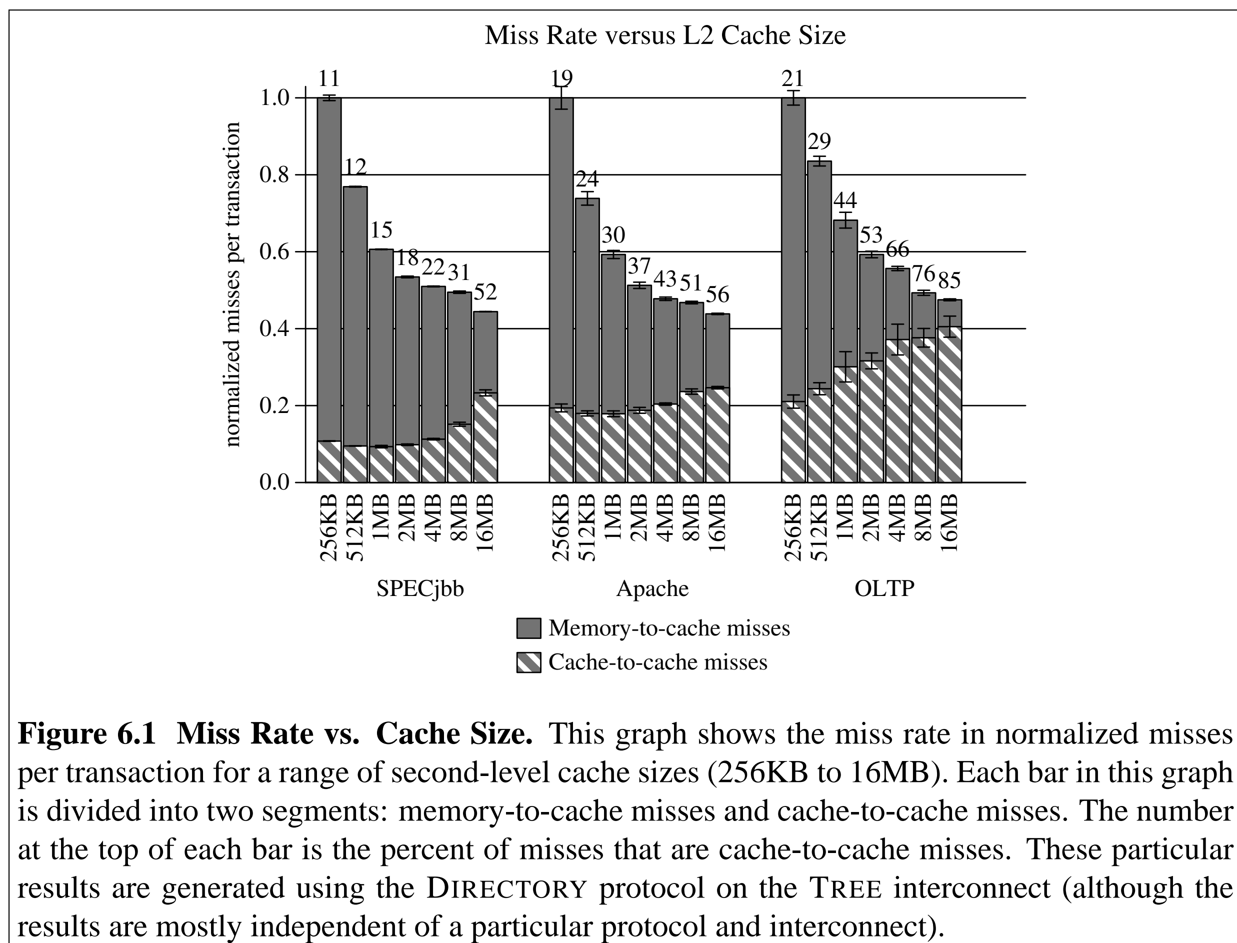


Figure 6.1 Miss Rate vs. Cache Size. This graph shows the miss rate in normalized misses per transaction for a range of second-level cache sizes (256KB to 16MB). Each bar in this graph is divided into two segments: memory-to-cache misses and cache-to-cache misses. The number at the top of each bar is the percent of misses that are cache-to-cache misses. These particular results are generated using the DIRECTORY protocol on the TREE interconnect (although the results are mostly independent of a particular protocol and interconnect).

transaction between the "perfect L2" configuration and our base protocols. For example, on the TORUS interconnect the perfect L2 configuration is about twice as fast as our base protocols for SPECjbb; for OLTP perfect L2 can be over 5 times faster. Because these workloads are spending the majority of their time in the memory system, and many of the cache misses are cache-to-cache misses, there is a significant performance opportunity for protocols that optimize for cache-to-cache misses (discussed next).

6.4.2 Cache-to-Cache Misses Occur Frequently

In Chapter 1, we asserted that cache-to-cache misses are frequent in commercial workloads and that their effect on performance is significant. To support this claim, Figure 6.1 shows the

normalized misses per transaction for our three workloads for a range of caches sizes. This graph provides several different insights into our workloads:

- First, the overall second-level miss rate for these workloads is significant, even for systems with large caches. For example, for the DIRECTORY with a fast directory and 4MB caches with the TREE interconnect, the miss rate is 2.5–10.1 misses per thousand instructions (as shown in Table 6.2 and Table 6.3). This corresponds to 98.6–393.9 instructions executed between each second-level miss on average. Because each second-level miss is hundreds of processor cycles long, these misses have a substantial impact on performance.

- Second, the graphs in Figure 6.1 show the unsurprising results that the overall number of misses per transaction (total height of each bar) and the memory-to-cache miss rate (the solid segment of each bar) both decrease as the cache size increases.

- Third, the number of cache-to-cache misses (the striped segment of each bar) actually *increases* as cache size increases. This increase occurs because (1) misses caused by sharing are not eliminated by larger caches, and (2) larger caches increase the likelihood that a requested block is being cached by another processor (this effect is especially pronounced for systems that support the EXCLUSIVE and OWNED states). For example, in the limit of infinite caches, in a MOESI protocol only the first access by any processor to a block will be a memory-to-cache miss and all other misses to that block (by any processor) will be cache-to-cache misses.

  If cache-to-cache misses are slower than memory-to-cache misses (*e.g.*, in a directory protocol with a high-latency directory) the increasing cache-to-cache miss rate is a detriment to performance. For such systems, researchers have proposed reducing cache-to-cache miss rates by using predictive invalidation techniques to proactively evict blocks [70, 73]. In contrast, many snooping protocols and directory protocols with a low-latency directory have faster cache-to-cache misses than memory-to-cache misses. In such systems, large caches that transform memory-to-cache misses into cache-to-cache misses improve performance.

- Fourth, the combined effect of the previous two effects—the increasing number of cache-to-cache misses and decreasing number of memory-to-cache misses—results in a large percentage of cache-to-cache misses (the number at the top of each bar in Figure 6.1). For our workloads on a system with a 4MB second-level cache, 22–66% of all second-level misses are cache-to-cache misses.

6.4.3 The Performance Cost of Indirection

Directory protocols use indirection to avoid broadcast, but this indirection places a third interconnect traversal and directory access latency on the critical path of cache-to-cache misses. Figure 6.2 shows the performance reduction due to these two sources of overhead for a range of second-level cache sizes.[3] We calculated the contribution of these sources of overhead (the individual segments in the bars) from three simulations: (1) SNOOPING, (2) DIRECTORY with a fast SRAM directory, and (3) DIRECTORY with a DRAM directory. This experiment uses the TREE interconnect and—to isolate the effects due only to uncontended miss latency—unbounded link bandwidth.

We quantify the two sources of directory protocol overhead by plotting bars with four segments. The bottom segment (solid dark grey) represents runtime of a system with a perfect L2 cache. The combined height of the bottom two segments is the runtime of SNOOPING, which suffers from neither overhead. The solid light grey segment shows the effect of DIRECTORY's additional interconnect traversal, calculated as the difference in runtime between SNOOPING and DIRECTORY with the a fast SRAM directory. The top segment (striped dark grey) shows the effect of DIRECTORY's directory lookup latency, calculated as the difference in runtime between DIRECTORY with a fast, SRAM directory and with a slow, DRAM directory.

As follows from the results in Figure 6.1, both (1) the absolute number of cycles of overhead and (2) the percentage of the runtime due to indirection increase as cache size increases. In this particular set of experiments, the directory latency and interconnect traversal each contribute ap-

[3]Although cache hit latency often increases with cache size, these simulations use the same cache hit latency for all cache sizes to isolate the effect of the higher hit rate of larger caches.

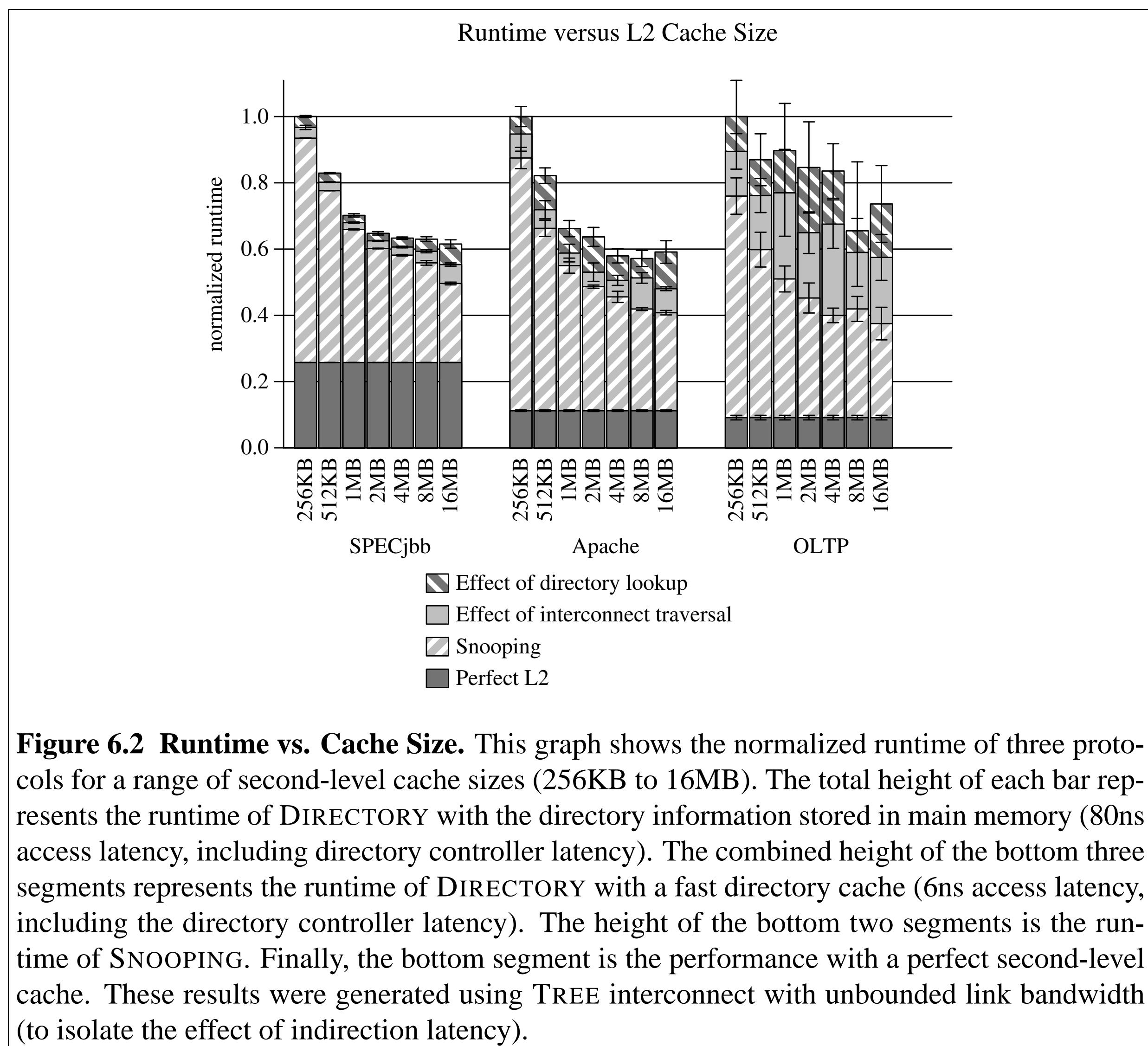


Figure 6.2 Runtime vs. Cache Size. This graph shows the normalized runtime of three protocols for a range of second-level cache sizes (256KB to 16MB). The total height of each bar represents the runtime of DIRECTORY with the directory information stored in main memory (80ns access latency, including directory controller latency). The combined height of the bottom three segments represents the runtime of DIRECTORY with a fast directory cache (6ns access latency, including the directory controller latency). The height of the bottom two segments is the runtime of SNOOPING. Finally, the bottom segment is the performance with a perfect second-level cache. These results were generated using TREE interconnect with unbounded link bandwidth (to isolate the effect of indirection latency).

proximately half of the runtime overhead. However, the relative importance of these two factors depends directly on the relative latencies of average interconnect latency and directory access latency.

Placing a directory looking and third interconnect traversal on the critical path of cache-to-cache misses has a significant impact on performance. For example, for the 4MB cache configuration with the TREE interconnect, eliminating the DRAM directory lookup using an SRAM directory results in a protocol that is 4–24% faster; eliminating only the interconnect traversal re-

sults in a protocol that is 4–69% faster, and eliminating both overheads results in a protocol that is 9–109% faster (*i.e.*, SNOOPING is 9–109% faster than DIRECTORY with a DRAM directory).

Although these performance differences seem large, a simple back-of-an-envelope calculation shows these speeds are reasonable. For example, consider OLTP. 66% of OLTP's misses are cache-to-cache misses. For SNOOPING on the TREE interconnect, memory-to-cache and cache-to-cache misses are 444 cycles and 296 cycles long, respectively (these latencies were presented in Table 6.1). The average miss latency for OLTP and SNOOPING is (296 x 0.66) + (444 x 0.34) = 346 cycles. For DIRECTORY using a DRAM directory on the TREE interconnect, the memory-to-cache miss latency is the same (444 cycles), but the cache-to-cache miss latency increases to 592 cycles. The average miss latency for OLTP and DIRECTORY is (592 x 0.66) + (444 x 0.34) = 542 cycles. In this example, DIRECTORY's average miss latency is 70% more than SNOOPING's miss latency. As these workloads spend a majority of their time in the memory system (shown by the large gap between perfect and non-perfect second-level caches), a 70% increase in average cache miss latency will have a significant effect on runtime.

6.4.4 The Bandwidth Cost of Broadcasting

Although directory protocols do suffer a performance penalty because of their use of indirection, the traffic-reduction benefits are substantial. The reduction in request traffic for directory protocols (and later our destination-set predictors) directly corresponds to the average number of processors that need to observe a request.

Figure 6.3 shows the percentage of misses that need to contact zero, one, or more than one processor. This graph reinforces that a substantial fraction of misses need to contact at least one processor (*i.e.*, are cache-to-cache misses). However, the percent of requests that the directory must forward to more than one other processor is extremely small (0.2–2.8%). This result is partially due to the significant fraction of read requests (the solid portion of each bar), because a read request needs to find *at most* one other processor: the owner. The small percentage of misses that are write requests (the striped portion of each bar) is partially due to (1) the migratory sharing optimization and (2) support for the EXCLUSIVE state (both of which are used in all our simulated protocols). As

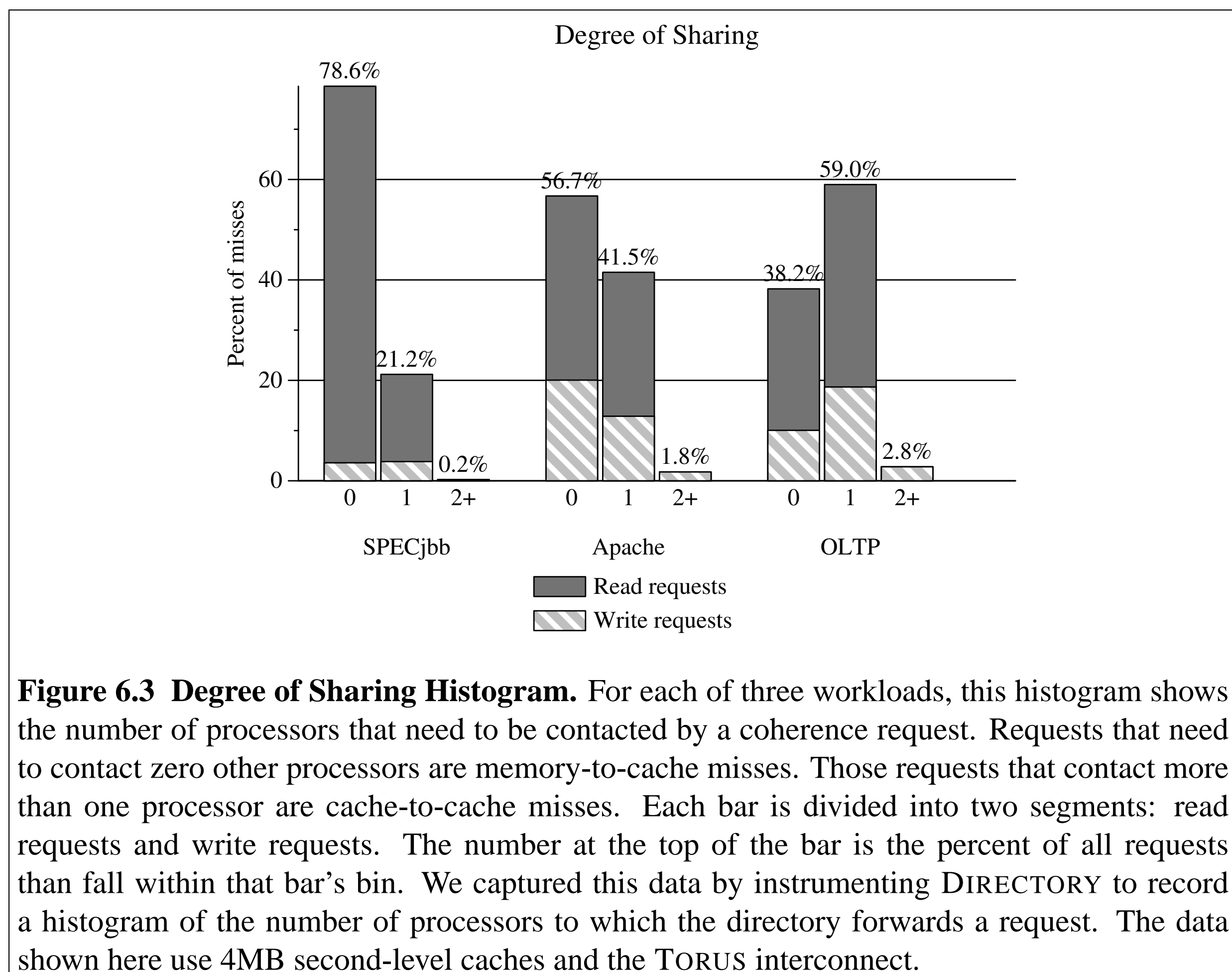


Figure 6.3 Degree of Sharing Histogram. For each of three workloads, this histogram shows the number of processors that need to be contacted by a coherence request. Requests that need to contact zero other processors are memory-to-cache misses. Those requests that contact more than one processor are cache-to-cache misses. Each bar is divided into two segments: read requests and write requests. The number at the top of the bar is the percent of all requests than fall within that bar's bin. We captured this data by instrumenting DIRECTORY to record a histogram of the number of processors to which the directory forwards a request. The data shown here use 4MB second-level caches and the TORUS interconnect.

part of evaluating the TOKENB performance policy, the next chapter further explores the relative amount of traffic consumed by SNOOPING and DIRECTORY in terms of both bytes per miss and coherence messages per miss.

This chapter presented our evaluation methods and a brief workload characterization focusing on the potential benefits of reducing cache-to-cache miss latency and avoiding broadcasts. In each of the next three chapters we describe and evaluate—using the methods described in this chapter–a Token Coherence performance policy.

Chapter 7

TOKENB: A Low-Latency Performance Policy Using Unordered Broadcast

Ideally, a coherence protocol would both avoid indirection latency for cache-to-cache misses (like snooping protocols) and not require any interconnect ordering (like directory protocols). One seemingly obvious approach is to directly send broadcasts on an unordered interconnect. This general approach has not been used, however, because it has previously suffered from numerous race cases that were difficult or impossible to make correct using traditional coherence techniques. With Token Coherence's flexibility and strong correctness guarantees, such an approach now becomes both feasible and attractive. The TOKENB performance policy follows this approach by broadcasting unordered transient requests to directly find the data in most cases, relying on the correctness substrate to guarantee correctness and prevent starvation. This chapter explores TOKENB's operation (Section 7.1) and performance (Section 7.2).

7.1 TOKENB Operation

The TOKENB performance policy uses three policies to avoid both interconnect ordering and indirection overheads.

7.1.1 Issuing Transient Requests

TOKENB broadcasts all transient requests (*i.e.*, it sends them to all processors and the home memory for the block). This policy works well (1) for moderate-sized systems in which interconnect bandwidth is plentiful and (2) when racing requests are rare.

7.1.2 Responding to Transient Requests

Components (processors and the home memory module) react to transient requests as they would in most MOESI protocols (as previously described in Section 2.2). A component with no tokens (INVALID) ignores all requests. A component with only non-owner tokens (SHARED) ignores transient read requests, but responds to a transient write request by sending all of its tokens in a dataless message (like an invalidation acknowledgment in a directory protocol). A component with the owner token but not all other tokens (OWNED) sends the data with one token (usually not the owner token) on a read request, and it sends the data and all of its tokens on an write request. A component with all the tokens (MODIFIED or EXCLUSIVE) responds in the same way as a component in OWNED, with the exception given in the next paragraph.

To optimize for common migratory sharing patterns, TOKENB implements a well-known optimization for migratory data (described earlier in Section 2.2.4 and illustrated in Table 2.4). If a processor with all tokens has written the block since it has received the block, it responds to read requests by sending data and all tokens (instead of the data and one token). We also implement an analogous optimization in all other protocols we compare against in the evaluation.

7.1.3 Reissuing Requests and Invoking Persistent Requests

If a transient request has not completed after a short timeout interval, the performance policy reissues the transient request. If the request has still not completed after an even longer interval, the processor invokes the persistent request mechanism. This approach allows the occasional race to be handled without the overhead of persistent requests, but yet it invokes persistent requests soon enough not to waste bandwidth and time reissuing transient requests many times. We use the distributed-arbitration approach for implementing persistent requests (previously described in Section 4.5).

To adjust to different interconnect topologies and congestion, the transient request timeout interval is set to twice the processor's average miss latency. Using twice the average miss latency prevents a slightly delayed response from causing a reissued request, but it also reissues quickly

enough to avoid to large a performance penalty for reissued requests. The persistent request interval is set to four times the average miss latency (for similar reasons). This policy adapts to the average miss latency of the system (to avoid reissuing too soon), but it also quickly reissues requests that do not succeed because of occasional races.

These intervals are tracked by a hardware counter that increments each cycle. The counter is reset (1) when a request is first issued and (2) when the request completes. When the counter reaches each of the two interval thresholds, it triggers its respective event. To prevent the estimate from growing without bound in pathological situations (*e.g.*, all requests are reissued), the estimate is capped at several times the uncontended interconnect's round-trip latency.

A processor can calculate the average latency of its recent misses by using a single unsigned accumulator and tracking the latency of each miss. The latency of each miss is easily recorded by sampling the value of the timer counter described in the last paragraph. The accumulator is initially set to an initial estimate (E_0), and each time a miss completes the new value of the accumulator is set to the weighted average of the old value in the accumulator and the most recent sample (*i.e.*, $E_n \equiv w \cdot L_n + (1 - w) \cdot E_{n-1}$, where E_n is the new estimate, E_{n-1} is the previous estimate, L_n is the most recent sample, and w is a constant between zero and one that controls the amount of hysteresis). This approach weighs recent samples more heavily in calculating the estimate while utilizing sufficient history to provide the desired hysteresis. Figure 7.1 shows that this approach results in a diminishing weighted average, and Figure 7.2 describes a hardware implementation of this algorithm that uses only bit selection, addition, and subtraction (but only when w is of the form $\frac{1}{2^k}$). In our experiments, w is $\frac{1}{2^8}$ ($\approx .0039$) and the initial estimate (E_0) is 500.

7.2 Evaluation of TOKENB

We answer seven questions as evidence that TOKENB can improve performance over snooping and directory protocols. Table 7.1 and Table 7.2 contain the raw data for TREE and TORUS, respectively, used to answer these questions.

Figure 7.1 A Diminishing Weighted Average. TokenB uses the following recurrence to calculate a diminishing-weight weighted average of the miss latency (E_n) using only a single accumulator:

$$\begin{aligned} E_0 &\equiv L_0 \\ E_n &\equiv wL_n + (1-w)E_{n-1}, n > 0 \end{aligned}$$

where w is a constant between zero and one, and L_n is the nth latency sample. This recurrence can be expressed as a summation by letting v be $1-w$ and repeatedly substituting for E_{n-1}:

$$\begin{aligned} E_n &= wL_n + v(wL_{n-1} + vE_{n-2}) \\ &= wL_n + wvL_{n-1} + v^2E_{n-2} \\ &= wL_n + wvL_{n-1} + wv^2L_{n-2} + v^3E_{n-3} \\ &= wv^0L_n + wvL_{n-1} + wv^2L_{n-2} + wv^3L_{n-3} + v^4E_{n-4} \\ &= \left(\sum_{i=0}^{n-1} wv^iL_{n-i}\right) + v^nL_0 \end{aligned}$$

This summation shows that this recurrence calculates a diminishing weighted sum of the miss latency samples. To show that this summation results in a proper weighted average, we next show that the sum of the weights is equal to one. We show this result below by (1) splitting the finite summation into the difference of two infinite geometric series summations, and (2) converting these summations using the closed form of the geometric series (the summation $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$, when $|x| < 1$). Since v is defined as $1-w$, the summation $\sum_{i=0}^{\infty} v^i$ is $\frac{1}{w}$.

$$\begin{aligned} \left(\sum_{i=0}^{n-1} wv^i\right) + v^n &= \left(\sum_{i=0}^{\infty} wv^i - \sum_{i=n}^{\infty} wv^i\right) + v^n \\ &= \left(\sum_{i=0}^{\infty} wv^i - v^n\sum_{i=0}^{\infty} wv^i\right) + v^n \\ &= \left(w\sum_{i=0}^{\infty} v^i - wv^n\sum_{i=0}^{\infty} v^i\right) + v^n \\ &= w\frac{1}{w} - wv^n\frac{1}{w} + v^n \\ &= 1 - v^n + v^n \\ &= 1 \end{aligned}$$

Figure 7.2 Implementing a Diminishing Weighted Average in Hardware. The recurrence for calculating a diminishing weighted average, illustrated in Figure 7.1, can be implemented efficiently in hardware by restricting w to be of the form $\frac{1}{2^k}$. The constant k determines the rate at which the weighting diminishes, effectively controlling the amount of hysteresis in the calculation. With this restriction, the recurrence for the diminishing weighted average can be rewritten as:

$$\begin{aligned} E_n &= wL_n + (1-w)E_{n-1} \\ &= \frac{1}{2^k}L_n + (1-\frac{1}{2^k})E_{n-1} \\ &= \frac{L_n}{2^k} + E_{n-1} - \frac{E_{n-1}}{2^k} \\ &= \frac{L_n + 2^k E_{n-1} - E_{n-1}}{2^k} \end{aligned}$$

Dividing an unsigned binary representation of a number by a power of two can be approximated with a right shift (using C-like notation, $\frac{n}{2^k} = n >> k$), and multiplication can be performed using a left shift ($n \cdot 2^k = n << k$). These properties allow us to rewrite the recurrence as:

$$E_n = (L_n + (E_{n-1} << k) - E_{n-1}) >> k$$

Thus, the system can calculate the new estimate of the average (E_n) using only shifts by a constant (implemented without logic by simply selecting the proper bits), addition, and subtraction. For increased precision, the accumulator (A_n) holds the pre-shifted value (*i.e.*, $A_n = E_n << k$):

$$\begin{aligned} A_n &= L_n + A_{n-1} - (A_{n-1} >> k) \\ E_n &= A_n >> k \end{aligned}$$

7.2.1 Question#1: Are reissued and persistent requests uncommon?

Answer: Yes; for our workloads, 97.5–99.5% of TOKENB's cache misses are issued only once. Since reissued requests are slower and consume more bandwidth than misses that succeed on the first attempt, reissued requests must be uncommon for TOKENB to perform well. Races are rare in our workloads, because—even though synchronization and sharing are common—multiple

configuration	all cycles per transaction	L2 misses per transaction	instructions per transaction	cycles per instruction	misses per thousand instructions	endpoint msgs per miss	interconnect bytes per miss
SPECjbb							
Perfect L2	19,903	NA	54,820	0.36	NA	NA	NA
TokenB	43,483	182	57,627	0.75	3.16	18.14	255.24
TokenNull	43,186	182	57,595	0.75	3.16	34.04	382.01
Snooping	44,941	185	57,884	0.78	3.20	20.63	274.94
Apache							
Perfect L2	176,559	NA	273,711	0.65	NA	NA	NA
TokenB	689,765	3,949	361,509	1.90	10.86	18.13	233.58
TokenNull	690,699	3,968	364,189	1.90	10.87	33.96	358.98
Snooping	718,417	4,041	401,851	1.79	10.10	19.32	242.85
OLTP							
Perfect L2	1,747,629	NA	3,385,566	0.52	NA	NA	NA
TokenB	7,931,573	45,233	7,375,716	1.08	6.09	18.36	221.38
TokenNull	7,958,491	45,936	7,742,584	1.08	6.30	33.70	342.34
Snooping	7,656,243	45,707	7,079,692	1.09	6.40	18.04	218.72

Table 7.1 TOKENB Results for the TREE Interconnect. The metrics presented in this table were described in Table 6.2 on page 101.

processors rarely access the same data simultaneously due to the large amount of shared data. Table 7.3 shows the percentage of all TOKENB misses that are not reissued, are reissued once, and that invoke persistent requests. For our workloads, only 0.2–1.6% of cache misses are reissued once and only 0.2–1.0% resort to persistent requests. (Table 7.3 shows TORUS interconnect results, but TREE results, not shown, are similar.)

To quantify the latency and bandwidth costs of reissued and persistent requests, we created a new performance policy, TOKENNULL. The TOKENNULL performance policy does nothing; it simply waits for the correctness substrate to issue a persistent request. As we use the low-latency distributed-arbitration persistent request mechanism (Section 4.5), the miss latency of TOKENNULL is actually quite good: the uncontended latency of a transient and persistent request is the same.

configuration	all cycles per transaction	L2 misses per transaction	instructions per transaction	cycles per instruction	misses per thousand instructions	endpoint msgs per miss	interconnect bytes per miss
SPECjbb							
Perfect L2	19,903	NA	54,820	0.36	NA	NA	NA
TokenB	36,604	180	56,911	0.64	3.17	18.20	376.51
Directory - fast	39,307	182	57,201	0.69	3.20	6.01	319.32
Directory - slow	41,102	182	57,270	0.72	3.18	6.01	318.37
HammerOpt	39,504	182	57,181	0.69	3.18	35.56	682.82
Apache							
Perfect L2	176,559	NA	273,711	0.65	NA	NA	NA
TokenB	534,197	3,833	339,043	1.56	11.15	18.16	332.93
Directory - fast	609,117	4,017	367,030	1.66	10.89	5.36	263.93
Directory - slow	702,715	4,063	390,853	1.78	10.31	5.35	263.71
HammerOpt	614,838	3,981	354,503	1.71	11.16	34.37	619.88
OLTP							
Perfect L2	1,747,629	NA	3,385,566	0.52	NA	NA	NA
TokenB	5,727,614	42,412	5,802,205	0.99	7.28	18.36	306.82
Directory - fast	7,301,596	47,189	7,000,286	1.00	6.40	4.64	227.65
Directory - slow	9,463,958	48,304	11,557,401	0.85	4.32	4.65	227.30
HammerOpt	7,640,467	47,538	8,281,252	0.91	5.53	33.19	576.64

Table 7.2 TOKENB Results for the TORUS Interconnect. The metrics presented in this table were described in Table 6.2 on page 101.

	SPECjbb			Apache			OLTP		
	0	1	P.R.	0	1	P.R.	0	1	P.R.
TokenB	99.5	0.2	0.3	99.1	0.7	0.2	97.5	1.6	1.0

Table 7.3 TOKENB Reissued Requests. This table shows the percent of cache misses that succeed on the first transient request and thus are not reissued (the “0” column), are reissued once and succeed on this second transient request (the “1” column), and those that invoke a persistent request (the “P.R.” column).

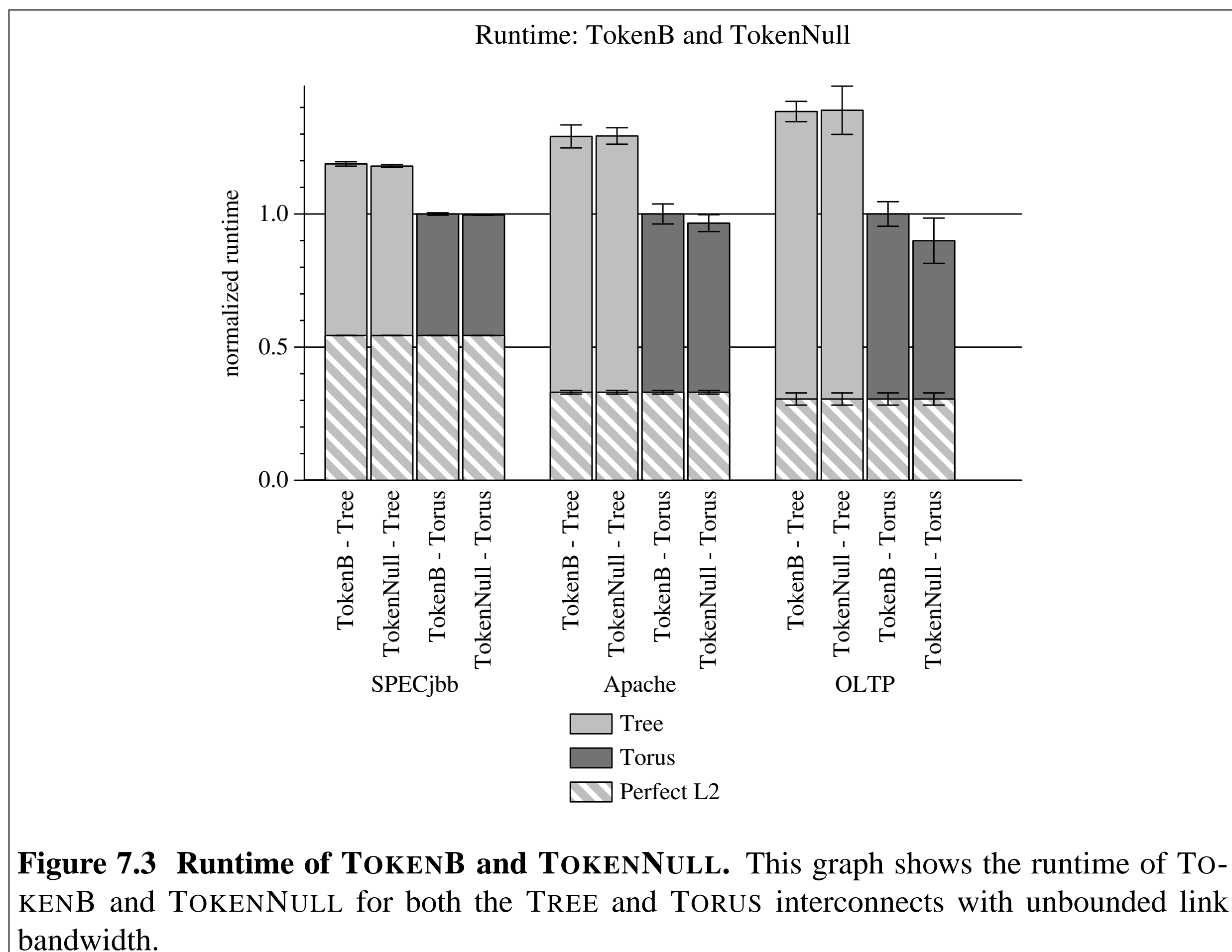


Figure 7.3 Runtime of TOKENB and TOKENNULL. This graph shows the runtime of TOKENB and TOKENNULL for both the TREE and TORUS interconnects with unbounded link bandwidth.

For the simulations of TOKENNULL we also make two changes to the correctness substrate. First, we change the substrate to issue persistent requests immediately (*i.e.*, it does not wait for a timeout). Second, we extend the persistent request mechanism to support multiple outstanding persistent requests per processor. This change greatly increases the persistent request table size, but without this change TOKENNULL's performance would be artificially limited by a single outstanding miss (because TOKENNULL only uses persistent requests).

Figure 7.3 shows that TOKENB and TOKENNULL perform similarly for two of the three workloads on both interconnects (this graph plots runtime with unbounded bandwidth to separate the effect of contention-free latency from interconnect contention). For OLTP, TOKENNULL is perhaps faster than TOKENB, although the large error bars make determining the exact magnitude

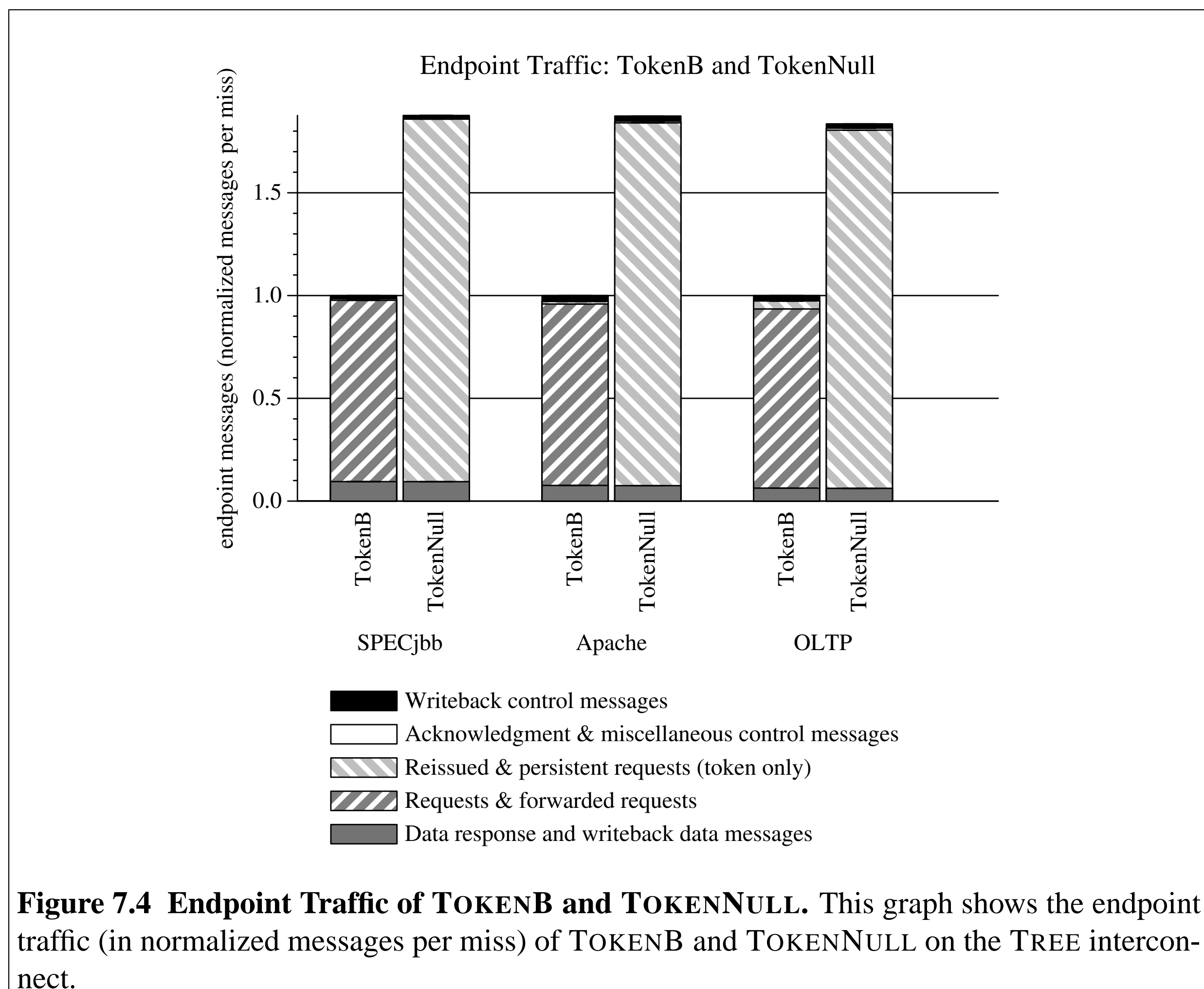


Figure 7.4 Endpoint Traffic of TOKENB and TOKENNULL. This graph shows the endpoint traffic (in normalized messages per miss) of TOKENB and TOKENNULL on the TREE interconnect.

difficult. OLTP's largest difference in performance among the three workloads is consistent with its highest rate of reissued and persistent requests of these workloads.

The generally small difference in performance between TOKENNULL and TOKENB is important for two reasons. First, it shows that TOKENB is not suffering an experimentally-significant performance loss due to reissued requests. Second, it shows that this implementation of persistent requests has the same low latency as transient requests.

In contrast, the traffic of TOKENNULL is substantially higher than TOKENB because of the additional overhead of persistent requests due to deactivation messages. The implementation of persistent requests used in this evaluation avoids persistent request acknowledgments (as described

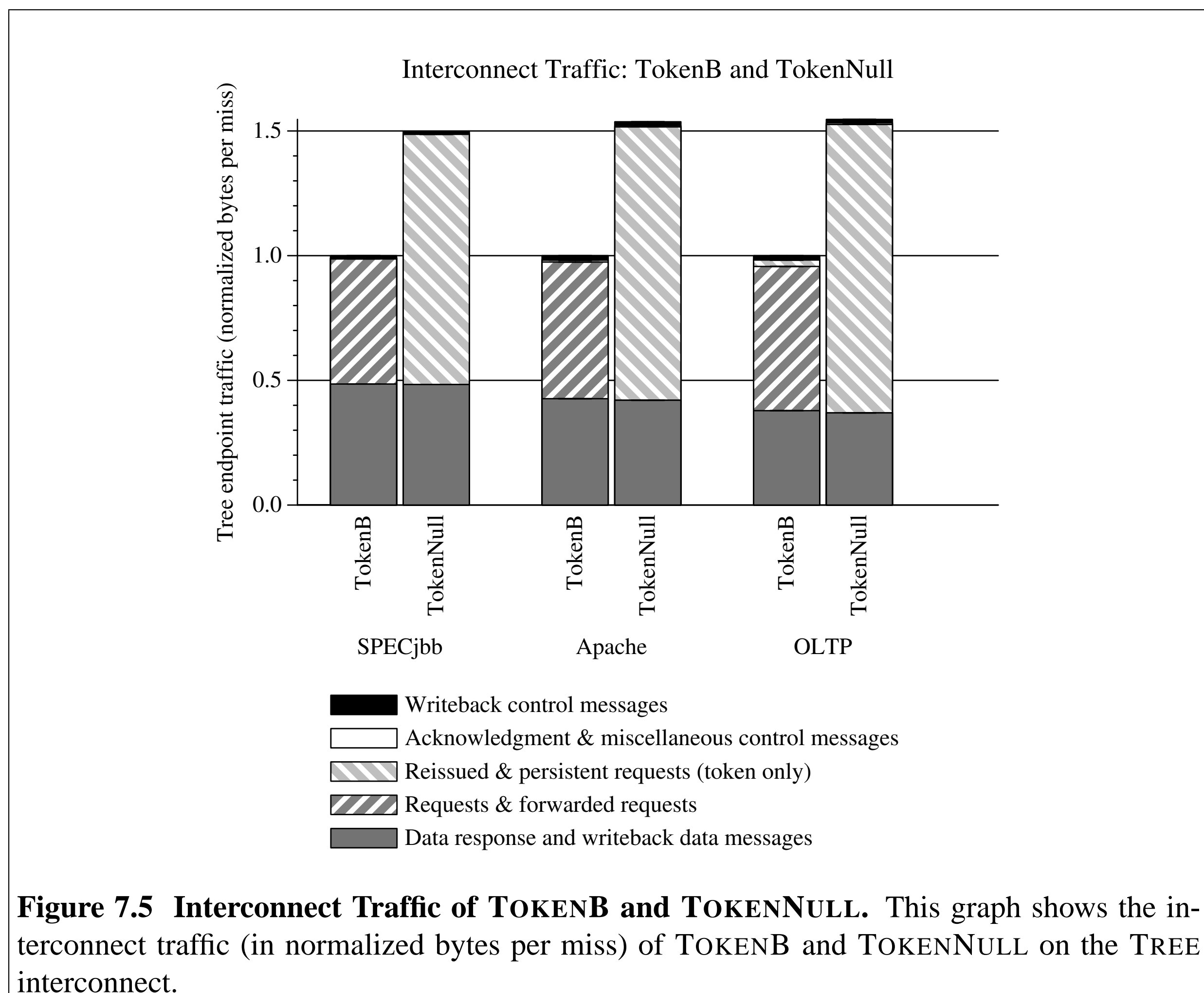


Figure 7.5 Interconnect Traffic of TOKENB and TOKENNULL. This graph shows the interconnect traffic (in normalized bytes per miss) of TOKENB and TOKENNULL on the TREE interconnect.

in Section 4.7.2). This implementation results in persistent requests that use twice the bandwidth (an activation and a deactivation versus only a transient request). In terms of *endpoint messages per miss* delivered to system components, TOKENNULL's traffic is 84–87% more than TOKENB (as shown in Figure 7.4). In terms of interconnect link *bytes per miss* delivered to system components, TOKENNULL's traffic is 32–36% more then TOKENB (as shown in Figure 7.5). The interconnect traffic overhead is lower because 72-byte data messages are much larger than 8-byte request messages (as illustrated by the increase in size of the solid grey segment from Figure 7.4 to Figure 7.5.

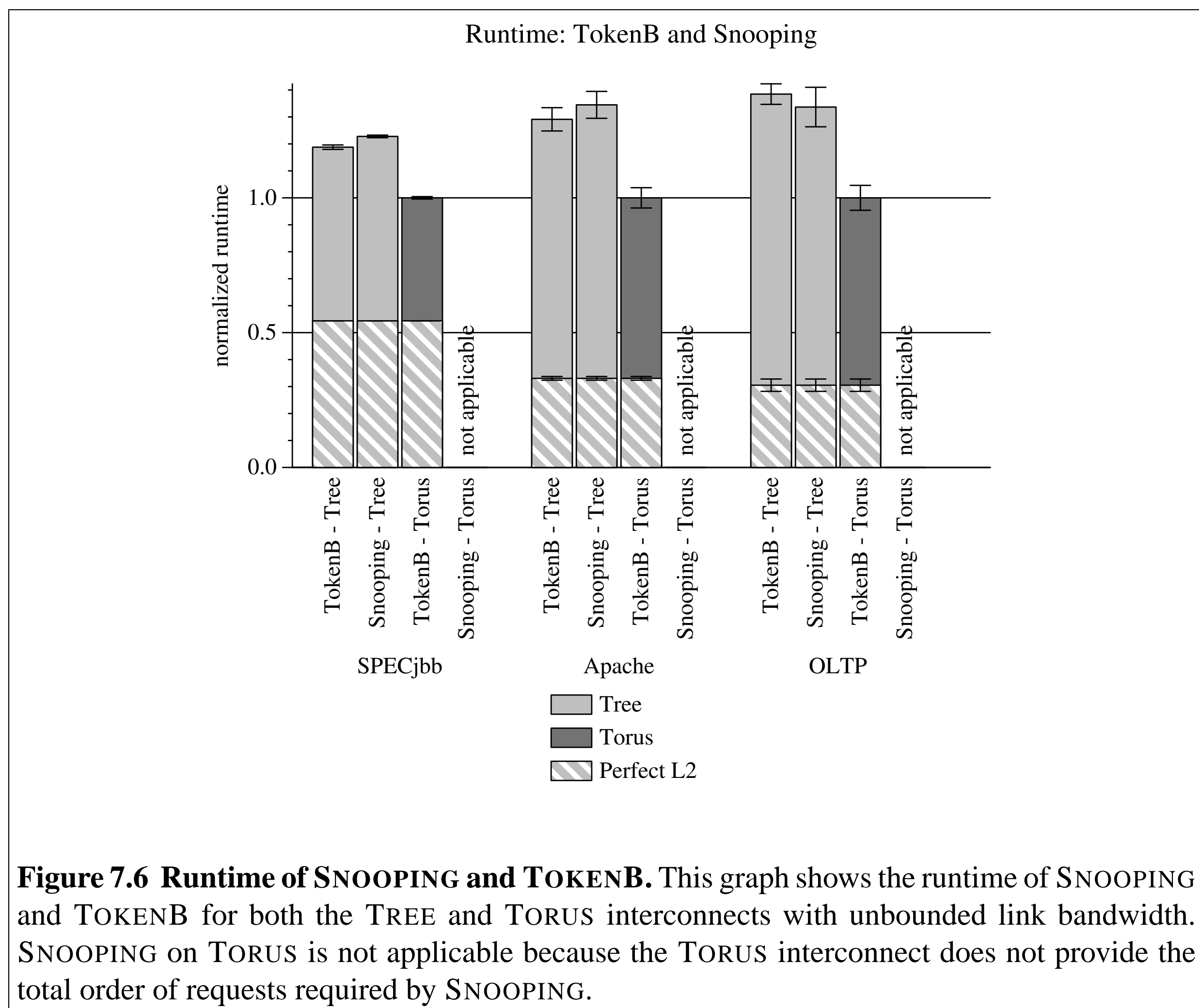


Figure 7.6 Runtime of SNOOPING and TOKENB. This graph shows the runtime of SNOOPING and TOKENB for both the TREE and TORUS interconnects with unbounded link bandwidth. SNOOPING on TORUS is not applicable because the TORUS interconnect does not provide the total order of requests required by SNOOPING.

7.2.2 Question#2: Can TOKENB outperform SNOOPING?

Answer: Yes; with the same interconnect, TOKENB and SNOOPING perform similarly for our workloads; however, by exploiting the lower-latency unordered TORUS, TOKENB on the TORUS is faster than SNOOPING on the TREE interconnect (23–34% faster). Figure 7.6 shows the normalized runtime (smaller is better) of TOKENB on the TREE and TORUS interconnects and SNOOPING on the TREE interconnect. SNOOPING on the TORUS is not applicable, because the TORUS does not provide the required total order of requests.

On the TREE interconnect, SNOOPING's performance is similar to TOKENB (as shown in Figure 7.6).[1] However, since SNOOPING requires a totally-ordered interconnect, only TOKENB can exploit a lower-latency unordered interconnect. Thus, by using the TORUS, TOKENB is 23–34% faster with unlimited bandwidth links (Figure 7.6), and 47–82% faster than SNOOPING on TREE with limited bandwidth links (not shown). This speedup results from (1) lower latency for all misses (cache-to-cache or otherwise) due to lower average interconnect latency, and (2) lower contention in TORUS (by avoiding TREE's central-root bottleneck).

7.2.3 Question#3: Is TOKENB's traffic similar to SNOOPING?

Answer: Yes; to the first order, both TOKENB's endpoint traffic and interconnect traffic are similar to or less than SNOOPING. Figure 7.7 shows the endpoint traffic (in normalized messages per miss received at each endpoint coherence controller), and Figure 7.8 shows the interconnect traffic (in normalized bytes per miss). When considering only data and non-reissued request traffic, TOKENB and SNOOPING are practically identical. TOKENB adds some additional traffic overhead (as shown by the light grey striped segment, only visible for OLTP), but the overhead is small for all three of our workloads. SNOOPING and TOKENB both use additional traffic for writeback control messages (the solid black segment), but due to detailed implementation decisions in SNOOPING involving writeback acknowledgment messages, SNOOPING uses more traffic for writebacks than TOKENB. SNOOPING sends a writeback request on the ordered interconnect to both the memory and to itself as a marker message. If it is still the owner of the block, it receives the marker message and sends the data back to the memory. Ignoring this specific implementation overhead leads us to the conclusion that these protocols generate similar amounts of traffic.

As TORUS is faster and has more effective bandwidth then TREE, the remainder of the results in this dissertation use the TORUS interconnect to compare protocols.

[1] Although TOKENB and SNOOPING have the same uncontended miss latencies, TOKENB is actually slightly faster then SNOOPING for SPECjbb. Limited investigation of this phenomenon indicates it may be caused by SNOOPING's handling of incoming requests while in certain transient states. SNOOPING refuses to process an incoming request for a block for which (1) it has received its own marker message, and (2) it is waiting a data response. As SNOOPING processes incoming requests in order (to maintain both consistency and a total order of requests), this situation delays all requests—including requests for other blocks.

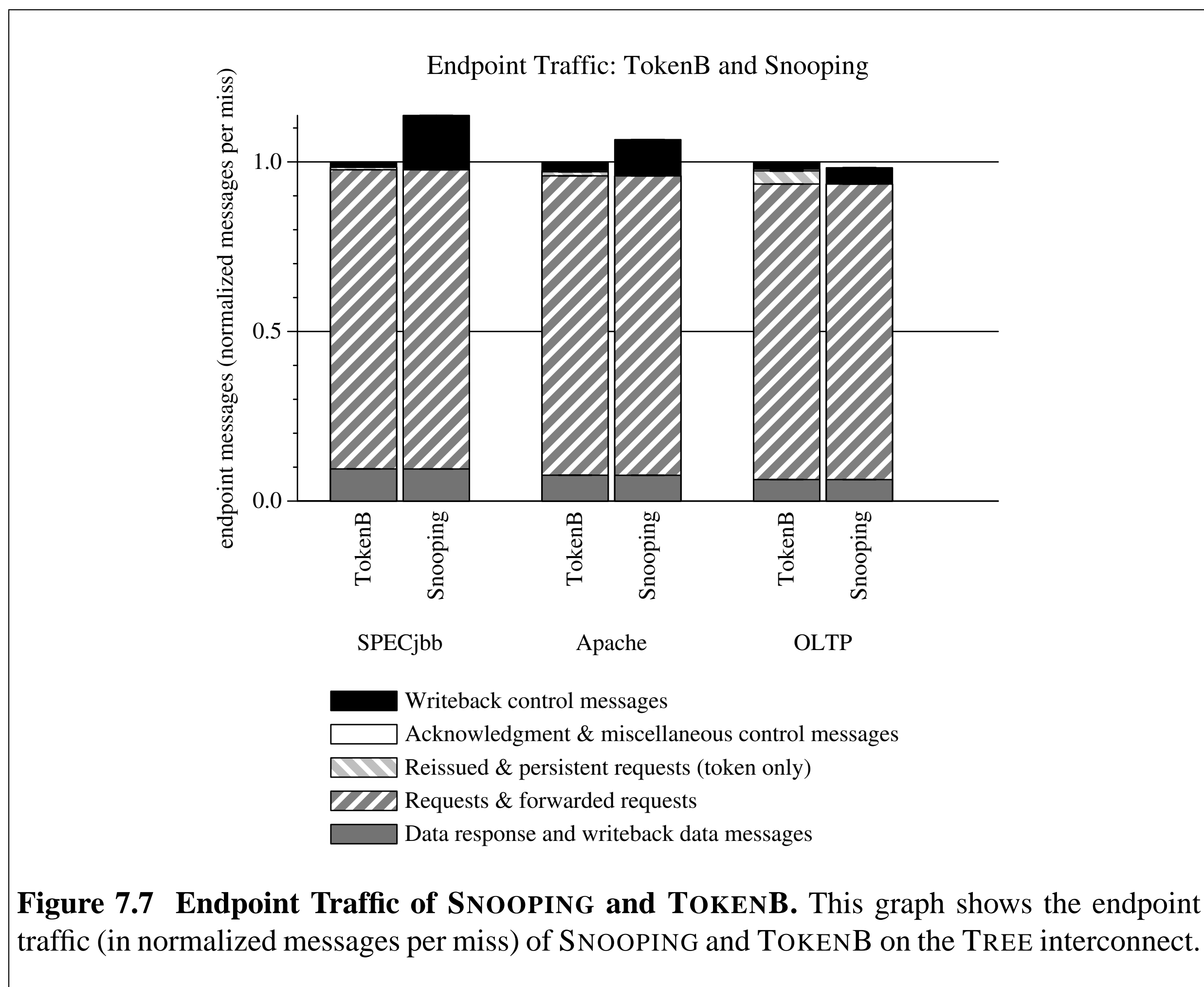


Figure 7.7 Endpoint Traffic of SNOOPING and TOKENB. This graph shows the endpoint traffic (in normalized messages per miss) of SNOOPING and TOKENB on the TREE interconnect.

7.2.4 Question#4: Can TOKENB outperform DIRECTORY or HAMMEROPT?

Answer: Yes; by removing the directory lookup latency and interconnect traversal from the critical path of cache-to-cache misses, TOKENB is faster than both DIRECTORY and HAMMEROPT (12–65% and 8–33% faster, respectively). Figure 7.9 shows the normalized runtime (smaller is better) for TOKENB, HAMMEROPT, and DIRECTORY on the TORUS interconnect with unbounded link bandwidth. The solid grey bar for DIRECTORY illustrates the runtime increase due to the DRAM directory lookup latency.

TOKENB is faster than DIRECTORY and HAMMEROPT because it (1) avoids the third interconnect traversal for cache-to-cache misses, (2) avoids the directory lookup latency (DIRECTORY

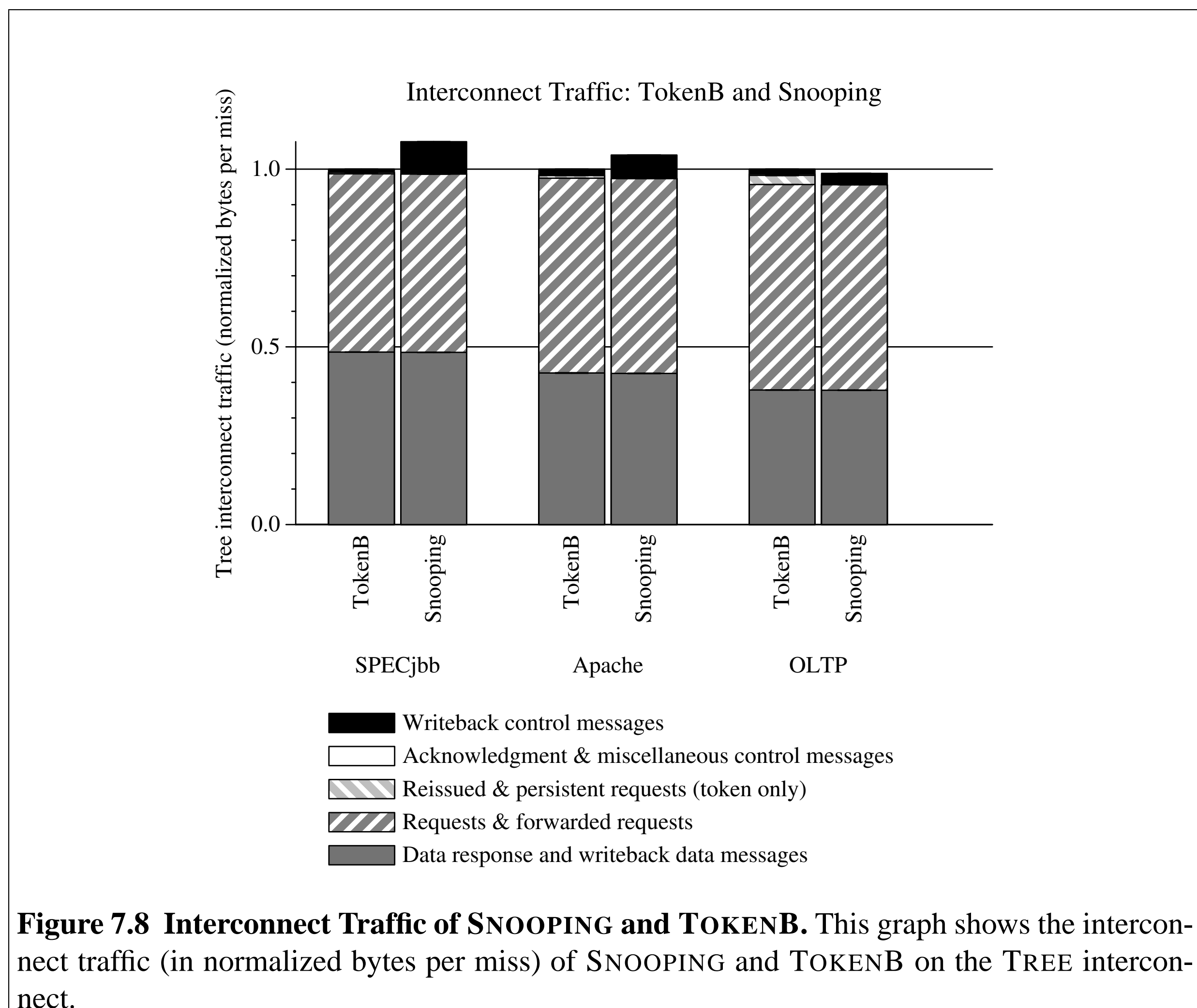


Figure 7.8 Interconnect Traffic of SNOOPING and TOKENB. This graph shows the interconnect traffic (in normalized bytes per miss) of SNOOPING and TOKENB on the TREE interconnect.

only), and (3) removes blocking states in the memory controller. Even if the directory lookup latency is reduced to 6ns (to approximate a fast SRAM directory or directory cache), shown by disregarding the solid grey bar in Figure 7.9, TOKENB is still faster than DIRECTORY by 7–27%. HAMMEROPT is 4–24% faster than DIRECTORY with a DRAM directory, because HAMMEROPT avoids the directory lookup latency (but not the third interconnect traversal). DIRECTORY with the fast directory latency has similar performance as HAMMEROPT.

We also simulated TOKENB on the TORUS interconnect with 4GB/s link bandwidth (not shown), and we found that for this 16-processor system, the performance impact of TOKENB's additional traffic is not statistically significant for these workloads. The impact is negligible be-

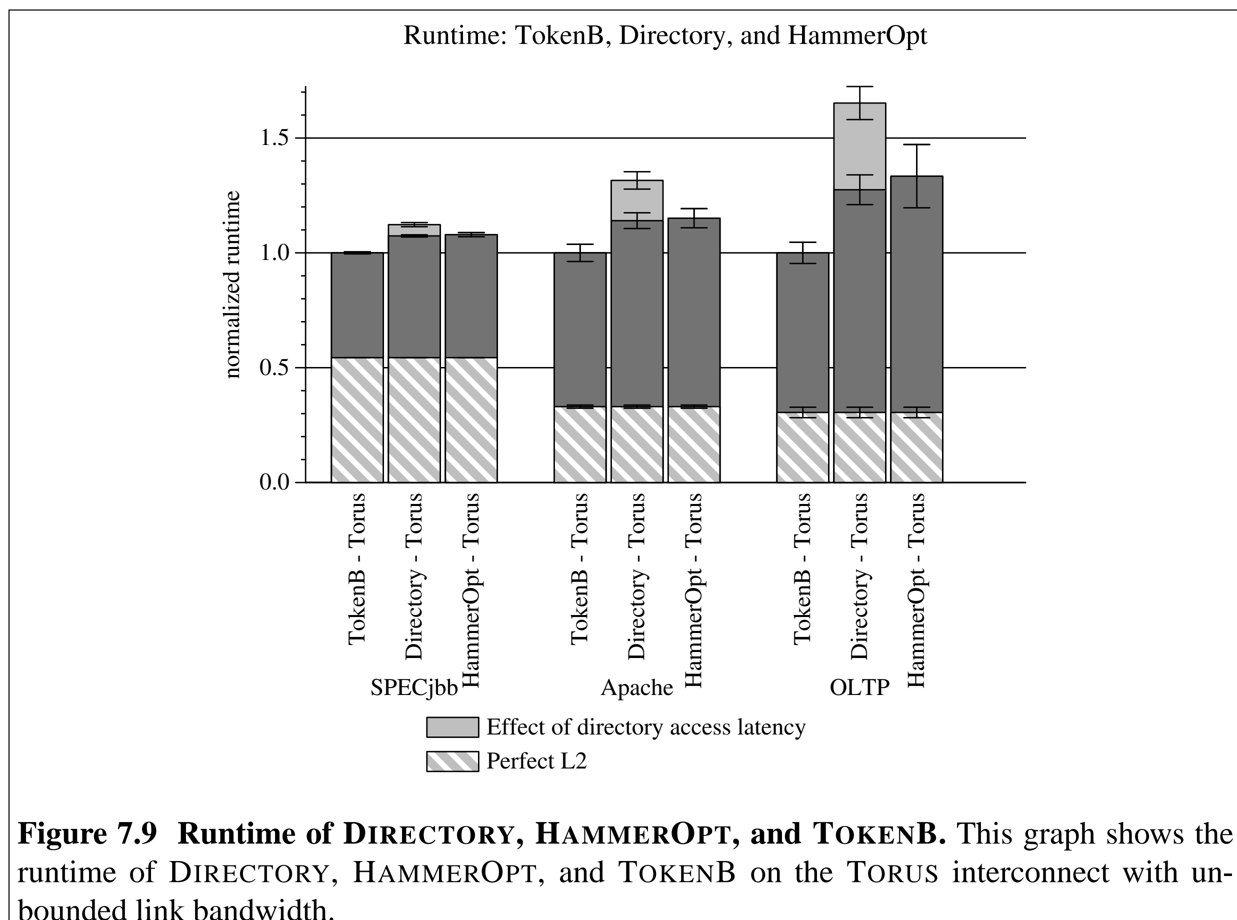


Figure 7.9 Runtime of DIRECTORY, HAMMEROPT, and TOKENB. This graph shows the runtime of DIRECTORY, HAMMEROPT, and TOKENB on the TORUS interconnect with unbounded link bandwidth.

cause (1) the TORUS interconnect has sufficient bandwidth due to high-speed point-to-point links, and (2) the additional traffic of TOKENB is moderate, discussed next.

7.2.5 Question#5: How does TOKENB's traffic compare to DIRECTORY and HAMMEROPT?

Answer: TOKENB generates less endpoint traffic and interconnect traffic than HAMMEROPT, but it generates more traffic than DIRECTORY. Figure 7.10 shows a traffic breakdown in normalized endpoint messages per miss (smaller is better) for TOKENB, HAMMEROPT, and DIRECTORY. Figure 7.11 shows traffic in terms of interconnect traffic in bytes per miss (smaller is better) for the same three protocols. The dark segment at the top of each bar is the traffic due to writeback control messages, and this segment is larger on HAMMEROPT and DIRECTORY due to their use of three-

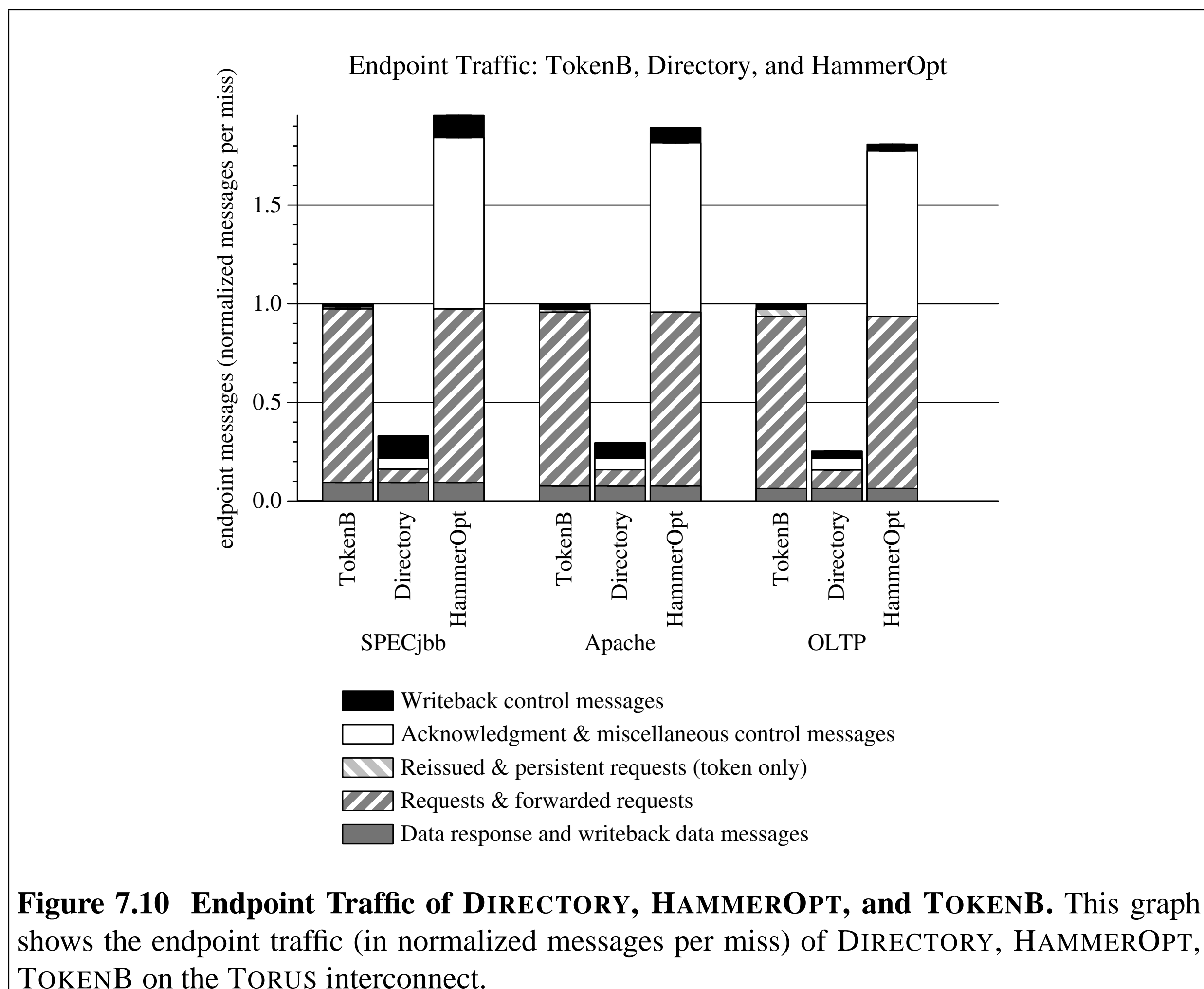


Figure 7.10 Endpoint Traffic of DIRECTORY, HAMMEROPT, and TOKENB. This graph shows the endpoint traffic (in normalized messages per miss) of DIRECTORY, HAMMEROPT, TOKENB on the TORUS interconnect.

phase writebacks. These figures show: (1) that HAMMEROPT generates *more* traffic than TOKENB (81–95% more endpoint traffic and 81–88% more interconnect traffic), and (2) that DIRECTORY generates *less* traffic than TOKENB (67–75% less endpoint traffic and 15–26% less interconnect traffic).

The extra interconnect traffic of TOKENB over DIRECTORY is not as large as one might expect (only 18–35% more), because (1) both protocols send a similar number of 72-byte data messages (more than 80% of DIRECTORY's interconnect traffic), (2) request messages are small (8 bytes), and (3) TORUS supports broadcast tree routing (as stated in Section 6.2.2 and described in Section 2.3.2). HAMMEROPT, which targets smaller systems, uses much more bandwidth than TO-

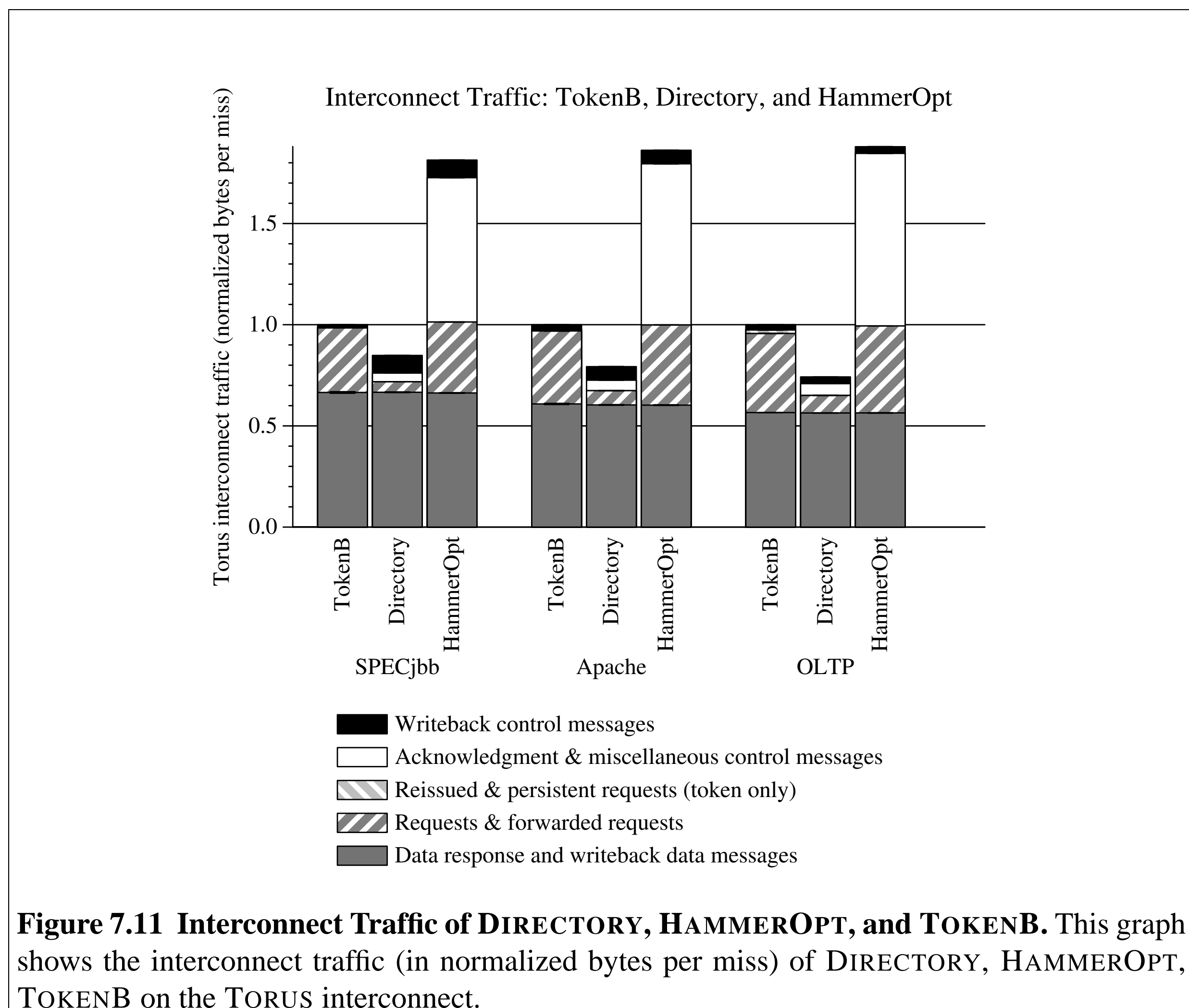


Figure 7.11 Interconnect Traffic of DIRECTORY, HAMMEROPT, and TOKENB. This graph shows the interconnect traffic (in normalized bytes per miss) of DIRECTORY, HAMMEROPT, TOKENB on the TORUS interconnect.

KENB or DIRECTORY, because every processor acknowledges each request (shown by the white segment). The acknowledgment messages generate more interconnect traffic than the broadcast requests, because n unicasts on the TORUS interconnect use $\Theta(n\sqrt{n})$ links; a broadcast uses only $\Theta(n)$ links.

Although TOKENB's greater use of interconnect bandwidth (when compared with DIRECTORY) is modest (DIRECTORY uses 15–26% less interconnect traffic than TOKENB), TOKENB's endpoint traffic is significantly higher than DIRECTORY (DIRECTORY uses 67–75% less endpoint traffic). As a result, TOKENB requires significantly higher bandwidth coherence controllers. Fortunately, researchers have proposed several techniques for creating high-bandwidth and low-power

state	DIRECTORY	TOKENB	SPECjbb	Apache	OLTP
MODIFIED	writeback	writeback	84%	48%	20%
OWNED	writeback	writeback	3%	10%	8%
EXCLUSIVE	eviction notification	eviction notification	6%	6%	4%
SHARED	silent eviction	eviction notification	6%	23%	25%
INVALID	null	null	0%	14%	43%

Table 7.4 Distribution of Evictions per State. This table shows the distribution of states for blocks evicted from the second-level cache. The first column is the state of the evicted block, in which INVALID means the block is either not present or invalid. The second the third columns indicate the action required to evict a block in that state for DIRECTORY and TOKENB protocols, respectively. The final three columns show the percentage of evictions for our three workloads for each state.

coherence controllers (*e.g.*, [89, 93, 103]), and these techniques can be readily applied to Token Coherence. Since Token Coherence does not rely upon a total order of requests—in contrast with traditional snooping protocols—the coherence controllers can be aggressively banked by address with low complexity (without the worry of synchronizing the banks to prevent subtle memory consistency model violations).

7.2.6 Question#6: How frequently do non-silent evictions occur?

Answer: Table 7.4 shows that TOKENB generates more eviction messages due to non-silent replacements than DIRECTORY. For our workloads, TOKENB initiated 7–80% more non-silent evictions that DIRECTORY. These additional eviction messages increase interconnect and memory controller traffic. However, when the size of the messages is taken into account, the additional traffic in bytes is only 1–10% greater, because data messages are much larger than token-only messages. When the total system traffic is considered, the overhead is even smaller.

7.2.7 Question#7: Does TOKENB scale to an unlimited number of processors?

Answer: No; TOKENB relies on broadcast, limiting its scalability. TOKENB is more scalable than HAMMEROPT, because HAMMEROPT uses broadcast and many acknowledgment messages. TOKENB is less scalable than DIRECTORY, because DIRECTORY avoids broadcast. A simple

Figure 7.12 An Analytical Model of the Traffic of TOKENB and DIRECTORY. We use the following analytical model of the traffic of TOKENB and DIRECTORY to explore the relative traffic generated by these protocols over a range of processors. To simplify the model, we focus on a simple memory-to-cache miss and ignore writeback data and control messages. This model has several parameters:

- n is the number of processors in the system. The cost of a unicast point-to-point message on TORUS is $\frac{1}{2}\sqrt{n}$ link crossings on average. A broadcast on TORUS crosses $n-1$ links.
- C is the size of control messages such as request and acknowledgment messages. C is 8 bytes for our simulations and for this model.
- D is the size of data messages such as data responses. D is 72 bytes (64-byte datablock plus a 8-byte header) for our simulations and for this model.
- p is the fraction of misses that invoke persistent requests. As a simplification, this model assumes that a persistent request is invoked after the first transient request fails to complete (*i.e.*, transient requests are not reissued). We assume this rate is constant as system size increases.
- $E_{protocol}$ is the endpoint traffic in messages per miss per endpoint for *protocol*.
- $I_{protocol}$ is the TORUS interconnect traffic in bytes per miss per link for *protocol*.

$$E_{TokenB} \equiv \overbrace{1 \cdot \underbrace{(n-1)}_{broadcast}}^{transient\ request} + \overbrace{p \cdot \underbrace{(n-1)}_{broadcast}}^{persistent\ request} + \overbrace{1 \cdot \underbrace{1}_{unicast}}^{data\ response}$$

$$E_{Directory} \equiv \overbrace{1 \cdot \underbrace{1}_{unicast}}^{request} + \overbrace{1 \cdot \underbrace{1}_{unicast}}^{data\ response} + \overbrace{1 \cdot \underbrace{1}_{unicast}}^{completion} \equiv 3$$

$$I_{TokenB} \equiv \overbrace{C \cdot \underbrace{(n-1)}_{broadcast}}^{transient\ request} + \overbrace{C \cdot p \cdot \underbrace{(n-1)}_{broadcast}}^{persistent\ request} + \overbrace{D \cdot \underbrace{\frac{1}{2}\sqrt{n}}_{unicast}}^{data\ response}$$

$$I_{Directory} \equiv \overbrace{C \cdot \underbrace{\frac{1}{2}\sqrt{n}}_{unicast}}^{request} + \overbrace{D \cdot \underbrace{\frac{1}{2}\sqrt{n}}_{unicast}}^{data\ response} + \overbrace{C \cdot \underbrace{\frac{1}{2}\sqrt{n}}_{unicast}}^{completion} \equiv \frac{1}{2}(D+2C)\sqrt{n}$$

The above definitions calculate the traffic caused by a single miss on the assumption that a linearly-increasing system-wide miss rate and a linearly-increasing number of interconnect links and endpoints offset each other.

n	TOKENB v. DIRECTORY, $p = 0.05$
4	1.1x
8	1.3x
16	1.5x
32	1.9x
64	2.3x
128	3.0x
256	3.9x
512	5.1x

Table 7.5 Results from an Analytical Model of Traffic: TOKENB versus DIRECTORY. This table shows the increase in interconnect traffic on TOKENB over DIRECTORY on the TORUS interconnect using the simple analytical model of traffic described in Figure 7.12. The left column is the number of processors. The right column is the amount of traffic used by TOKENB over DIRECTORY expressed as a traffic multiple (*i.e.*, 2.0x means TOKENB uses twice as much bandwidth as DIRECTORY). For these results, the persistent request rate (p) is 0.05 (5%).

analytical model (described in Figure 7.12) indicates that the endpoint bandwidth of TOKENB increases linearly as the number of processors increases. The interconnect traffic difference between TOKENB and DIRECTORY increases more slowly ($\Theta(\sqrt{n})$). As shown in Table 7.5, for a 64-processor system, this model predicts TOKENB will use 2.3 times the interconnect bandwidth of DIRECTORY on the TORUS interconnect. A 128-processor system would use 3 times the bandwidth. Thus, TOKENB can perform well for perhaps 32 or 64 processors if bandwidth is abundant (by using high-bandwidth links and high-throughput coherence controllers). However, TOKENB is a poor choice for larger or more bandwidth-limited systems.

7.2.8 TOKENB Results Summary

TOKENB is both (1) better than SNOOPING and (2) faster then DIRECTORY when bandwidth is plentiful. TOKENB is better than SNOOPING because it uses similar amounts of traffic and can outperform SNOOPING by exploiting a faster, unordered interconnect. As discussed in Chapter 1, such interconnects may also provide high bandwidth more cheaply by avoiding dedicated switch chips. TOKENB is faster than DIRECTORY in bandwidth-rich situations by avoiding placing directory lookup latency and a third interconnect traversal on the critical path of common cache-to-cache

misses. However, for small systems, TOKENB uses a moderate amount of additional interconnect traffic and significantly more endpoint message bandwidth than DIRECTORY. Thus, in a bandwidth constrained situation, DIRECTORY will outperform TOKENB. Although TOKENB is a message-intensive protocol, it is only one of many possible performance policies; Chapter 8 and Chapter 9 present more bandwidth-efficient performance policies that do not always rely on broadcast.

Chapter 8

TOKEND: A Directory-Like Performance Policy

The last chapter described TOKENB and showed that it performs well when bandwidth is plentiful, but TOKENB can use significantly more bandwidth than a directory protocol. In contrast, TOKEND's goal is to exhibit similar performance and traffic as a traditional directory protocol (*e.g.*, DIRECTORY). By achieving this goal, TOKEND (1) shows that Token Coherence is a suitable framework for bandwidth-efficient protocols, and (2) provides a foundation for TOKENM, our predictive hybrid protocol described in the next chapter. This chapter explores TOKEND's operation (Section 8.1), implementation (Section 8.2), and performance (Section 8.3).

8.1 TOKEND's Operation

The TOKEND performance policy uses transient requests to emulate a directory protocol. In a directory protocol, processors send their requests to the directory located at the home memory module. This directory is responsible for forwarding read requests to the current owner and forwarding write requests to the current owner and all sharers. TOKEND operates similarly: processors send transient requests to a directory at the home memory module only. This directory forwards the transient request to one or more processors that are likely to be holding tokens (*i.e.*, it forwards read requests to the processor that is likely holding the owner token and forwards write requests to any processor that is likely holding a token). Processors respond to these forwarded transient requests in a traditional MOESI-like fashion, sending tokens and data directly to the requester (using the same policy as TOKENB, including the migratory sharing optimization). This

basic policy of forwarding transient requests closely imitates a directory protocol, giving TOKEND much of the same first-order performance characteristics.

Transient requests in TOKEND may fail to complete (*e.g.*, when forwarded transient requests are reordered or a request is forwarded to an insufficient set of processors). To handle the occasional failure of transient requests, TOKEND borrows TOKENB's policy for reissuing transient requests: the processor reissues transient requests once (after twice the average miss latency) and the processor invokes a persistent request (after four times the average miss latency). In contrast to TOKENB, in TOKEND all transient requests (even reissued ones) are sent only to the home memory module.

In TOKEND, how does the memory module know to which processors it should forward the request? Much like a directory protocol, TOKEND uses per-block state at memory to decide to which processors (if any) it should forward the request; however, unlike the count of tokens held at memory, TOKEND's per-block information for forwarding requests is *soft state* (*i.e.*, it is not required to be accurate). In contrast, all the per-block state in a traditional directory protocol is *hard state* that must correctly identify a superset of the sharers for the protocol to function correctly. TOKEND's operation is always correct—even when the directory information is inaccurate—because (1) the token counting rules prevent unsafe behavior[1] and (2) the processors eventually invoke persistent requests to avoid starvation. Although the directory-like soft state should reflect the actual state of the system as accurately as possible for performance reasons, it is not required to be correct.

The lack of correctness requirements of the soft-state directory in TOKEND grants significant design flexibility. Unlike many directory protocols that use blocking or busy states to prevent ambiguous concurrent requests, TOKEND has no such requirement because it can rely on the correctness substrate to correctly handle these situations. For example, TOKEND can make use of a directory cache that simply evicts entries without needing to worry about recall messages or writing the information to a full-sized directory (directory caches and associated implementation options were described in Section 3.2.4). Verifying traditional directory structures can be difficult, especially when using non-trivial coarse vectors or other efficient encodings (*e.g.*, [8, 13, 45, 52,

[1]The token counts are hard state.

97]). In contrast, verifying TOKEND's directory structure should be simpler because errors in updating the soft-state directory will not cause incorrect operation.

8.2 Soft-State Directory Implementations

This section presents two implementations of TOKEND's soft-state directory.

8.2.1 A Simple Soft-State Directory

The simplest approach to implementing the soft-state directory is to update the approximate sharing information whenever the memory controller receives a transient or persistent request. In many cases, this approach will allow the memory controller to draw the same conclusions as in a traditional directory protocol. Whenever the memory controller receives a read request, it adds that processor to the list of sharers; when it receives a write request, it clears the set of sharers and updates the current owner of the block.

Although this approach works well in the absence of races, racing requests can confuse this scheme. For example, due to delays in the interconnect, a pair of forwarded write requests could arrive at all the processors in one order (P_0 before P_1), but arrive at the home memory in the reverse order (P_1 before P_0). In this situation P_1 will likely hold all the tokens, but the soft-state directory will believe that P_0 holds all the tokens. Thus, the soft-state directory may not always accurately reflect that current location of tokens in the system. Fortunately, the next time a non-racing write miss occurs for the block, the soft-state entry for that block will once again correctly reflect the current state of the block.

The advantage of this approach is that it is familiar and simple (due to its similarly to a directory protocol and lack of blocking states). Unfortunately, with this approach the soft-state directory can easily become confused and lose track of which processors are holding tokens (in which case the system may need to resort to a persistent request). Another disadvantage of this approach arises from our use of the migratory sharing optimization; that is, the memory cannot easily distinguish between a read request that receives a single-token response (in which case the memory controller should add the requester to the list of sharers) and a request that receives a response with all tokens

(in which case the memory controller should clear the sharers and change the owner). This problem is solved by adding explicit completion messages, described next.

8.2.2 A More Accurate Soft-State Directory

Although the previously described approach is reasonable, TOKEND adopts an approach that better handles racing requests and migratory sharing. In this approach, processors send *completion messages* to the home memory whenever they complete a request. The 8-byte completion message includes the identify of the processor, the address of the block, and the new MOESI state of the processor. Although these completion messages increase interconnect traffic, DIRECTORY introduces similar overheads due to its analogous messages that "unblock" the directory.

In this scheme, the soft-state directory is updated in response to three events. First, when the memory receives a read or write request, it adds the processor to a special *pending* set of processors (much like the set of sharers). When the memory forwards a request to the sharers or the owner, the memory also forwards the request to all processors in the pending set. Second, when the home memory receives a completion message, it (1) removes that processor from the pending set, (2) uses the MOESI state of the processor included in the completion message to update the soft-state directory. For example, if a processor enters the MODIFIED state, the directory updates the entry for that block by clearing the list of sharers and updating the owner. If the processor enters the SHARED state, it is added to the list of sharers. Third, the memory module also uses writeback and eviction notification messages to update the soft-state directory. If the memory receives the writeback of the owner token, it clears the owner entry in the soft-state directory. If the memory receives a non-owner token, it removes that processor from the set of sharers.

8.3 Evaluation of TOKEND

We use four questions to present evidence that TOKEND has similar characteristics (performance and traffic) as directory protocols. Table 8.1 contains the raw simulation data used to answer these questions. In these experiments, the soft-state directory tracks the owner and uses a simple bit-vector (one bit per processor) for encoding sharers and pending processors. As with DI-

configuration	all cycles per transaction	L2 misses per transaction	instructions per transaction	cycles per instruction	misses per thousand instructions	endpoint msgs per miss	interconnect bytes per miss
SPECjbb							
Perfect L2	19,903	NA	54,820	0.36	NA	NA	NA
TokenD - fast	38,764	182	57,121	0.68	3.20	4.53	295.07
TokenD - slow	41,985	182	57,370	0.73	3.18	4.52	295.73
Directory - fast	39,307	182	57,201	0.69	3.20	6.01	319.32
Directory - slow	41,102	182	57,270	0.72	3.18	6.01	318.37
TokenB	36,604	180	56,911	0.64	3.17	18.20	376.51
Apache							
Perfect L2	176,559	NA	273,711	0.65	NA	NA	NA
TokenD - fast	599,928	3,910	361,479	1.66	10.92	4.62	252.38
TokenD - slow	727,229	4,176	405,382	1.79	10.14	4.60	251.13
Directory - fast	609,117	4,017	367,030	1.66	10.89	5.36	263.93
Directory - slow	702,715	4,063	390,853	1.78	10.31	5.35	263.71
TokenB	534,197	3,833	339,043	1.56	11.15	18.16	332.93
OLTP							
Perfect L2	1,747,629	NA	3,385,566	0.52	NA	NA	NA
TokenD - fast	6,978,361	47,241	6,515,156	0.99	6.39	4.90	229.78
TokenD - slow	9,109,960	45,520	10,398,260	0.89	4.45	4.98	230.54
Directory - fast	7,301,596	47,189	7,000,286	1.00	6.40	4.64	227.65
Directory - slow	9,463,958	48,304	11,557,401	0.85	4.32	4.65	227.30
TokenB	5,727,614	42,412	5,802,205	0.99	7.28	18.36	306.82

Table 8.1 TOKEND Results for the TORUS Interconnect. The metrics presented in this table were described in Table 6.2 on page 101.

RECTORY in the last chapter, we perform simulations in which TOKEND's per-block information is either held in fast SRAM or in the main memory's DRAM (both of these approaches to storing per-block state were described in Section 3.2.4).

8.3.1 Question#1: Is TOKEND's soft-state directory effective?

Answer: Yes; an ineffective soft-state directory would increase the number of reissued requests; since TOKEND has similar rates of reissued requests, the soft-state directory is effective.

	SPECjbb			Apache			OLTP		
	0	1	P.R.	0	1	P.R.	0	1	P.R.
TokenB	99.5	0.2	0.3	99.1	0.7	0.2	97.5	1.6	1.0
TokenD	99.6	0.1	0.3	99.4	0.4	0.2	98.3	0.9	0.8

Table 8.2 TOKEND Reissued Requests. This table shows the percent of cache misses that (a) succeed the first transient request and thus are not reissued even once (the "0" column), (b) are reissued once and succeed on this second transient request (the "1" column), and (c) invoke a persistent request (the "P.R." column).

Table 8.2 shows the percent of reissued requests and persistent requests for TOKENB and TOKEND. Interestingly, TOKEND actually has slightly *fewer* reissued requests than TOKENB. We suspect the memory is acting as a per-block ordering point (because all requests in TOKEND are first sent to the directory, and messages are commonly delivered in order by the interconnect), resulting in a slight drop in the number of reissued requests.

8.3.2 Question#2: Is TOKEND's traffic similar to DIRECTORY?

Answer: Yes; to the first order, TOKEND's endpoint and interconnect traffic is similar to DIRECTORY. However, TOKEND actually uses more bandwidth in some cases. Figure 8.1 compares the endpoint traffic of TOKEND and DIRECTORY. As with previous traffic graphs, the traffic is segmented into five categories as described in the graph's legend. Starting with the lowest segment (the solid grey segment), TOKEND and DIRECTORY (unsurprisingly) generate the same about of data traffic . The second-lowest segment (dark grey striped) shows that TOKEND generates slightly more request and forwarded request traffic, likely due to the extra "pending" processors included by the soft-state directory when forwarding requests. The next segment (light grey striped)—only present for TOKEND—shows that reissued requests and persistent requests do increase TOKEND's traffic. The white segment shows a similar amount of traffic due to DIRECTORY's unblock messages and TOKEND's completion messages. Both protocols send one completion/unblock message per request, and both protocols could perhaps be improved to reduce the frequency of these messages.

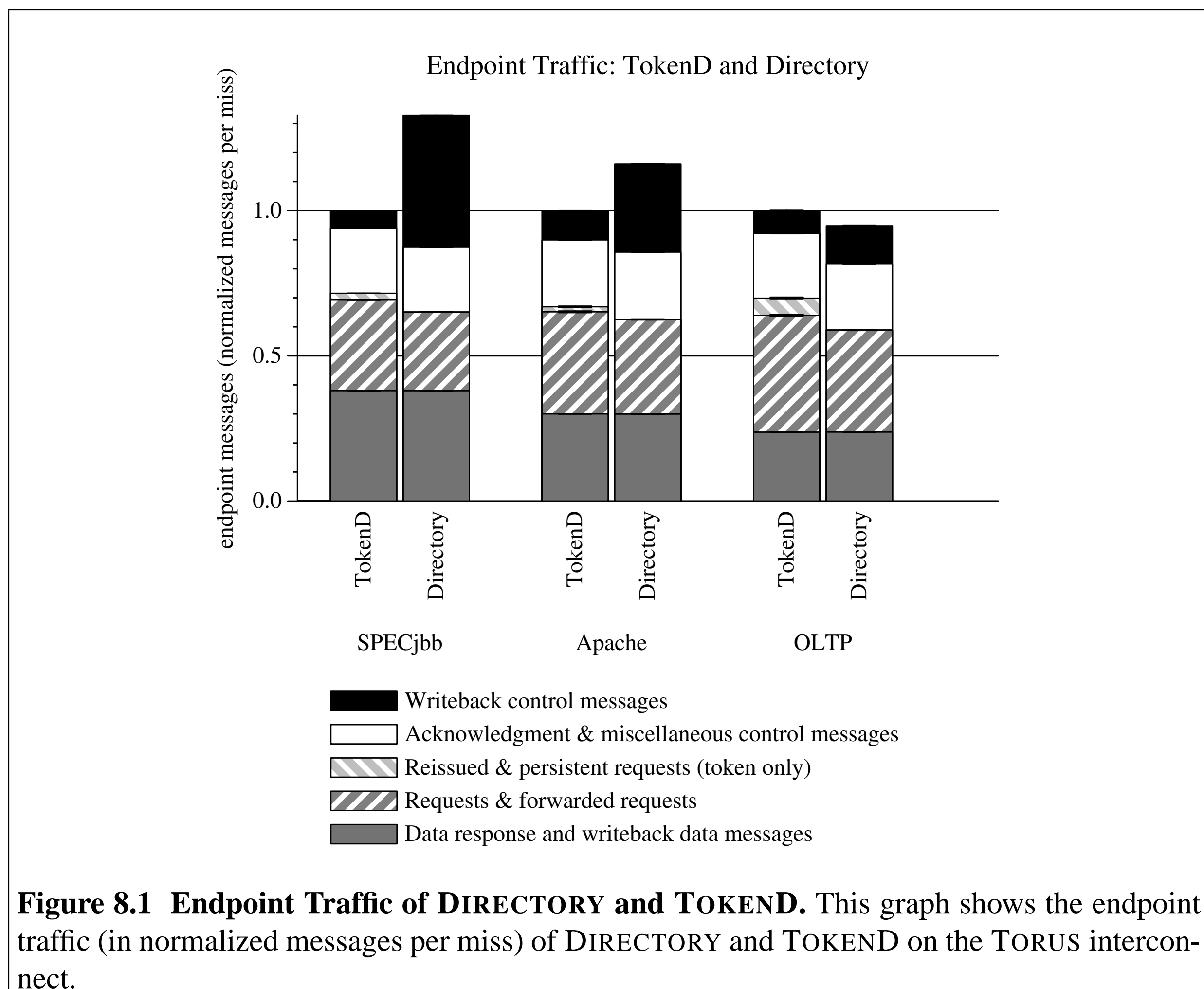


Figure 8.1 Endpoint Traffic of DIRECTORY and TOKEND. This graph shows the endpoint traffic (in normalized messages per miss) of DIRECTORY and TOKEND on the TORUS interconnect.

When taking only the four lower segments into account, TOKEND actually generates somewhat more traffic than DIRECTORY. However, when adding in the overhead of writeback control messages, DIRECTORY uses more bandwidth than TOKEND for two of the three workloads. The large amount of traffic caused by writebacks in DIRECTORY is due to its three-phase writeback implementation (Section 2.5.5) and lack of support for silent replacement of blocks in EXCLUSIVE. (Recall that we described a similar effect when comparing SNOOPING and TOKENB in Section 7.2.3.) In contrast, TOKEND uses a simple, bandwidth-efficient eviction mechanism that requires only a single “fire-and-forget” message (*i.e.*, it does not require an acknowledgment from the memory). A different directory protocol implementation might reduce the writeback over-

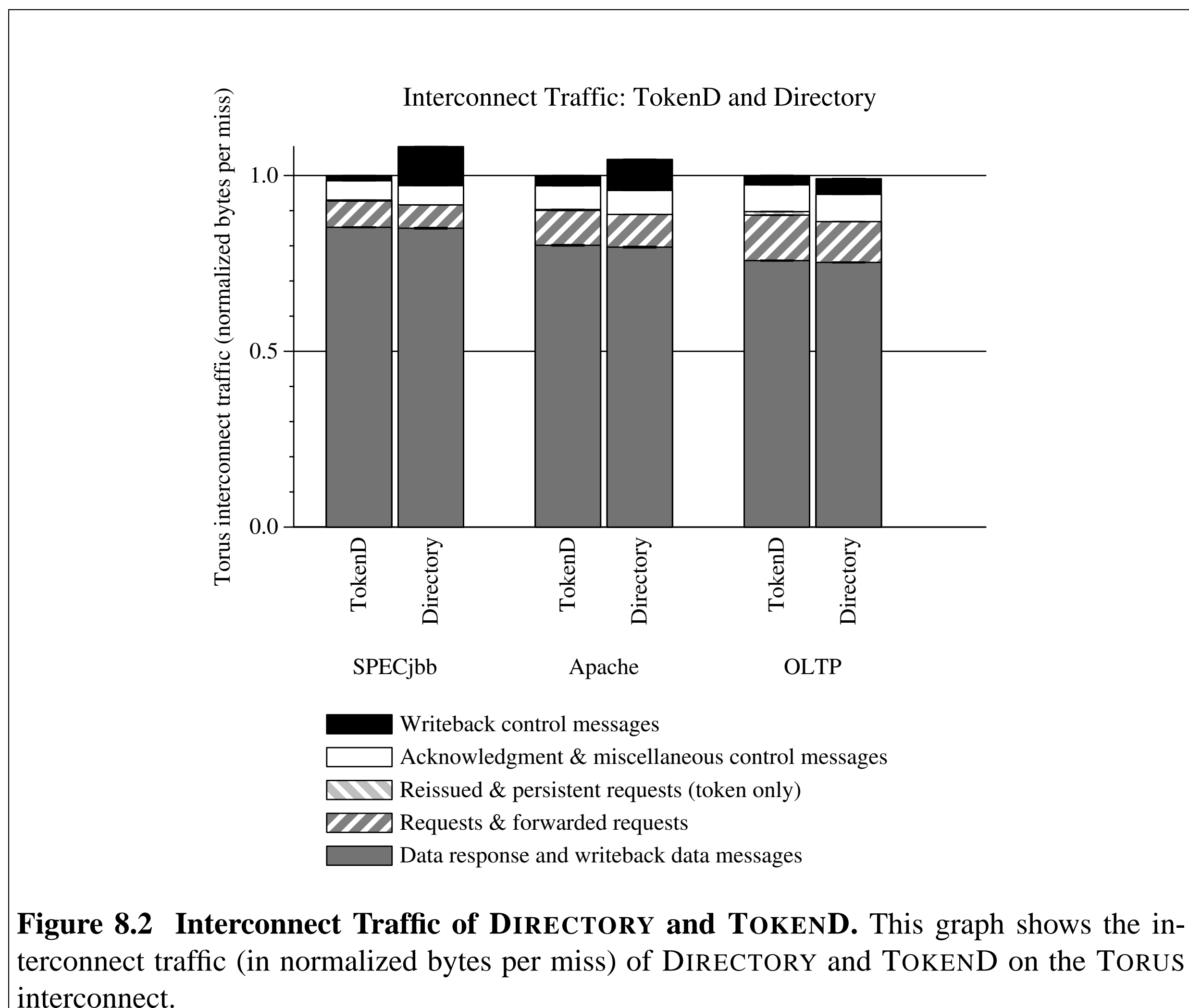


Figure 8.2 Interconnect Traffic of DIRECTORY and TOKEND. This graph shows the interconnect traffic (in normalized bytes per miss) of DIRECTORY and TOKEND on the TORUS interconnect.

heads, but this graph shows that the amount of traffic would still be similar. In contrast, TOKEND does not support silent evictions (in any state), but a directory protocol often supports silent evictions in both EXCLUSIVE and SHARED. A directory protocol with full support for silient evictions and traffic-efficient writebacks could thus use less traffic than TOKEND.

Examination of the traffic in terms of bytes of interconnect traffic (Figure 8.2) shows a similar comparison, but because data traffic is at least 80% of the interconnect traffic, the relative contribution of the non-data message is much smaller.

To investigate the scalability of TOKEND, we extended the analytical model from the last chapter to model TOKEND (described in Figure 8.3). The largest detriment to TOKEND's scalability is

Figure 8.3 An Analytical Model of the Traffic of TOKEND and DIRECTORY. Similarly to the model we introduced in Figure 7.12, we use the following analytical model of the traffic of TOKEND and DIRECTORY to explore the relative traffic generated by these protocols over a range of processors. To simplify the model, we focus on a simple memory-to-cache miss and ignore writeback data and control messages. This model has several parameters:

- n is the number of processors in the system. The cost of a unicast point-to-point message on TORUS is $\frac{1}{2}\sqrt{n}$ link crossings on average. A broadcast on TORUS crosses $n-1$ links.
- C is the size of control messages such as request and acknowledgment messages. C is 8 bytes for our simulations and for this model.
- D is the size of data messages such as data responses. D is 72 bytes (64-byte datablock plus a 8-byte header) for our simulations and for this model.
- p is the fraction of misses that invoke persistent requests. As a simplification, this model assumes that a persistent request is invoked after the first transient request fails to complete (*i.e.*, transient requests are not reissued). We assume this rate is constant as system size increases.
- $E_{protocol}$ is the endpoint traffic in messages per miss per endpoint for *protocol*.
- $I_{protocol}$ is the TORUS interconnect traffic in bytes per miss per link for *protocol*.

$$E_{TokenD} \equiv \overbrace{1 \cdot \underbrace{1}_{unicast}}^{transient\ request} + \overbrace{p \cdot \underbrace{(n-1)}_{broadcast}}^{persistent\ request} + \overbrace{1 \cdot \underbrace{1}_{unicast}}^{data\ response} + \overbrace{1 \cdot \underbrace{1}_{unicast}}^{completion}$$

$$E_{Directory} \equiv \overbrace{1 \cdot \underbrace{1}_{unicast}}^{request} + \overbrace{1 \cdot \underbrace{1}_{unicast}}^{data\ response} + \overbrace{1 \cdot \underbrace{1}_{unicast}}^{completion} \equiv 3$$

$$I_{TokenD} \equiv \overbrace{C \cdot \underbrace{\frac{1}{2}\sqrt{n}}_{unicast}}^{transient\ request} + \overbrace{C \cdot p \cdot \underbrace{(n-1)}_{broadcast}}^{persistent\ request} + \overbrace{D \cdot \underbrace{\frac{1}{2}\sqrt{n}}_{unicast}}^{data\ response} + \overbrace{C \cdot \underbrace{\frac{1}{2}\sqrt{n}}_{unicast}}^{completion}$$

$$I_{Directory} \equiv \overbrace{C \cdot \underbrace{\frac{1}{2}\sqrt{n}}_{unicast}}^{request} + \overbrace{D \cdot \underbrace{\frac{1}{2}\sqrt{n}}_{unicast}}^{data\ response} + \overbrace{C \cdot \underbrace{\frac{1}{2}\sqrt{n}}_{unicast}}^{completion} \equiv \frac{1}{2}(D+2C)\sqrt{n}$$

The above definitions calculate the traffic caused by a single miss on the assumption that a linearly-increasing system-wide miss rate and a linearly-increasing number of interconnect links and endpoints offset each other.

n	TOKEND v. DIRECTORY, $p = 0.00$	TOKEND v. DIRECTORY, $p = 0.05$
4	1.00	1.01
8	1.00	1.02
16	1.00	1.03
32	1.00	1.05
64	1.00	1.07
128	1.00	1.10
256	1.00	1.14
512	1.00	1.21

Table 8.3 Results from an Analytical Model of Traffic: TOKEND versus DIRECTORY. This table shows the increase in interconnect traffic on TOKEND over DIRECTORY on the TORUS interconnect using the simple analytical model of traffic described in Figure 8.3. The left column is the number of processors. The other two columns show the amount of traffic used by TOKEND over DIRECTORY, expressed as a traffic multiple (*i.e.*, 2x means TOKEND uses twice as much bandwidth as DIRECTORY). The middle column assumes the rate of persistent requests (p) is zero; the right column assumes the rate of persistent requests is 0.05 (5%).

our broadcast-based persistent request mechanism. Assuming the rate of persistent requests does not increase dramatically as the number of processors increases, the results from this new analytical model (Table 8.3) show that TOKEND scales almost as well as DIRECTORY. For example, TOKEND generates 21% more interconnect traffic than DIRECTORY for a 512-processor system with a 5% persistent request rate.

8.3.3 Question#3: Does TOKEND perform similarly to DIRECTORY?

Answer: Yes; TOKEND's performance is similar to DIRECTORY, because they use the same interconnect and add the same indirection latency to cache-to-cache misses. Figure 8.4 shows that the performance of DIRECTORY and TOKEND is similar (*i.e.*, the differences in runtime between the two protocols is not statistically significant).

8.3.4 Question#4: Does TOKEND outperform TOKENB?

Answer: With sufficient bandwidth, no. TOKENB is faster than TOKEND due to TOKEND's slower cache-to-cache transfers. Figure 8.4 shows that with unbounded link bandwidth TOKENB

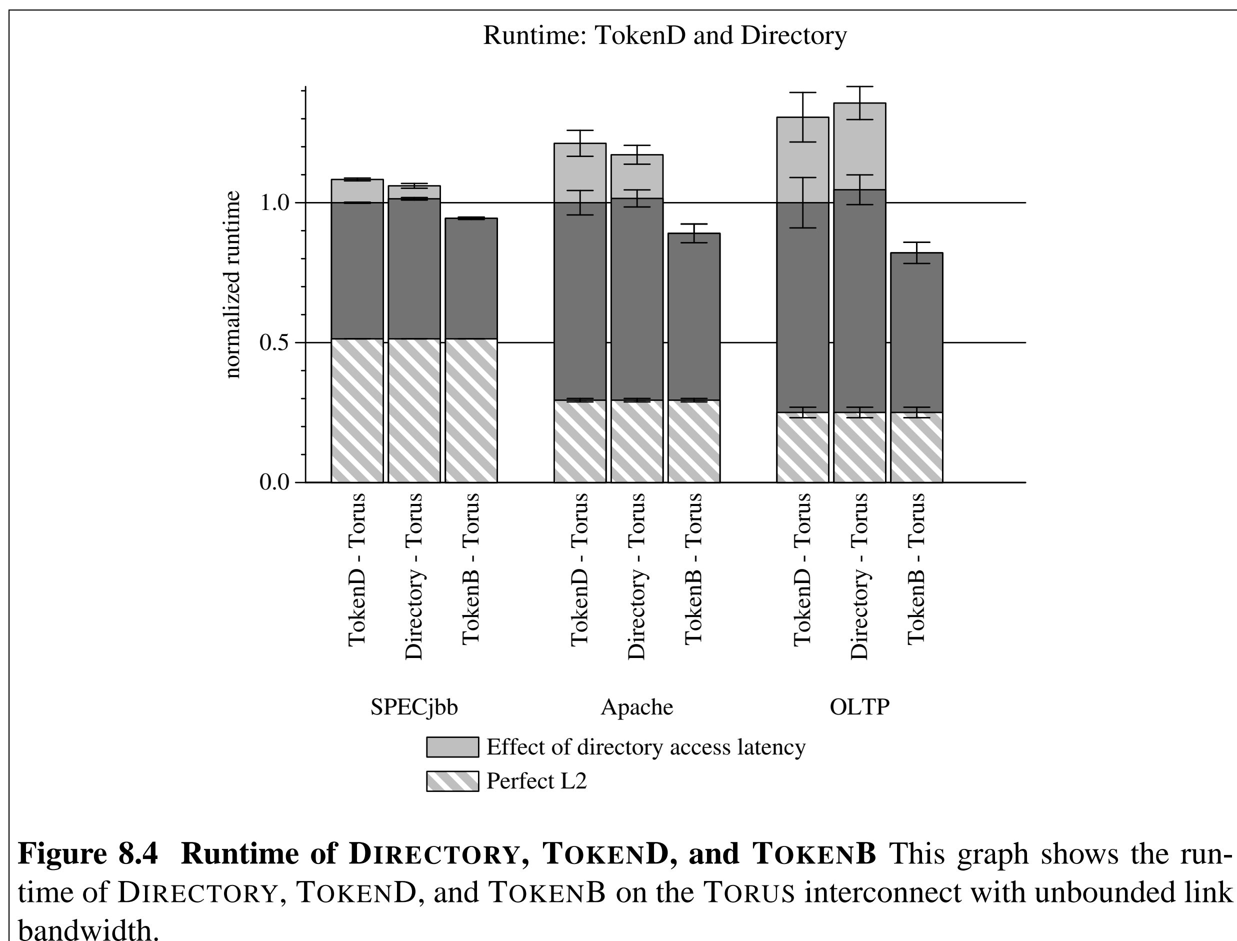


Figure 8.4 Runtime of DIRECTORY, TOKEND, and TOKENB This graph shows the runtime of DIRECTORY, TOKEND, and TOKENB on the TORUS interconnect with unbounded link bandwidth.

is 15–59% faster than TOKEND with a slow directory and 6–22% faster then TOKEND with a fast directory. As TOKENB performs similarly even when the link bandwidth is limited to 4GB/s for these 16-processor simulations (as discussed in Section 7.2.4), TOKENB is still faster than TOKEND in such systems. However, TOKEND will become relatively faster as the link bandwidth becomes more constrained or the system becomes larger.

8.3.5 TOKEND Results Summary

TOKEND has similar performance and traffic characteristics as DIRECTORY. The bandwidth-efficiency of these protocols is desirable for cost and scalability reasons. The effective soft-state directory implementation and low rate of persistent requests allows TOKEND's traffic to scale similarly to DIRECTORY. Although these protocols use less traffic than TOKENB, they are slower

than TokenB when bandwidth is plentiful. Thus, neither TokenB nor TokenD is "better" than the other; the relative merits of these protocols depend on the specific system configuration. In essence, the differences between TokenB and TokenD are an example of a bandwidth/latency tradeoff. In the next chapter we evaluate TokenM, a performance policy that attempts to improve this bandwidth/latency tradeoff by capturing both the bandwidth efficiency of TokenD and the low latency of TokenB.

Chapter 9

TOKENM: A Predictive-Multicast Performance Policy

TOKENM is a hybrid protocol that uses predictive multicast to capture most of the latency benefits of TOKENB, while capturing some of the bandwidth efficiency of TOKEND. TOKENB avoids the overhead of directory indirection for cache-to-cache misses (reducing average miss latency) by broadcasting all requests, but uses substantial interconnect bandwidth. In contrast, TOKEND is bandwidth efficient, but it adds indirection through a directory for cache-to-cache misses (like all directory protocols). An ideal protocol would achieve the low latency of TOKENB with the bandwidth efficiency of TOKEND. Figure 9.1 illustrates this bandwidth/latency tradeoff.

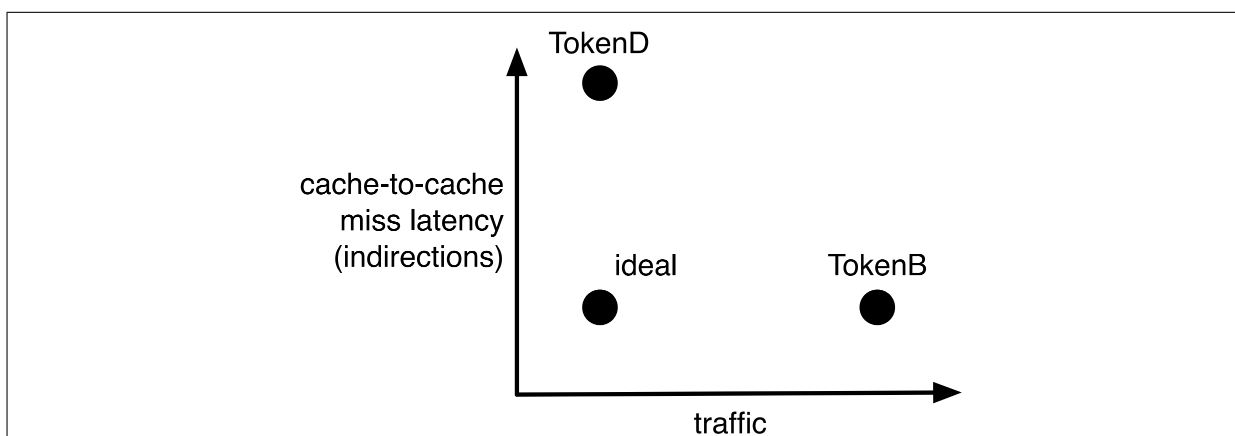


Figure 9.1 A Bandwidth/Latency Tradeoff. TOKENB has low-latency cache-to-cache misses, but uses significant amounts of bandwidth. In contrast, TOKEND is bandwidth efficient, but has higher-latency cache-to-cache misses. An ideal protocol would have the low latency of TOKENB with the bandwidth efficiency of TOKEND. TOKENM attempts to achieve this ideal design point by using destination-set prediction.

TOKENM strives for this ideal design point by extending TOKEND to support destination-set prediction. In TOKENM, when a processor issues a request, it sends the transient request to a predicted destination set. A *destination set* is the set of system components (processors and memory modules) to which a processor sends its request. The components react to incoming transient requests in the same manner as TOKENB and TOKEND, and when the recipients respond with sufficient tokens, the requester completes its request without indirection and without broadcast. When the predicted destination set is insufficient (*e.g.*, it does not include all the processors that need to see the request), a soft-state directory forwards the request to all processors that it believes the requester left out of the destination set (TOKENM's operation is an extension to TOKEND). To allow the soft-state directory to filter out those processors that received the request initially, request messages are extended to include the destination set to which each request was sent. TOKENM's operation is further described in Section 9.1). TOKENM—like all performance policies—is always correct because transient requests are only hints and the system relies upon the correctness substrate to enforce safety and prevent starvation.

TOKENM uses a destination-set predictor to improve the bandwidth/latency characteristics of the system. In the limit of perfect prediction, TOKENM provides a near-ideal bandwidth/latency design point, capturing both the low latency of TOKENB and the bandwidth efficiency of TOKEND. Section 9.2 describes three destination-set predictors we developed in our prior work [79]. In this earlier work we explored the destination-set predictor design space and evaluated several predictors in the context of Multicast Snooping [20, 114]. This dissertation does not include a detailed design space exploration, but it instead relies on the predictors Martin *et al.* [79] found to be the most promising. Our evaluations (in Section 9.3) show that these predictors are accurate enough to create attractive alternatives in the bandwidth/latency design space. As we are not the first to use prediction to accelerate coherence protocols, we discuss related work in Section 9.4.

9.1 TOKENM's Operation

TOKENM is a minor extension of TOKEND that uses a sequence of two predictions. The requester performs the first prediction—the destination-set prediction—when it selects the set of

destinations that will receive its request (Section 9.2 describes our specific predictors). The requester sends the transient request to these destinations (which—for our predictors—will always include the home memory module). The memory module performs the second prediction by using TOKEND's soft-state directory to determine which additional processors it believes need to see the request. In essence, this operation is also a prediction, because it is not guaranteed to always be correct. The directory forwards the transient request to those processors (if any) that it thinks should have been included, but were not included, in the predicted destination set. Using TOKEND's soft-state directory efficiently handles the situation in which the first prediction was insufficient. This second prediction may also fail to gather sufficient tokens (just as TOKEND's soft-state directory is not always successful). To recover from this double mis-prediction, the requester will reissue the request (again, much like TOKEND). Unlike TOKEND, TOKENM makes a destination-set prediction on both the initial and reissued request.

TOKENM's operation is identical to TOKEND's operation with only two exceptions. First, in TOKENM processors send transient requests to the predicted destination set (which includes the home memory module). Second, the soft-state directory forwards the transient requests as in TOKEND, but removes any destinations in the original destination set from the set of destinations to which it forwards the request. For example, if the soft-state directory determines that the memory should forward the request to both P_0 and P_1, but the original destination-set included only P_0, the memory forwards the request to only P_1. This filtering of forwarded requests is an optional traffic-reduction enhancement to prevent processors from receiving redundant transient requests. The main cost of this enhancement is that processors must encode the destination set (or an approximation of it) in the transient request message sent to the memory. The size of this encoding is determined by the destination sets generated by the predictors.[1]

In the extreme cases of always predicting broadcast or always predicting to send the request only to the home memory, TOKENM degenerates to TOKENB and TOKEND, respectively. Thus,

[1]For example, a predictor that includes at most one processor could encode its destination set in $\log_2 n$ bits. A predictor that is restricted to either broadcasting or sending a request only to the home memory could encode this difference using a single bit. A predictor that can generate a general destination-set prediction requires a n-bit vector.

by building upon TOKENB and TOKEND, TOKENM (mostly) subsumes them. The only disadvantages that TOKENM has over TOKENB are (1) the cost of implementing the soft-state directory, (2) the overhead of the completion messages (both described previously in Section 8.1), and (3) encoding of the destination set in request messages. The two additional mechanisms that TOKENM adds over TOKEND are (1) a small amount of logic to filter the recipients of forwarded requests, and (2) encoding of the destination set in request messages.

TOKENM reissues requests that fail to complete using a similar approach as TOKEND. In TOKENM, the initial request and first reissued request are each sent to the predicted destination-set. The system resorts to a broadcast-based persistent request after the first reissued request fails to complete.

9.2 Destination-Set Predictors

This section describes the goals of destination-set prediction (Section 9.2.1), our approach to prediction (Section 9.2.2), common mechanisms used for all our predictors (Section 9.2.3), three specific predictors (Section 9.2.4), and the use of macroblock indexing to capture spatial predictability (Section 9.2.5). This discussion is a more focused version of the discussion found in our prior work [79].

9.2.1 Predictor Goals

The goal of the predicted destination set in TOKENM is the same as that of the soft-state directory in TOKEND: include all of the token holders for a write request and include the holder of the owner token for a read request. Over-predicting (*i.e.*, including unnecessary processors in the destination set) increases system traffic, but does not directly increase the miss latency (because unnecessary processors do not respond, and thus the requester does not need to wait for responses from them). Under-predicting (*i.e.*, not including all necessary processors) increases the miss latency. In most cases of under prediction, the soft-state directory will forward the request to the remaining necessary processors (adding indirection latency to the critical path of the miss). If this forwarded transient request fails to find sufficient tokens (due to racing requests or an inaccurate

soft-state directory), the processor will reissue the request after a timeout interval (and eventually resort to a persistent request, if necessary).

9.2.2 Our Approach

Like most predictors used throughout computer architecture, our destination-set predictors use past behavior to make guesses about the future. Since the processor is predicting a property of coherence events, it uses coherence events as training input to its predictor. For example, if two processors are sharing the same block (*i.e.*, pairwise sharing), both processors can observe the consistent sharing pattern. This allows each processor to know it should predict that the other processor is the only processor holding the block when issuing a request for the block.

The prediction policies we discuss use two types of training cues to predict sharing behavior: (1) transient and persistent requests from other processors and (2) coherence responses. In both cases, the predictor learns the identity of a processor that has recently accessed the block. On requests from other processors, the predictor automatically receives the requesting processor's identity (since this information is required to permit a response). For responses, we assume that data response messages include the sender's identity. Specific policies (described later in Section 9.2.4) use this information either to "train up" or "train down" (*i.e.*, increase or decrease the destination set).

9.2.3 Common Predictor Mechanisms

Each second-level cache controller in the system contains a destination-set predictor. Because only coherence controllers are responsible for interacting with the predictor, we require no modifications to the processor core. Predictors are tagged, set-associative, and indexed by data block address (actually they are indexed by macroblock index, described later in Section 9.2.5). Our evaluation of TOKENM uses 4-way set-associate predictors with 8,192 entries. Such a table ranges from 32kB to 64kB, because each entry ranges from 4 to 8 bytes in size (depending on the specific prediction policies described in Section 9.2.4). Predictors of this size have modest cost, but achieve most of the benefit of predictors of unbounded size [79].

The predictor generates a prediction whenever a processor issues a transient request. The coherence controller accesses the predictor in parallel with the second-level cache access.[2] In the event of a cache miss, the controller uses the predicted destination set when sending the resulting transient request. If the predictor hits, it generates a prediction according to the policies discussed next. On a predictor miss, the requester sends the request to only the home memory module.

A processor updates its persistent request table based on incoming training information. To update existing entries for a block, a predictor uses (1) its own incoming responses and (2) requests from other processors. A predictor entry for a block is allocated—and the least-recently-used (LRU) entry discarded—when the processor receives a token response from another processor (but not when it receives a response from memory). That is, it allocates an entry when it receives a response signifying that another processor was sharing the data (*i.e.*, the miss was a cache-to-cache miss). This approach results in efficient utilization of the predictor entries because (1) a processor's predictor allocates entries only for blocks the processor has recently requested, and (2) a processor does not allocate entries for blocks that can always be found at the home memory (*e.g.*, private data, instructions, or read-only data structures).

9.2.4 Three Specific Predictor Policies

Different prediction policies use training information to target different points in the bandwidth/latency spectrum [79]. Figure 9.2 shows the approximate bandwidth/latency tradeoff space targeted by our three predictors: (1) OWNER, a predictor with similar traffic but fewer indirections than TOKEND, (2) BCAST-IF-SHARED (broadcast-if-shared), a predictor with similar latency but lower traffic than TOKENB, and (3) GROUP, a predictor that strives to provide a design point close to ideal. This section describes these three prediction policies, and Table 9.1 specifies them in table format.

[2]Predictor updates complete in a single cycle, and the predictors train only on data responses or on requests from other processors. Because multiple misses are generated in parallel, later misses do not always benefit from the training responses from the earlier misses before being issued into the memory system.

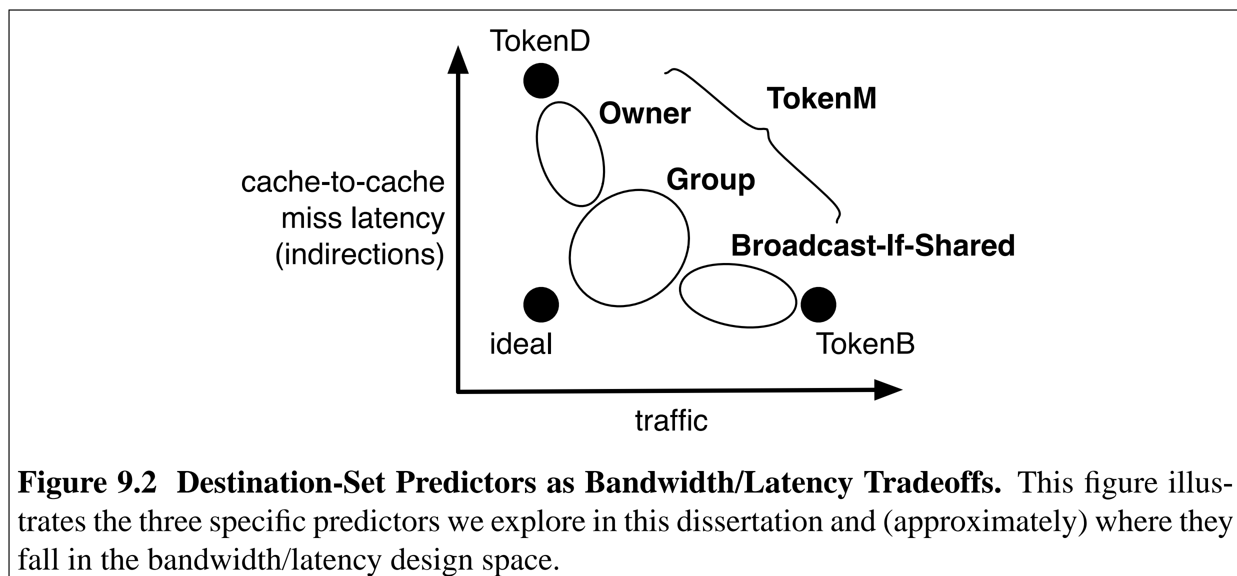


Figure 9.2 Destination-Set Predictors as Bandwidth/Latency Tradeoffs. This figure illustrates the three specific predictors we explore in this dissertation and (approximately) where they fall in the bandwidth/latency design space.

The OWNER predictor. OWNER provides lower latency than TOKEND while capturing most of TOKEND's bandwidth efficiency. OWNER targets scenarios in which either (a) only one other processor needs to be in the destination set (*e.g.*, pairwise sharing) (b) the number of processors in the system is large, or (c) bandwidth is limited. The predictor records the last processor that requested the block[3] or responded with a block. On a prediction, the predictor returns the union of the predicted owner and the home memory module. OWNER works well for pairwise sharing, because both processors include each other in their predictions. OWNER also works well for systems with many processors or with limited bandwidth because it sends at most one additional control message for each request, independent of the number of processors in the system.

The BCAST-IF-SHARED predictor. BCAST-IF-SHARED uses less traffic than TOKENB while capturing most of the low latency of TOKENB. BCAST-IF-SHARED targets scenarios in which either (a) most shared data are widely shared, (b) most data are not shared, (c) the number of processors in the system is small, or (d) bandwidth is plentiful. BCAST-IF-SHARED selects either a broadcast destination set (if the block is predicted to be shared) or only the home memory module (otherwise). A two-bit saturating counter—incremented on requests and responses from other pro-

[3]We update the owner on read requests because the migratory sharing optimization allows ownership to transfer on read requests.

Table 9.1 Predictor Policies

Name		Owner	Broadcast-If-Shared	Group
Entry Structure		*Owner* ID and *Valid* bit	2-bit saturating counter, *Counter*	N 2-bit saturating counters, *Counters[0..N-1]* saturating *RolloverCounter*
Entry Size		$\log_2$N bits + 1 bit + tag (approximately 4 bytes)	2 bits + tag (approximately 4 bytes)	2N bits + (1+$\log_2$N) bits + tag (approximately 8 bytes)
Prediction Action		If *Valid*, predict *Owner* Otherwise, predict home	If *Counter* > 1, broadcast Otherwise, predict home	For each processor *n*, if *Counters[n]* > 1, add *n* to predicted set; always include home
Training Action	**Data Response**	If response from memory, clear *Valid*. Else, set *Owner* to responder, and set *Valid*	If response from memory, decrement *Counter*. Else, increment *Counter*	If response not from memory, increment *Counters[responder]*. Increment *RolloverCounter*†
	Other Request	Set *Owner* to requester and set *Valid*	Increment *Counter*	Increment *Counters[requester]*. Increment *RolloverCounter*†

N is the number of processors in the system. †If *RolloverCounter* rolls over, decrement *Counter[i]* for all *i*.

cessors and decremented otherwise—determines which prediction to make. BCAST-IF-SHARED performs comparably to snooping in many cases, but it uses less bandwidth by not broadcasting all requests.

BCAST-IF-SHARED implicitly adapts to common characteristics of some classes of workloads. For example, BCAST-IF-SHARED should perform well in two common situations: (1) a "latency-bound" commercial workload with a moderate miss rate, lots of sharing, and few parallel misses (*e.g.*, some OLTP workloads) by broadcasting many requests and (2) a "bandwidth-bound" technical workload with a high miss rate, limited sharing, and many parallel misses (*e.g.*, a DAXPY kernel) by broadcasting few requests. To more explicitly adapt to workloads, a predictor could use bandwidth adaptive techniques [84] (not explored further in this dissertation).

The GROUP predictor. GROUP targets scenarios in which (a) stable groups of processors (less than all processors) share blocks and (b) bandwidth is neither extremely limited nor plentiful. Each predictor entry contains a two-bit counter per processor in the system. On each request or response, the predictor increments the corresponding counter. GROUP also increments the entry's rollover counter; when the value of this counter reaches twice the number of processors, the predictor decrements all 2-bit counters in the entry and resets the rollover counter to zero. This train-down mechanism ensures that the predictor eventually removes inactive processors from the destination set (*i.e.*, removes processors that are no longer accessing the block). GROUP should work well on a large multiprocessor system in which not all processors are working on the same aspect of the computation or when the system is logically partitioned.

9.2.5 Capturing Spatial Predictability via Macroblock Indexing

Our predictors capture the significant spatial predictability exhibited by cache-to-cache misses [79] by aggregating training information from several contiguous blocks (a *macroblock*) into a single predictor entry. Consider a processor reading a large buffer that was recently written by another processor. The last processor to write the buffer may be difficult to predict; however, once a processor observes that several data blocks of the buffer were supplied by the same processor, a macroblock-based predictor can learn to find other spatially-related blocks at that same processor. To exploit this spatial predictability, we index these predictors with 1024-byte macroblock addresses by simply dropping the least significant four bits of the 64-byte coherence block address. Macroblock indexing has the additional benefit of increasing the effective reach of the predictor, thereby reducing pressure on the predictor's capacity. Martin *et al.* [79] also explored program-counter based prediction and some hybrid predictors. We use macroblock index predictors because this previous work indicated they perform well in practice.

9.3 Evaluation of TOKENM

We use four questions to present evidence that TOKENM captures much of the bandwidth-efficiency of TOKEND while achieving much of the low latency for cache-to-cache transfers as with TOKENB. Table 9.2 contains the raw data used to answer these questions.

9.3.1 Question#1: Does TOKENM Use Less Traffic than TOKENB?

Answer: Yes; TOKENM uses the same as or less traffic than TOKENB. As shown in Figure 9.3 and Figure 9.4, the specific amount of traffic reduction depends on the predictor employed: BCAST-IF-SHARED uses 26–51% less endpoint traffic and 7–14% less interconnect traffic than TOKENB, GROUP uses 62–69% less endpoint traffic and 19–20% less interconnect traffic than TOKENB, and OWNER uses 72–75% less endpoint traffic and 21–24% less interconnect traffic than TOKENB.

configuration	all cycles per transaction	L2 misses per transaction	instructions per transaction	cycles per instruction	misses per thousand instructions	endpoint msgs per miss	interconnect bytes per miss
SPECjbb							
Perfect L2	19,903	NA	54,820	0.36	NA	NA	NA
TokenB	36,604	180	56,911	0.64	3.17	18.20	376.51
TokenM (BcastShared) - fast	37,015	180	56,878	0.65	3.17	8.86	322.54
TokenM (BcastShared) - slow	37,849	182	56,913	0.67	3.20	8.95	322.95
TokenM (Group) - fast	37,504	181	56,999	0.66	3.18	5.64	305.84
TokenM (Group) - slow	38,696	181	57,053	0.68	3.19	5.64	305.14
TokenM (Owner) - fast	37,853	180	57,049	0.66	3.16	4.64	296.22
TokenM (Owner) - slow	39,638	181	57,158	0.69	3.18	4.64	295.29
TokenD - fast	38,764	182	57,121	0.68	3.20	4.53	295.07
TokenD - slow	41,985	182	57,370	0.73	3.18	4.52	295.73
Apache							
Perfect L2	176,559	NA	273,711	0.65	NA	NA	NA
TokenB	534,197	3,833	339,043	1.56	11.15	18.16	332.93
TokenM (BcastShared) - fast	565,363	3,922	356,656	1.60	11.16	10.46	288.80
TokenM (BcastShared) - slow	559,364	3,833	335,966	1.68	11.44	10.43	289.46
TokenM (Group) - fast	561,300	3,883	345,907	1.62	11.29	6.22	265.61
TokenM (Group) - slow	619,625	3,978	375,616	1.65	10.62	6.21	265.21
TokenM (Owner) - fast	559,769	3,802	340,426	1.63	10.90	4.79	255.01
TokenM (Owner) - slow	666,077	4,113	403,831	1.66	10.23	4.81	254.02
TokenD - fast	599,928	3,910	361,479	1.66	10.92	4.62	252.38
TokenD - slow	727,229	4,176	405,382	1.79	10.14	4.60	251.13
OLTP							
Perfect L2	1,747,629	NA	3,385,566	0.52	NA	NA	NA
TokenB	5,727,614	42,412	5,802,205	0.99	7.28	18.36	306.82
TokenM (BcastShared) - fast	5,761,074	40,861	5,469,998	1.04	7.38	13.53	283.99
TokenM (BcastShared) - slow	6,464,493	43,578	5,937,644	1.09	7.34	13.48	283.54
TokenM (Group) - fast	6,005,692	41,863	5,866,853	1.03	7.10	6.97	248.04
TokenM (Group) - slow	6,972,473	44,364	6,576,820	1.07	6.78	6.96	247.80
TokenM (Owner) - fast	6,741,946	44,638	7,050,252	0.98	6.54	5.18	233.26
TokenM (Owner) - slow	7,829,489	45,522	8,768,484	0.91	5.17	5.22	233.86
TokenD - fast	6,978,361	47,241	6,515,156	0.99	6.39	4.90	229.78
TokenD - slow	9,109,960	45,520	10,398,260	0.89	4.45	4.98	230.54

Table 9.2 TokenM Results for the Torus Interconnect. The metrics presented in this table were described in Table 6.2 on page 101.

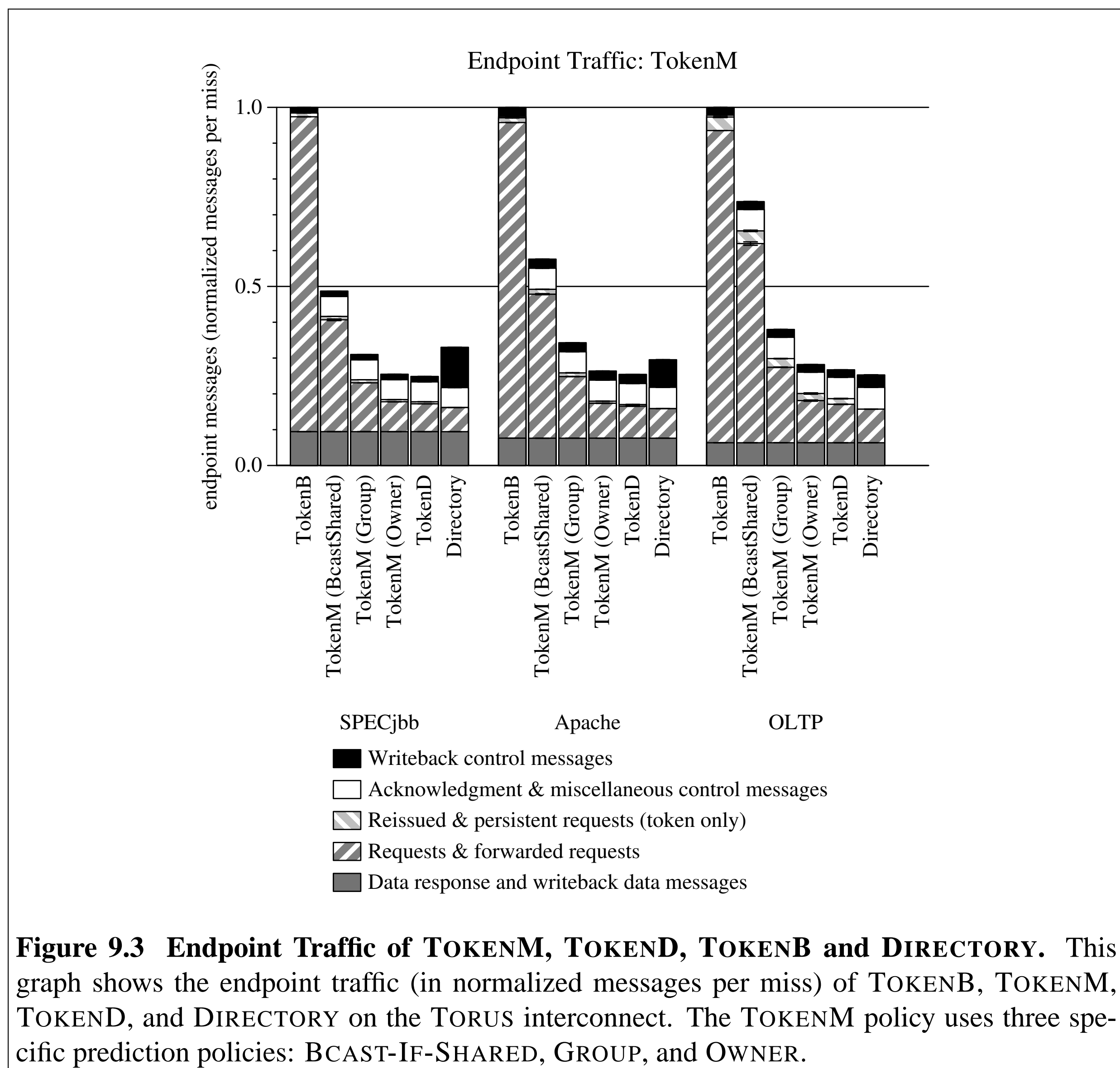


Figure 9.3 Endpoint Traffic of TOKENM, TOKEND, TOKENB and DIRECTORY. This graph shows the endpoint traffic (in normalized messages per miss) of TOKENB, TOKENM, TOKEND, and DIRECTORY on the TORUS interconnect. The TOKENM policy uses three specific prediction policies: BCAST-IF-SHARED, GROUP, and OWNER.

9.3.2 Question#2: Does TOKENM Outperform TOKEND?

Answer: Yes; TOKENM performs the same as or better than TOKEND. Figure 9.5 shows the relative performance of TOKENM without the effect of interconnect contention. In such a situation, TOKENB will always be fastest. However, we already discussed the relative traffic comparison of the protocols in question #1. The specific amount of performance increase of TOKENM over TOKEND depends on the predictor employed and speed of the directory:

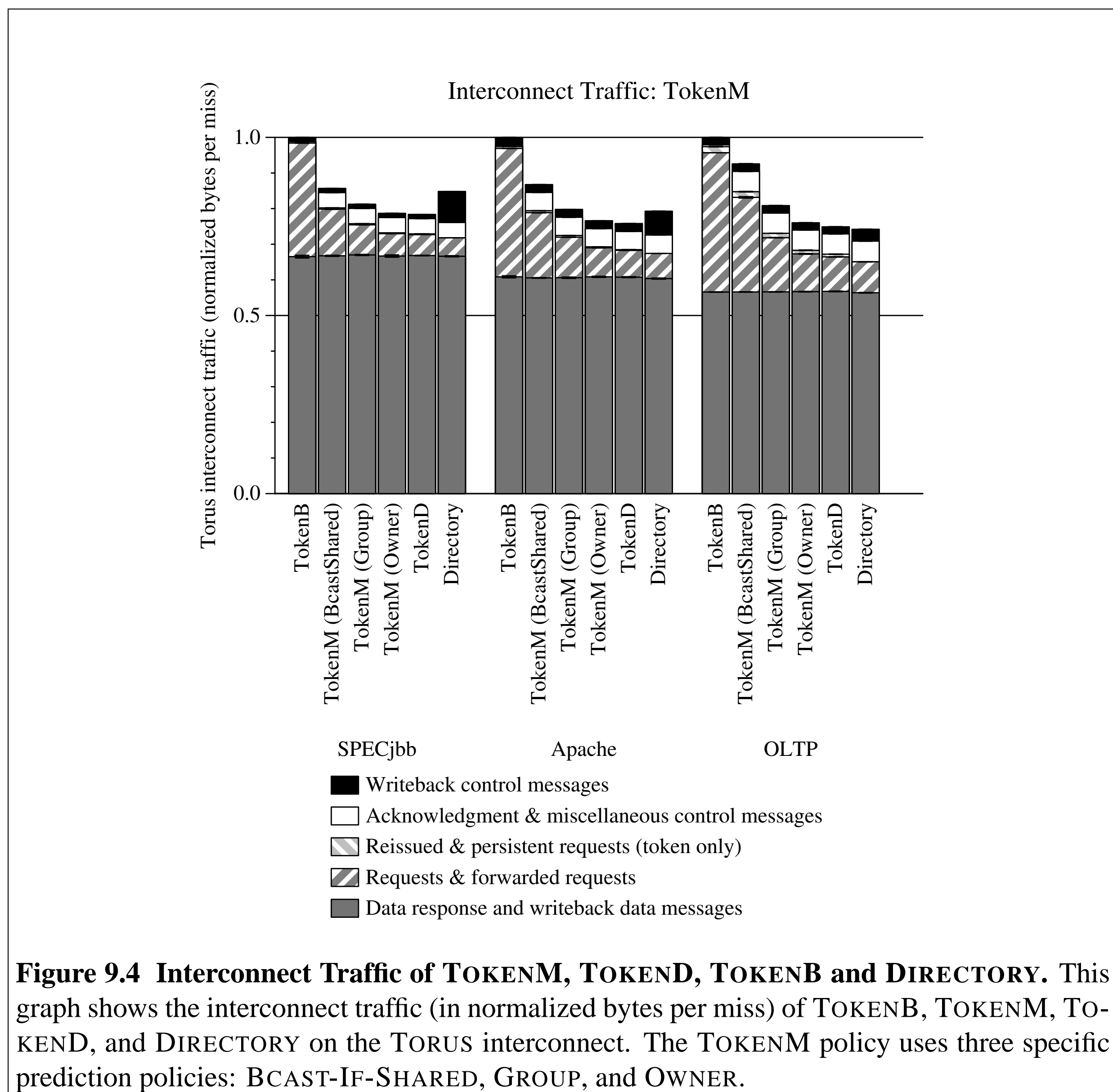


Figure 9.4 Interconnect Traffic of TOKENM, TOKEND, TOKENB and DIRECTORY. This graph shows the interconnect traffic (in normalized bytes per miss) of TOKENB, TOKENM, TOKEND, and DIRECTORY on the TORUS interconnect. The TOKENM policy uses three specific prediction policies: BCAST-IF-SHARED, GROUP, and OWNER.

- When using the fast directory: OWNER is 2–7% faster than TOKEND, GROUP is 3–16% faster then TOKEND, and BCAST-IF-SHARED is 5–21% faster then TOKEND. For comparison, TOKENB is 6–22% faster than TOKEND.

- When using the slow directory: OWNER is 6–16% faster than TOKEND, GROUP is 9–31% faster then TOKEND, and BCAST-IF-SHARED is 11–41% faster then TOKEND. For comparison, TOKENB is 15–59% faster than TOKEND.

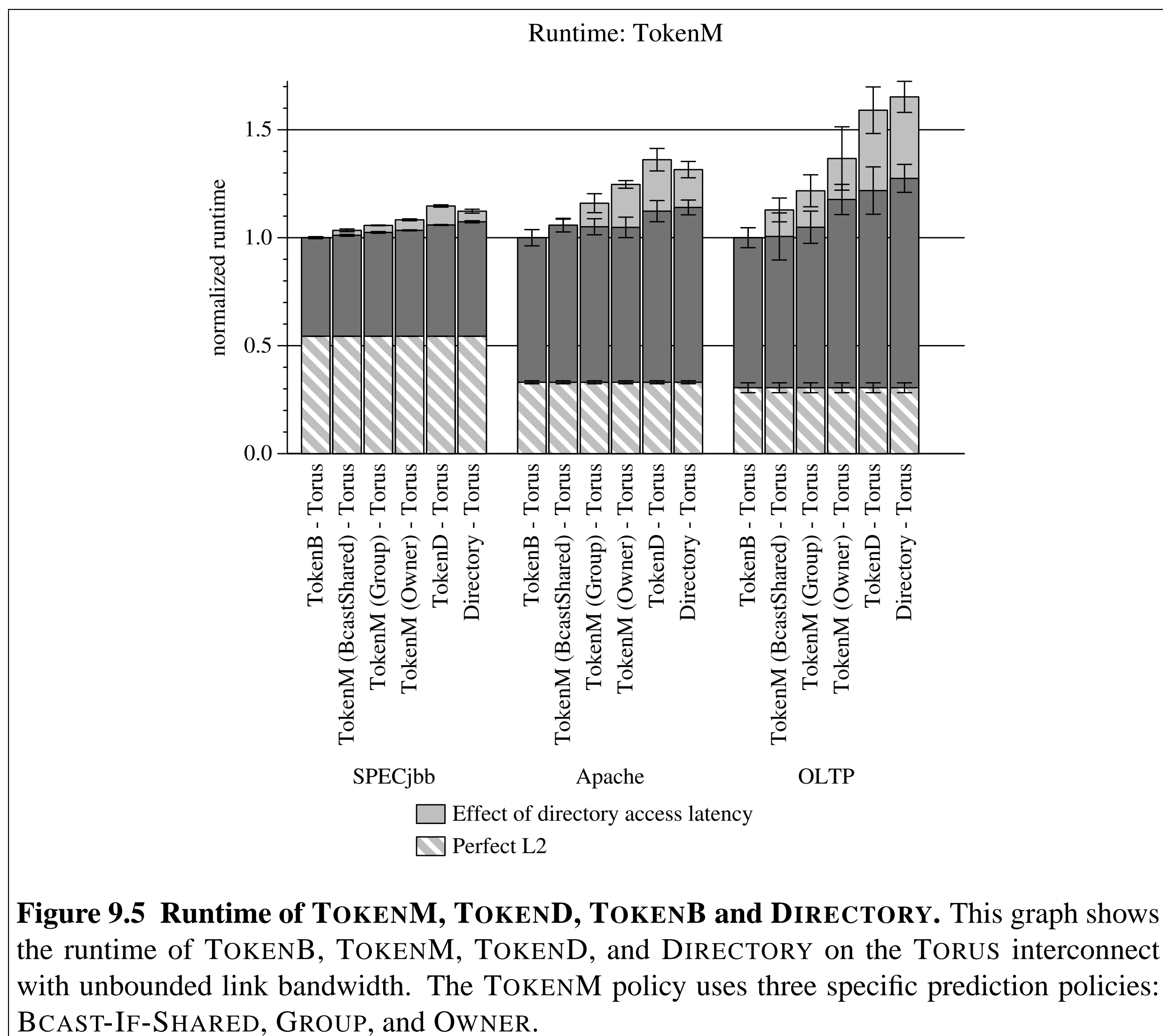


Figure 9.5 Runtime of TOKENM, TOKEND, TOKENB and DIRECTORY. This graph shows the runtime of TOKENB, TOKENM, TOKEND, and DIRECTORY on the TORUS interconnect with unbounded link bandwidth. The TOKENM policy uses three specific prediction policies: BCAST-IF-SHARED, GROUP, and OWNER.

As discussed next, the relative performances of these protocols is also affected by interconnect contention (because they use more traffic than TOKEND).

9.3.3 Question#3: Is TOKENM Always Better than TOKENB and TOKEND?

Answer: No; While TOKENM captures many of the benefits of TOKENB and TOKEND by providing an attractive bandwidth/latency tradeoff, TOKENB has a lower average latency and TOKEND is more bandwidth efficient than TOKENM. Neither protocol is always best; as with the

	SPECjbb			Apache			OLTP		
	0	1	P.R.	0	1	P.R.	0	1	P.R.
TokenB	99.5	0.2	0.3	99.1	0.7	0.2	97.5	1.6	1.0
TokenM (BcastShared)	99.5	0.2	0.3	99.0	0.7	0.3	97.7	1.4	0.9
TokenM (Group)	99.5	0.2	0.3	99.1	0.6	0.2	97.9	1.3	0.8
TokenM (Owner)	99.5	0.1	0.3	99.1	0.6	0.3	97.9	1.2	0.9
TokenD	99.6	0.1	0.3	99.4	0.4	0.2	98.3	0.9	0.8

Table 9.3 TOKENM Reissued Requests. This table shows the percent of cache misses that succeed the first transient request and thus are not reissued even once (the "0" column), are reissued once and succeed on this second transient request (the "1" column), and invoke a persistent request (the "P.R." column).

comparison between TOKEND and TOKENB, TOKENM provides additional design points in the space of bandwidth/latency tradeoffs in cache coherence protocols.

9.3.4 TOKENM Results Summary

The relative attractiveness of TOKENB, TOKEND, and TOKENM is a bandwidth/latency tradeoff. It depends on the cost of bandwidth, the amount of interconnect bandwidth, the amount of endpoint controller bandwidth, and system size. Experiments with 4GB/second link bandwidth for 16 processors (not shown), indicates no significant performance advantage for TOKENM or TOKEND over TOKENB.

However, TOKENM provides some attractive design points that might result in a cheaper or more scalable system than TOKENB while capturing most of TOKENB's performance. For example, TOKENM with the OWNER predictor uses 21–24% less interconnect traffic than TOKENB (only 0–2% more than TOKEND), and OWNER is 6–16% faster than TOKEND. Since the OWNER predictor only includes one other processor, its traffic is independent of number of processors in the system, and thus it maintains its bandwidth efficiency as system size increases. The end result is a protocol that is faster than TOKEND (and thus DIRECTORY) with a negligible amount of additional traffic. Similarly, BCAST-IF-SHARED results in a protocol that is slightly slower than TOKENB with a significant reduction in traffic.

9.4 Related Work

We are not the first to apply prediction to coherence protocols or observe that processor can predict the necessary recipients of coherence protocol messages. However, these previous proposals required complicated protocols or protocol extensions. By multicasting transient requests, Token Coherence provides a simpler implementation of these proposals, while eliminating the totally-ordered interconnect required by some proposals (*e.g.*, [20, 79, 114]) and complex races in other proposals (*e.g.*, [3, 4, 20, 79]). This chapter relied heavily on our previous investigation of destination-set prediction [79].

In many respects, TOKENM is a simpler implementation of Multicast Snooping [20, 114] without any interconnect ordering requirements. Sorin *et al.* [114] introduced an optimization to Multicast Snooping that reduced the latency of failed destination-set predictions by allowing the directory to immediately reissue the request (similar to forwarding the request in a directory protocol); TOKENM adopts a similar approach to handling insufficient destination-set predictions.

Acacio *et al.* [3, 4] extended a traditional directory protocol to exploit a limited form of destination-set prediction. These two proposals target read requests and write requests separately, and rely upon complicated race-resolution mechanisms that limit cases in which a processor can use prediction. Although these two proposals do not require a totally-ordered interconnect (unlike Multicast Snooping [20, 114]), they are still significantly more complicated than a traditional directory protocol.

Many other researchers have examined or exploited other forms of coherence prediction (*e.g.*, dynamic self-invalidation [70, 73]). Coherence predictors have been indexed with addresses [92], program counters [63], message history [69], and other state [64]. Researchers have also developed protocols that optimize for specific sharing behaviors [19], read-modify-write sequences [94, 95], and migratory sharing [31, 115]. Other hybrid protocols adapt between write-invalidate and write-update [12, 34, 62, 90, 110], by migrating data near to where it is being used [37, 48, 125] or by adapting to available bandwidth [84].

Chapter 10

Conclusions

In this chapter we review our finds on Token Coherence (Section 10.1), and outline Token Coherence's future directions and further challenges (Section 10.2). Finally, I reflect on the current state of cache coherence and provide some unsubstantiated opinions concerning the design of future multiprocessors (Section 10.3).

10.1 Token Coherence Summary and Conclusions

We have introduced Token Coherence, a new framework for cache coherence that decouples performance and correctness. We motivated the desire for such a framework by highlighting the performance limiting attributes of current protocols (indirection and totally-ordered interconnects). We described the relationship between Token Coherence and the traditional MOESI states and discussed the current state of the art of coherence protocol design. We described the operation and implementation of the correctness substrate which uses token counting to ensure safety and persistent requests to prevent starvation. We outlined the approach of using performance policies to guide the operation of the system, and explained that such policies have no correctness requirements because the correctness substrate is ultimately in control. We then showed the general applicability and flexibility of this decoupled approach by exploring three proof-of-concept performance policies: TOKENB, TOKEND, and TOKENM.

We motivated and described these three performance policies, and used a state-of-the-art simulation infrastructure to quantitatively evaluate the protocols. Our results show:

- TokenB, our broadcast-based performance policy, can outperform both snooping protocols (by avoiding the higher latency of a totally-ordered interconnect) and directory protocols (by avoiding indirection for frequent cache-to-cache misses).

- TokenD allows Token Coherence to approximate the latency and traffic characteristics of a directory protocol. Although TokenD is not faster than TokenB for the 16-processor systems with high-speed links that we simulated, TokenD uses much less bandwidth as the number of processors in the system increases.

- TokenM, a hybrid of TokenB and TokenD that uses destination-set prediction, provides a range of design points in the space of bandwidth/latency tradeoffs. Depending on the specific predictor being used, TokenM can achieve (1) the low latency of TokenB while using less bandwidth, (2) the bandwidth efficiency of TokenD while reducing the number of indirections, or (3) most of both the latency benefits of TokenB and the traffic advantages of TokenD.

We conclude that Token Coherence is an attractive substrate for building future cache-coherent multiprocessors. Token Coherence can emulate the desirable characteristics of existing protocols while reducing protocol requirements by not relying on a totally-ordered interconnect or on the accuracy of the directory state that encodes the sharers and owner. In addition, Token Coherence's flexibility allows for the creation of hybrid protocols with only minor changes over the base protocols, and the resulting hybrid protocols can provide attractive protocols that provide a good balance between uncontended latency and system traffic.

10.2 Future Directions and Further Challenges

Although this dissertation has mostly focused on the performance characteristics of three particular performance policies, we believe that Token Coherence has other desirable attributes that were not fully explored in this dissertation.

10.2.1 How Else Can Systems Exploit Token Coherence's Flexibility?

Token Coherence's separation of correctness and performance provides great flexibility. We exploit this flexibility in this dissertation by creating better implementations of existing protocols (TOKENB and TOKEND) and creating a hybrid protocol based on destination-set prediction (TOKENM). However, Token Coherence's flexibility may provide significant additional opportunities. In Section 5.4 we briefly described several other potentially attractive performance policies. These examples include bandwidth-adaptive hybrids, protocols that reduce miss frequency by predictively pushing data between system components, multi-block prefetch, more pragmatic hierarchical systems, and techniques for reducing the frequency of persistent requests. By using the substrate to ensure correctness, system designers can implement these optimizations with little impact on system complexity. These policies are just a few examples of the many possible applications of Token Coherence's flexibility.

Token Coherence may also be a good synergistic match for implementing recently proposed synchronization enhancements [105, 106, 107, 108]. Implementations of these proposals might benefit from Token Coherence's flexibility by allowing the system to defer responses and change the order in which processors receive responses or observe requests. It may even be possible to combine the conflict resolution techniques proposed for Transactional Lock Removal (TLR) [107] and our starvation-avoidance mechanism, resulting in one mechanism for handling these two somewhat similar issues. An investigation into combining these mechanisms might result in new insights into both proposals.

10.2.2 Is There a Better Way to Prevent Starvation?

In this dissertation, we presented several mechanisms for preventing starvation based upon persistent requests. Although the worst-case performance, traffic, implementation overhead, and complexity of these persistent request mechanisms appears reasonable, other approaches not yet invented may provide a better solution. However, a greatly simplified approach for preventing starvation would make Token Coherence a more attractive option for future multiprocessor systems.

In essence, if further research produced a better approach than token counting for enforcing safety, we would be disappointed; however, if further research produced a better approach for preventing starvation, we would be delighted.

10.2.3 Does Token Coherence Simplify Coherence Protocol Implementation?

Cache coherence protocols are traditionally complicated and error prone, and errors in the coherence protocol can cause system deadlocks (bad) or silently corrupt data (worse). These protocol errors are often difficult to diagnose because they can be subtle and difficult to reproduce. Coherence errors often manifest as unintended behavior in low-level system software (such as a system crash or software deadlock), further obscuring the design mistake. As a result, designers invest significant resources verifying the protocol's design and implementation by employing a combination of model checking, other formal methods, and extensive simulation-based design validation.

We believe (but have not yet convincingly shown) that Token Coherence is a simpler and less error prone approach to building a cache-coherent memory system. We describe four reasons this assertion might be true.

- First, some aspects of our token-based protocols are conceptually simpler than in traditional protocols. For example, writing back a dirty block in Token Coherence involves simply sending a message back the memory. No extra writeback acknowledgment messages or reasoning about complicated writeback races are needed. In contrast, writebacks in traditional protocols often create difficult to resolve races that substantially complicate the protocol. In Token Coherence, if a request and writeback occur at the same time, the request will simply be reissued later and likely find the block at the memory. In general, Token Coherence uses its ability to reissue requests as a powerful and unified mechanism for handling many types of races and uncommon-case situations.

- Second, the token-counting rules provide a strong and simple foundation for safety. Designers can focus the bulk of the verification effort on the safety aspect of the correctness substrate, and as long as these token rules are enforced, the state of the system cannot be

corrupted. For example, if a design mistake manifests while sending a transient request or when updating the soft-state directory, the system will still function correctly.

- Third, if a protocol design mistake causes incorrect behavior, deadlock or livelock is greatly preferred over data corruption.[1] Much of the new complexity in Token Coherence as described in this dissertation lies in the starvation prevention mechanism described in Chapter 4. An implementation error in such a mechanism might cause system livelock, but it will not corrupt system data. In some cases, software may interpose and recover from these deadlocks or livelocks by temporarily suspending system activity. Alternatively, the hardware may be able to halt the system, drain the interconnect, enter a slow-start mode, and eventually resume normal execution. Many current systems have such mechanisms, and they may be even more effective when applied to a system that uses Token Coherence.

- Fourth, the delegation of many performance enhancements to the performance policy enables simple correctness reasoning, even about complicated protocols. This ability allows for simple implementation of some protocols that might otherwise be too complicated to implement. For example, in our experience with both TOKENM and protocols based on the original Multicast Snooping protocol [20, 84, 114], TOKENM is much simpler than these other two protocols—even though both protocols exploit destination-set prediction. Feedback that we received on Multicast Snooping [20, 114] and our bandwidth adaptive protocol [84] indicated that while interesting academic proposals, these protocols would perhaps be too complicated to implement as proposed. Based on our table-driven specification of these protocols [114] and our experience with Token Coherence, we find that the versions of these protocols based on Token Coherence are much simpler.

In an effort to more solidly support out above assertions, we are actively working toward a more formal (and quantitative) understanding of Token Coherence's complexity.

[1] An example of this "fail-safe" design philosophy is using parity in memory; parity cannot correct memory errors (ECC is needed to correct bit errors), but a hardware parity error alerts the operating system to crash the system before the corrupt data affects the system. Colloquially, designers sometimes talk about using such a mechanism to "avoid issuing an (erroneous) million-dollar check."

10.3 Reflections on Cache-Coherent Multiprocessors

In this section, I share some observations and opinions concerning multiprocessor systems.[2] These opinions are based on five years of academic research on multiprocessor systems and workloads, many conversations with system designers, and a summer spent working in industry. These opinions are based my experiences, and they are just that: opinions. Although these opinions are not supported by hard evidence, they may provide insight to those interested in Token Coherence. Some of these options may be obvious and some may be controversial, but they reflect my current thoughts on multiprocessor system design; I reserve the right to change my mind about them later. I advocate optimizing for migratory sharing (Section 10.3.1), decoupling coherence and consistency (Section 10.3.2), avoiding relying on a total order of requests (Section 10.3.3), revisiting the snooping versus directory protocol debate (Section 10.3.4), designing cost-effective multiprocessor systems (Section 10.3.5), and embracing the increasing influence of chip multiprocessors on system design (Section 10.3.6).

10.3.1 Optimize For Migratory Sharing

Multiprocessors should optimize for migratory sharing using either the approach described in Section 2.2.4 or a similar approach [31, 115]. Anecdotal evidence from our past experiments using protocols that *did not* optimize for migratory sharing and our more recent experiment using protocols that *did* optimize for migratory sharing, indicates that such an optimization has a first-order performance impact in that it reduces the number of misses, improving both average memory access latency and system traffic. The Alpha 21364/G1280 multiprocessor [33, 91] implements such an optimization (although it is not well documented that it does), and I advocate that all future systems should also include such an optimization. Academic research evaluations should also use protocols that include this optimization to avoid skewing their results by capturing the same improvements that might be more easily captured by targeting migratory sharing patterns.

[2] To emphasize that my past and current co-authors may not share these opinions, I use singular pronouns in this section.

10.3.2 Decouple Coherence and Consistency

All cache coherence protocols should be designed to support sequential consistency. This statement, however, does not imply that all multiprocessor *systems* should support sequential consistency,[3] but rather the system should decouple coherence and consistency: the *coherence protocol* should provide a strict coherence interface and mechanisms sufficient to provide a serializable view of memory. The *processor* should interact with that interface to provide whatever memory model the processor designer deems necessary (using prefetching and various speculative techniques to aggressively implement the consistency model [6, 40, 43, 83, 98, 109, 126]).

The only role of the coherence protocol should be to inform the processor when it can read a block and when it can write a block. This notion closely corresponds with the multiple-reader-single-writer coherence invariant described earlier in Section 2.1, and it can be directly enforced using token counting (*i.e.*, all to write, at least one to read). Such a coherence protocol (1) provides a simple interface to the processor and (2) allows the processor to aggressively implement the desired consistency model. To clarify these points, consider the implementation of memory barrier/ordering operations (used to establish memory ordering). A system that follows the above philosophy can quickly handle memory barriers completely within the processor core, freeing the coherence protocol from providing special-purpose operations with complicated semantics.

Although some systems have adopted such a design philosophy (*e.g.*, the SGI Origin 2000 [72]), many systems do not. Those systems intertwine coherence and consistency through the system, often relying on a patchwork of special ordering properties of processor queues and a total order of requests via a totally-ordered interconnect to carefully and delicately orchestrate the desired consistency model.

[3] Although—like Hill [56]—I believe multiprocessor systems should support simple memory consistency models.

10.3.3 Avoid Reliance on a Total Order of Requests

We have argued in this dissertation that a protocol that relies on total order of requests[4] is undesirable due to the higher latency or cost of a totally-ordered interconnect. In this section I argue that relying on a total order is undesirable because it fundamentally intertwines the processing of different addresses.

In protocols that do not rely on a total order of requests—*e.g.*, traditional directory protocols (Section 2.5), AMD's Hammer protocol (Section 2.6), and all Token Coherence protocols—requests and responses for different blocks have no need to unduly influence each other. Only when a processor decides to commit a read or write to a block does the processor need to consider the interactions with reads and writes to other blocks to ensure the particular access is within the bounds allowed by the system's consistency model. In contrast, in systems that rely on a total order of requests, requests (and sometimes responses) for different addresses must be kept in order with respect to each other throughout the system. This requirement forces cache controllers to process the requests in order (or give the appearance that the requests were processed in order). To use a uniprocessor analogy, relying on a total order of requests introduces "dependences" (both true and false dependencies) between requests for various blocks.

More concretely, relying on a total order of requests has one minor advantage and three significant disadvantages. The minor advantage is that relying on a total order of requests allows the system to avoid explicit acknowledgment messages, because the total order enables the requester to infer when invalidations have logically occurred. In essence, invalidations occur in logical time and not physical time (as we briefly discussed in Section 2.1). The traffic savings of implicit acknowledgments is minor because the non-data acknowledgment messages are both small and sent infrequently (as shown in Figure 6.3 on page 108). In certain cases, implicit acknowledgments may reduce the latency of upgrade requests, but—as we argued in Section 2.2.5—upgrade requests are often infrequent (especially when a migratory sharing optimization is employed).

[4]Recall that we defined the term *total order* in Section 2.3. Systems that depend on a total order of requests include traditional snooping protocols (Section 2.4) and some directory protocols (*e.g.*, AlphaServer GS320 [41]).

First disadvantage: logical time issues. The reliance on logical time directly results in the first disadvantage of relying on a total order of requests: the coherence invariants must be specified in logical time, significantly complicating protocol verification. In such a system—even one that provides sequential consistency—one processor is allowed to read an “old” value at the same instant as another processor is reading a different, new value. Only after the processor reading the old value’s logical time advances (by ingesting requests that might involve many different addresses) is the processor forbidden from reading the old value. In contrast, consider a system that is both based on explicit invalidations (Token Coherence and traditional directory protocols) and subscribes to the philosophy of decoupling coherence and consistency (as described in Section 10.3.2). In such a system, processors are allowed to read only the most recently created value for a block; *i.e.*, only a single value for an address is allowed at any instant. We referred to this as the *coherence invariant* and discussed it earlier in Section 2.1. Enforcing such an invariant in physical time should simplify verification of the protocol. In my experience, debugging protocols that rely on a total order of requests[5] is more difficult, partially because of the difficulty in separating true bugs from false positives (events that were legal due to strange distortions in logical time). In contrast, I found it easier to develop protocols that use a strong physical-time coherence invariant, because it was simpler to detect and correct not just coherence bugs, but also subtle errors that would lead to consistency violations.

Second disadvantage: in-order handling of messages. The second disadvantage of relying on a total order of requests is that internal pathways and coherence controllers are perhaps more difficult to bank (in the same manner in which memory and caches are banked for increased bandwidth). For example, in a system that does *not* rely on a total order of requests, a second-level cache can be easily segmented into many independent banks, each with its own coherence controller that operates asynchronously with respect to the other banks. Each of these controllers need provide only a small amount of bandwidth, because—in aggregate—all the controllers together would provide sufficient bandwidth. Such unsynchronized banking is more difficult when the sys-

[5] I have collaborated with others to implement various snooping protocols, a GS320-like directory protocol [41], Timestamp Snooping [82], and Multicast Snooping [20, 114].

tem relies on a total order of requests. In such systems, the order in which requests are processed by a cache controller is restricted. Although the processor may be able to process invalidations early or delay its own incoming marker messages (borrowing the terminology used by Gharachorloo *et al.* [41]), other reordering may not be allowed. Enforcing these constraints complicates the design and requires interaction between requests from different banks. Protocols that avoid such restrictions can simplify high-performance controller design.

Third disadvantage: cost and latency of interconnect. The third disadvantage is the first-order latency and cost issues of implementing a totally-ordered interconnect (described in Section 2.3). Martin *et al.* [82] described both the problem and a technique to avoid these overheads; however the technique is complex, delays non-speculative responses, and still suffers from the first two disadvantages of relying on a total order of requests.

10.3.4 Revisit Snooping vs. Directory Protocols

The choice of coherence protocol is as complicated and subtle today as it has ever been, and much of my research has focused on the relative merits of snooping protocols and directory protocols. In the past I have advocated the approach of improving coherence protocols by enhancing snooping protocols to mitigate many of their disadvantages (*e.g.*, [82, 84, 114]). Unfortunately, these aggressive snooping protocols all rely on a total order of requests (and suffer from all the related disadvantages). Token Coherence was originally invented to address the most serious disadvantages of these aggressive and/or predictive snooping systems (*i.e.*, relying on a total order of requests and complexity of predictive approaches). TOKENB provides a snooping-like system without many of its disadvantages (a much better solution than Timestamp Snooping [82]) while TOKENM enables a less complex approach for using destination-set prediction. Thus, Token Coherence achieved the goal of enabling better snooping systems.

In retrospect, however, the disadvantages of directory protocols are perhaps mitigated more easily than the disadvantages of snooping protocols. The high latency of directory indirection can be significantly reduced (but not eliminated) using a directory cache, and the cost of storing directory information is greatly reduced by encoding these bits in the main memory's DRAM

(as described in Section 2.5.3). In addition, the complexity of directory protocols and aggressive broadcast snooping protocols is comparable, especially when considering that directory protocols are a better match for the philosophy of decoupling coherence and consistency (Section 10.3.2). When compared to snooping protocols, directory protocols still suffer extra indirection latency for cache-to-cache misses. However, I believe that directory protocols are perhaps preferable to traditional snooping protocols because of the cost, latency, and complexity advantages of avoiding a total order of requests and the benefits of cleanly separating coherence and consistency. Another attractive option might be to use a variant of one of the recent non-traditional protocols (*e.g.*, a more bandwidth-efficient variant of AMD's Hammer protocol [9, 121]).

Fortunately, Token Coherence is not just a proposal for improving snooping protocols. Instead, it is a framework for building a wide range of protocols that both subsumes and blurs the distinction between snooping and directory protocols. Such protocols (1) can act as a better form of snooping (*e.g.*, TOKENB) or as a directory protocol (*e.g.*, TOKEND), (2) avoid totally-ordered interconnects, (3) subscribe to the separation of coherence and consistency, (4) can exploit prediction of various forms (*e.g.*, TOKENM), and (5) are perhaps less complicated than aggressive implementations of traditional protocols.

10.3.5 Design Cost-Effective Multiprocessor Systems

Although the volume of academic research on multiprocessor computer architecture has declined in recent years [57], the volume of multiprocessor systems being sold has never been greater. The success of multiprocessor systems is not due to expensive large-scale multiprocessors with custom software (as many thought it would). Instead, the success of multiprocessor systems has been due to moderate-size servers and cost-effective cluster nodes running commodity operating systems. For example, vast computing infrastructures have been created by clustering cost-effective dual-processor systems (*e.g.*, Intel x86) using commodity local area networking (*e.g.*, gigabit Ethernet) running an open-source operating system (*e.g.*, Linux or a variant of BSD Unix). Multiprocessor systems are used as cluster nodes because of their cost effectiveness over single-processor systems (in terms of both hardware costs and total cost of ownership).

Cost effectiveness. The popularity of clusters has made cost effectiveness the primary design criteria of multiprocessor systems and has forever doomed the possibility of wide-spread adoption of scalable multiprocessors sold at a substantial price premium. Instead, multiprocessor designs need to focus on being the most cost-effective source of computational resources. A multiprocessor design should first be cost effective and then incrementally scaled to larger systems. A much more difficult route is to design a system to be infinitely scalable (at great expense and higher latency) and then trying to make the system cost effective.[6] Unfortunately, much academic research has focused on the second of these two approaches (by focusing on methods that allow for scaling systems to hundreds or thousands of processors).

Cost-effective multiprocessors via resource sharing. Multiprocessor systems can be more cost effective then uniprocessors—even if they are more expensive *per processor*—by using resources more efficiently by dynamically sharing them [124]. Multiprocessors allow many tasks to dynamically share a large pool of system-wide resources (*e.g.*, memory capacity, memory bandwidth, disk capacity, disk bandwidth, and local-area network bandwidth). By using these resources more efficiently, fewer resources are often required. Consider the following three examples:

- **Dynamic resource allocation.** Eight uniprocessor servers that each use 30% of a single disk's bandwidth (or capacity) could be coalesced into a single eight-processor server with four disks used at 60% utilization (or capacity), reducing system cost by eliminating four disks. Similarly, a cluster of uniprocessor servers with less than 100% average CPU utilization can be coalesced into a smaller number of servers with fewer total processors.[7]

- **Resource sharing.** Greater benefit results when resources are shared among the tasks. For example, consider several tasks that are all accessing the same 2GB read-only in-memory

[6] Hill [55] expressed a similar opinion in 1990 by stating "Thus, a company designing a new architecture for which initial, low-end systems will have four processors may wish to consider ramifications for 80-processor systems when making architectural decisions, but should probably implement the first system with a bus. Furthermore, systems implementors should consider using ugly, unscalable things (*e.g.*, buses and broadcast) if such things simplify the design, reduce system cost, and improve performance."

[7] System administrators can consolidate servers by simply running many tasks on one system, or they may use more sophisticated techniques such as logical system partitioning or multiplexing many virtual machines on one multiprocessor.

database. A cluster of eight uniprocessors might require 2GBs of DRAM *each* to cache the database (a total of 16GBs). In contrast, a single eight-processor system might require only 4GBs or 8GBs of DRAM *total* (by avoiding redundant copies of the database). Considering that DRAM is often a major source of system cost, such sharing of resources can greatly increase cost effectiveness [124].

- **Reducing resource fragmentation.** Finally, pooling resources reduces the amount of resources wasted due to fragmentation. For example, a user has three 1.2 GB jobs to run on two 2GB dual-processor machines. In this situation, the user can only run two jobs (a single job on each machine). If the resources were pooled into a 4GB quad-processor system, the user could run all three jobs in parallel.[8]

Although multiprocessor systems have many opportunities to provide cost-effective systems, moderate-sized multiprocessors are currently priced too expensively to provide cost-effective computation for many application domains. For example, although dual-processor systems are cost-effective (currently approximately $3,000 each), eight-processor systems are currently 10 to 30 times more expensive (ideally they would be only four times more expensive). Only when system cost is reduced through better design and system price is reduced by further competitive pressures will these mid-sized multiprocessors become more common (which, in turn will further reduce costs due to economies of scale). Currently, many companies price these systems to maintain high margins from customers that demand servers with dozens of processors (and are willing to pay the price premium).[9]

[8]This situation is not contrived; while generating results for this dissertation, I was often forced to run only a single simulation task on a dual-processor cluster node due to memory capacity limitations.

[9]I had hoped AMD's Opteron [9, 65, 121] (also known by its codename "Hammer") would have substantially reduced the price of 8-processor systems. However, as of this writing, Opteron systems have not yet become mainstream, and the Opteron processors for use in 4-processor and 8-processor systems are priced more than three times higher *per processor* than the almost-identical part for 2-processor systems (as of November 2003, $3199 versus $913 per processor for the fastest model, $749 versus $198 per processor for the slowest model). I believe that future products will be forced by market pressures to target more cost-effective design points by bringing unit cost and price in greater parity. Such market forces may present an opportunity for companies that either (1) do not have an established market for expensive mid-

10.3.6 The Increasing Importance of Chip Multiprocessors

The trend of increasingly integrated systems will continue to have a first-order effect on multiprocessor system design. We have argued in this dissertation that higher levels of integration are influencing multiprocessor design by making totally-ordered interconnects less attractive. The influence of increasing integration will perhaps be even stronger as increasing transistor budgets make chip multiprocessors (CMPs) commonplace. All major manufactures have either shipped or announced CMP designs. Just as the single-chip microprocessor provided a point of inflection in the implementation trade-offs for processors, CMPs will change the implementation of multiprocessor systems. CMPs will have several major effects on multiprocessors:

- CMPs will enable even more cost-effective multiprocessors by reducing the number of discrete components in the system.
- CMPs may change the design tradeoff between large processor cores (for highest uniprocessor performance using a large number of transistors) and more area-efficient cores (for moderate per-core performance using dramatically fewer transistors).
- Scalability will be less important in both intra-chip and inter-chip coherence protocols. A system with eight chips in the system, eight processors per chip, and four threads per processor has 256 virtual processors. In such a system, the coherence protocol will likely be arranged hierarchically, reducing the need for a protocol that focuses on scalability at any single level of the protocol.
- Such multi-level and hierarchical coherence protocols often add complexity to already complex systems.

The trend toward chip multiprocessors provides an opportunity for adoption of Token Coherence. Token Coherence might reduce the complexity of the system by replacing a true hierarchical

sized multiprocessors (*e.g.*, Intel, AMD, and Apple) or (2) are struggling to find new markets (*e.g.*, Sun Microsystems).

system with a single-level Token Coherence protocol that uses a performance policy to emulate the performance characteristics of a multi-level coherence protocol. Such an approach should provide the advantages of a multi-level protocol with less complexity. Token Coherence is also an attractive protocol for use with on-chip point-to-point interconnects. Although some might advocate using a bus-based snooping system on the chip, a highly-connected point-to-point interconnect avoids any centralized arbitration or structures. In essence, Token Coherence is a good choice for CMPs for the same reasons it is good choice for traditional multiprocessor systems.

Bibliography

[1] H. Abdel-Shafi, J. Hall, S. V. Adve, and V. S. Adve. An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, Feb. 1997.

[2] D. Abts, D. J. Lilja, and S. Scott. So Many States, So Little Time: Verifying Memory Coherence in the Cray X1. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2003.

[3] M. E. Acacio, J. González, J. M. García, and J. Duato. Owner Prediction for Accelerating Cache-to-Cache Transfers in a cc-NUMA Architecture. In *Proceedings of SC2002*, Nov. 2002. URL `http://doi.acm.org/10.1145/762761.762762`.

[4] M. E. Acacio, J. González, J. M. García, and J. Duato. The Use of Prediction for Accelerating Upgrade Misses in cc-NUMA Multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 155–164, Sept. 2002. URL `http://doi.acm.org/10.1145/645989.674321`.

[5] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, Dec. 1996.

[6] S. V. Adve, V. S. Pai, and P. Ranganathan. Recent Advances in Memory Consistency Models for Hardware Shared Memory Systems. *Proceedings of the IEEE*, 87(3):445–455, Mar. 1999.

[7] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995. URL `http://doi.acm.org/10.1145/223982.223985`.

[8] A. Agarwal, R. Simoni, M. Horowitz, and J. Hennessy. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, May 1988.

[9] A. Ahmed, P. Conway, B. Hughes, and F. Weber. AMD Opteron Shared Memory MP Systems. In *Proceedings of the 14th HotChips Symposium*, Aug. 2002. URL `http://www.hotchips.org/archive/hc14/program/28_AMD_Hammer_MP_HC_v8.pdf`.

[10] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, D. J. Sorin, M. D. Hill, and D. A. Wood. Simulating a $2M Commercial Server on a $2K PC. *IEEE Computer*, 36(2):50–57, Feb. 2003. URL http://dx.doi.org/10.1109/MC.2003.1178046.

[11] A. R. Alameldeen and D. A. Wood. Variability in Architectural Simulations of Multi-threaded Workloads. In *Proceedings of the Ninth IEEE Symposium on High-Performance Computer Architecture*, pages 7–18, Feb. 2003.

[12] C. Anderson and A. R. Karlin. Two Adaptive Hybrid Cache Coherency Protocols. In *Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, Feb. 1996.

[13] J. Archibald and J.-L. Baer. An Economical Solution to the Cache Coherence Problem. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 355–362, June 1984. URL http://doi.acm.org/10.1145/800015.808205.

[14] J. Archibald and J.-L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, Nov. 1986. URL http://doi.acm.org/10.1145/6513.6514.

[15] M. Azimi, F. Briggs, M. Cekleov, M. Khare, A. Kumar, and L. P. Looi. Scalability Port: A Coherent Interface for Shared Memory Multiprocessors. In *Proceedings of the 10th Hot Interconnects Symposium*, pages 65–70, Aug. 2002. URL http://www.hoti.org/archive/hoti10/program/Kumar_ScalabilityPort.pdf.

[16] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the 1998 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 151–160, June 1998.

[17] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 3–14, June 1998.

[18] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293, June 2000.

[19] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 125–135, May 1990.

[20] E. E. Bilir, R. M. Dickson, Y. Hu, M. Plakal, D. J. Sorin, M. D. Hill, and D. A. Wood. Multicast Snooping: A New Coherence Method Using a Multicast Address Network. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 294–304, May 1999.

[21] J. Borkenhagen and S. Storino. 4th Generation 64-bit PowerPC-Compatible Commercial Processor Design. IBM Server Group Whitepaper, Jan. 1999. URL `http://www.ibm.com/servers/eserver/pseries/hardware/whitepapers/nstar.html`.

[22] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A Multithreaded PowerPC Processor for Commercial Servers. *IBM Journal of Research and Development*, 44(6):885–898, Nov. 2000.

[23] J. M. Borkenhagen, R. D. Hoover, and K. M. Valk. EXA Cache/Scalability Controllers. In *IBM Enterprise X-Architecture Technology: Reaching the Summit*, pages 37–50. International Business Machines, 2002.

[24] L. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, Dec. 1978.

[25] A. Charlesworth. Starfire: Extending the SMP Envelope. *IEEE Micro*, 18(1):39–49, Jan/Feb 1998.

[26] A. Charlesworth. The Sun Fireplane Interconnect. In *Proceedings of SC2001*, Nov. 2001.

[27] A. Charlesworth. The Sun Fireplane SMP Interconnect in the Sun 6800. In *Proceedings of the 9th Hot Interconnects Symposium*, Aug. 2001.

[28] C. L. Chen and M. Y. Hsiao. Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review. *IBM Journal of Research and Development*, 28(2), Mar. 1984.

[29] A. Condon and A. J. Hu. Automatable Verification of Sequential Consistency. In *Proceedings of the Thirteenth ACM Symposium on Parallel Algorithms and Architectures*, pages 113–121, July 2001.

[30] A. E. Condon, M. D. Hill, M. Plakal, and D. J. Sorin. Using Lamport Clocks to Reason About Relaxed Memory Models. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, pages 270–278, Jan. 1999.

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

[32] D. E. Culler and J. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., 1999.

[33] Z. Cvetanovic. Performance analysis of the Alpha 21364-based HP GS1280 multiprocessor. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 218–229, June 2003. URL `http://doi.acm.org/10.1145/859618.859643`.

[34] F. Dahlgren. Boosting the Performance of Hybrid Snooping Cache Protocols. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 60–69, June 1995.

[35] W. J. Dally and J. W. Poulton. *Digital Systems Engineering*. Cambridge University Press, 1998.

[36] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann, revised edition, 2003.

[37] B. Falsafi and D. A. Wood. Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 229–240, June 1997.

[38] S. J. Frank. Tightly Coupled Multiprocessor System Speeds Memory-access Times. *Electronics*, 57(1):164–169, Jan. 1984.

[39] K. Gharachorloo, L. A. Barroso, and A. Nowatzyk. Efficient ECC-Based Directory Implementations for Scalable Multiprocessors. In *Proceedings of the 12th Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD 2000)*, Oct. 2000.

[40] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the International Conference on Parallel Processing*, volume I, pages 355–364, Aug. 1991.

[41] K. Gharachorloo, M. Sharma, S. Steely, and S. V. Doren. Architecture and Design of AlphaServer GS320. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 13–24, Nov. 2000.

[42] P. B. Gibbons, M. Merritt, and K. Gharachorloo. Proving Sequential Consistency of High-Performance Shared Memories. In *Proceedings of the Third ACM Symposium on Parallel Algorithms and Architectures*, pages 292–303, July 1991.

[43] C. Gniady, B. Falsafi, and T. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 162–171, May 1999.

[44] A. Gupta and W.-D. Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 41(7):794–810, July 1992.

[45] 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 (ICPP)*, volume I, pages 312–321, 1990. URL `http://citeseer.nj.nec.com/gupta90reducing.html`.

[46] D. Gustavson. The Scalable Coherent Interface and related standards projects. *IEEE Micro*, 12(1):10–22, Feb. 1992.

[47] L. Gwennap. Alpha 21364 to Ease Memory Bottleneck. *Microprocessor Report*, Oct. 1998.

[48] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, pages 172–181, Jan. 1999.

[49] E. Hagersten and G. Papadopoulos. Parallel Computing in the Commercial Marketplace: Research and Innovation at Work. *Proceedings of the IEEE*, 87(3):405–411, March 1999.

[50] J. Heinlein, R. P. Bosch, K. Gharachorloo, M. Rosenblum, and A. Gupta. Coherent Block Data Transfer in the FLASH Multiprocessor. In *Proceedings of the 11th International Parallel Processing Symposium*, pages 18–27, Apr. 1997. URL `http://citeseer.nj.nec.com/heinlein97coherent.html`.

[51] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994.

[52] M. Heinrich, V. Soundararajan, J. Hennessy, and A. Gupta. A Quantitative Analysis of the Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols. *IEEE Transactions on Computers*, 28(2):205–217, Feb. 1999.

[53] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, third edition, 2003.

[54] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Verifying Sequential Consistency on Shared-Memory Multiprocessor Systems. *Lecture Notes in Computer Science*, 1633:301–315, 1999.

[55] M. D. Hill. What is Scalability? *Computer Architecture News*, 18(4):18–21, 1990. URL `http://doi.acm.org/10.1145/121973.121975`.

[56] M. D. Hill. Multiprocessors Should Support Simple Memory Consistency Models. *IEEE Computer*, 31(8):28–34, Aug. 1998.

[57] M. D. Hill and R. Rajwar. The Rise and Fall of Multiprocessor Papers in the International Symposium on Computer Architecture (ISCA). http://www.cs.wisc.edu/ markhill/mp2001.html, Mar. 2001.

[58] T. Horel and G. Lauterbach. UltraSPARC-III: Designing Third Generation 64-Bit Performance. *IEEE Micro*, 19(3):73–85, May/June 1999.

[59] M. Horowitz, C.-K. K. Yang, and S. Sidiropoulos. High-Speed Electrical Signaling: Overview and Limitations. *IEEE Micro*, 18(1), Jan/Feb 1998.

[60] T. Juhnke and H. Klar. Calculation of the Soft Error Rate of Submicron CMOS Logic Circuits. *IEEE Journal of Solid-State Circuits*, 30(7):830–834, July 1995.

[61] S. Kaneda. A Class of Odd-Weight-Column SEC-DED-SbED Codes for Memory System Applications. *IEEE Transactions on Computers*, 33(8):737–739, Aug. 1984.

[62] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive Snoopy Caching. *Algorithmica*, 3(1):79–119, 1988.

[63] S. Kaxiras and J. R. Goodman. Improving CC-NUMA Performance Using Instruction-Based Prediction. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, Jan. 1999.

[64] S. Kaxiras and C. Young. Coherence Communication Prediction in Shared-Memory Multiprocessors. In *Proceedings of the Sixth IEEE Symposium on High-Performance Computer Architecture*, Jan. 2000.

[65] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway. The AMD Opteron Processor for Multiprocessor Servers. *IEEE Micro*, 23(2):66–76, March-April 2003.

[66] D. A. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas. Data Forwarding in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 1995 International Conference on Supercomputing*, July 1995.

[67] S. Kunkel. Personal Communication, Apr. 2000.

[68] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. C. apin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, Apr. 1994.

[69] A.-C. Lai and B. Falsafi. Memory Sharing Predictor: The Key to a Speculative Coherent DSM. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 172–183, May 1999.

[70] A.-C. Lai and B. Falsafi. Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 139–148, June 2000.

[71] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.

[72] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[73] A. R. Lebeck and D. A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 48–59, June 1995.

[74] 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 International Symposium on Computer Architecture*, pages 148–159, May 1990.

[75] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, Mar. 1992.

[76] T. D. Lovett and R. M. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23th Annual International Symposium on Computer Architecture*, May 1996.

[77] P. S. Magnusson *et al.* Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2): 50–58, Feb. 2002.

[78] M. M. K. Martin *et al.* Protocol Specifications and Tables for Four Comparable MOESI Coherence Protocols: Token Coherence, Snooping, Directory, and Hammer, 2003. URL `http://www.cs.wisc.edu/multifacet/theses/milo_martin_phd/`.

[79] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared Memory Multiprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 206–217, June 2003.

[80] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token Coherence: A New Framework for Shared-Memory Multiprocessors. *IEEE Micro*, November-December 2003.

[81] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 182–193, June 2003.

[82] M. M. K. Martin, D. J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore, M. Plakal, M. D. Hill, and D. A. Wood. Timestamp Snooping: An Approach for Extending SMPs. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–36, Nov. 2000.

[83] M. M. K. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. H. Lipasti. Correctly Implementing Value Prediction in Microprocessors that Support Multithreading or Multiprocessing. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2001.

[84] M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Bandwidth Adaptive Snooping. In *Proceedings of the Eighth IEEE Symposium on High-Performance Computer Architecture*, pages 251–262, Feb. 2002.

[85] J. R. Mashey. NUMAflex Modular Design Approach: A Revolution in Evolution. Posted on comp.arch news group, Aug. 2000.

[86] C. J. Mauer, M. D. Hill, and D. A. Wood. Full System Timing-First Simulation. In *Proceedings of the 2002 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 108–116, June 2002.

[87] T. May and M. Woods. Alpha-Particle-Induced Soft Errors in Dynamic Memories. *IEEE Transactions on Electronic Devices*, 26(2), 1979.

[88] D. Milojicic, A. Messer, J. Shau, G. Fu, and A. Munoz. Increasing Relevance of Memory Hardware Errors: A Case for Recoverable Programming Models. In *9th ACM SIGOPS European Workshop 'Beyond the PC: New Challenges for the Operating System'*, Sept. 2000.

[89] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. JETTY: Filtering Snoops for Reduced Power Consumption in SMP Servers. In *Proceedings of the Seventh IEEE Symposium on High-Performance Computer Architecture*, Jan. 2001.

[90] F. Mounes-Toussi and D. J. Lilja. The Potential of Compile-Time Analysis to Adapt the Cache Coherence Enforcement Strategy to the Data Sharing Characteristics. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):470–481, May 1995.

[91] S. S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb. The Alpha 21364 Network Architecture. In *Proceedings of the 9th Hot Interconnects Symposium*, Aug. 2001.

[92] S. S. Mukherjee and M. D. Hill. Using Prediction to Accelerate Coherence Protocols. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 179–190, June 1998.

[93] A. K. Nanda, A.-T. Nguyen, M. M. Michael, and D. J. Joseph. High-Throughput Coherence Controllers. In *Proceedings of the Sixth IEEE Symposium on High-Performance Computer Architecture*, Jan. 2000.

[94] J. Nilsson and F. Dahlgren. Improving Performance of Load-Store Sequences for Transaction Processing Workloads on Multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 246–255, Sept. 1999.

[95] J. Nilsson and F. Dahlgren. Reducing Ownership Overhead for Load-Store Sequences in Cache-Coherent Multiprocessors. In *Proceedings of the 2000 International Parallel and Distributed Processing Symposium*, May 2000.

[96] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, and M. Parkin. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the International Conference on Parallel Processing*, volume I, pages 1–10, Aug. 1995.

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

[98] V. Pai, P. Ranganathan, S. Adve, and T. Harton. An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 273–298, Oct. 1996.

[99] W. W. Peterson and E. J. Weldon, Jr. *Error-Correcting Codes*. MIT Press, 1972.

[100] M. Plakal, D. J. Sorin, A. E. Condon, and M. D. Hill. Lamport Clocks: Verifying a Directory Cache-Coherence Protocol. In *Proceedings of the Tenth ACM Symposium on Parallel Algorithms and Architectures*, pages 67–76, June 1998.

[101] F. Pong and M. Dubois. Verification Techniques for Cache Coherence Protocols. *ACM Computing Surveys*, 29(1):82–126, Mar. 1997.

[102] D. Poulsen and P.-C. Yew. Data Prefetching and Data Forwarding in Shared-Memory Multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 296–280, Aug. 1994.

[103] I. Pragaspathy and B. Falsafi. Address Partitioning in DSM Clusters with Parallel Coherence Controllers. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2000.

[104] S. Qadeer. On the Verification of Memory Models of Shared-Memory Multiprocessors. In *Tutorial and Workshop on Formal Specification and Verification Methods for Shared Memory Systems*, Oct. 2000.

[105] R. Rajwar. *Speculation-Based Techniques for Transactional Lock-Free Execution of Lock-Based Programs*. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, 2002.

[106] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2001.

[107] R. Rajwar and J. R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[108] R. Rajwar, A. Kagi, and J. R. Goodman. Inferential Queueing and Speculative Push for Reducing Critical Communication Latencies. In *Proceedings of the 17th International Conference on Supercomputing*, June 2003.

[109] P. Ranganathan, V. S. Pai, and S. V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *Proceedings of the Ninth ACM Symposium on Parallel Algorithms and Architectures*, pages 199–210, June 1997.

[110] A. Raynaud, Z. Zhang, and J. Torrellas. Distance-Adaptive Update Protocols for Scalable Shared-Memory Multiprocesors. In *Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, Feb. 1996.

[111] E. Rosti, E. Smirni, T. Wagner, A. Apon, and L. Dowdy. The KSR1: Experimentation and Modeling of Poststore. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 74–85, May 1993.

[112] X. Shen and Arvind. Specification of Memory Models and Design of Provably Correct Cache Coherence Protocols. Group Memo 398, Massachusetts Institute of Technology, June 1997.

[113] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yaun, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvery, E. Hagersten, and B. Liencres. Gigaplane: A High Performance Bus for Large SMPs. In *Proceedings of the 4th Hot Interconnects Symposium*, pages 41–52, Aug. 1996.

[114] D. J. Sorin, M. Plakal, M. D. Hill, A. E. Condon, M. M. K. Martin, and D. A. Wood. Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):556–578, June 2002.

[115] P. Stenström, M. Brorsson, and L. Sandberg. Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[116] P. Sweazey and A. J. Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 414–423, June 1986.

[117] C. K. Tang. Cache Design in the Tightly Coupled Multiprocessor System. In *Proceedings of the AFIPS National Computing Conference*, pages 749–753, June 1976.

[118] D. M. Taub. Improved Control Acquisition Scheme for the IEEE 896 Futurebus. *IEEE Micro*, 7(3), June 1987.

[119] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. IBM Server Group Whitepaper, Oct. 2001. URL http://www.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.pdf.

[120] M. K. Vernon and U. Manber. Distributed round-robin and first-come first-serve protocols and their applications to multiprocessor bus arbitration. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 289–279, May 1988. URL http://doi.acm.org/10.1145/633625.52431.

[121] F. Weber. AMD's Next Generation Microprocessor Architecture, Oct. 2001. URL http://www.amd.com/us-en/assets/content_type/DownloadableAssets/MPF_Hammer_Presentation.PDF.

[122] W.-D. Weber and A. Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, Apr. 1989.

[123] 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 20th Annual International Symposium on Computer Architecture*, May 1993.

[124] D. A. Wood and M. D. Hill. Cost-Effective Parallel Computing. *IEEE Computer*, pages 69–72, Feb. 1995.

[125] Q. Yang, G. Thangadurai, and L. N. Bhuyan. Design of Adaptive Cache Coherence Protocol for Large Scale Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3 (3):281–293, May 1992.

[126] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, Apr. 1996.

Appendix A: Differences from Martin *et al.*, ISCA 2003

This appendix summarizes the most important differences between this dissertation and our prior work [79, 80, 81]. This dissertation extends our earlier work which introduced Token Coherence [80, 81] by refining the token-counting rules, describing the overheads associated with token counting, introducing another approach for implementing persistent requests, and evaluating three distinct performance policies (in contrast, the earlier work explored only one performance policy). One of these performance policies, TOKENM, is based upon the destination set predictors described by Martin *et al.* [79].

The following is a list in text order of the most important differences from our prior work.

- **Additional Background.** The coherence background in Chapter 2 emphasizes Token Coherence's relationship to the traditional MOESI-based coherence protocols. Chapter 2 also provides additional background information on our base coherence protocols and interconnects.

- **Token Counting.** The description of token counting in Chapter 3 significantly refines the token counting rules (called invariants in the original paper). This chapter introduces the notion of a clean/dirty owner token for fully supporting the EXCLUSIVE state. We also discuss support for cache allocation instructions that avoid data transfers (such as Alpha's Write Hint 64 instruction). Finally, Section 3.2 contains a detailed discussion of token storage and manipulation overheads including (1) token storage in caches, (2) transferring tokens in messages, (3) non-silent cache evictions, and (4) token storage in memory.

- **Persistent Requests.** The discussion of persistent requests in Chapter 4 significantly extends our prior work. The most important addition is the introduction of a new distributed-arbitration approach to implementing persistent requests. We also introduce persistent read requests. In contrast, our prior work described only a single persistent request that gathered all tokens. The chapter sketches an argument for why persistent requests prevent starvation. Finally, the chapter also includes a substantial discussion of advantages and disadvantages

of several implementation alternatives for preventing the reordering of persistent request messages.

- **Performance Policies.** Chapter 5 describes Token Coherence's performance policies (previously called performance protocols) and discusses other possible performance protocols not further explored. The most significant difference between this dissertation and our prior work is that we explore three performance policies (Chapter 7, Chapter 8, and Chapter 9). In contrast, our prior work only explored a single performance policy. The inclusion of these two additional performance policies is critical for showing the flexibility of Token Coherence.

- **Quantitative Evaluation.** The methods for our experimental evaluation have also improved (Chapter 6). The workloads we use have been improved (larger working sets, larger number of concurrent users, and more recent versions of system software). We also provide a brief quantitative characterization of the new versions of these workloads. We evaluate three different performance protocols, and we report traffic metrics in terms of messages per miss (in addition to the bytes per miss presented in our prior work).

- **Destination-Set Prediction Applied to Token Coherence.** Chapter 9 presents a performance protocol that is a straightforward application of destination-set prediction to Token Coherence. We used the destination-set predictors for multicast snooping we found to be the best in our prior work [79], and we do not repeat the detailed design space exploration included in this original paper. As described in Chapter 9, the predictors were modified slightly to work within the Token Coherence framework.