CONTEXT-AWARE COHERENCE PROTOCOLS FOR FUTURE PROCESSORS

by

Liqun Cheng

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

May 2007

Copyright © Liqun Cheng 2007

All Rights Reserved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Liqun Cheng

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: John B. Carter

Rajeev Balasubramonian

Faye A. Briggs

Al Davis

Ganesh Gopalakrishnan

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Liqun Cheng in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

John B. Carter
Chair: Supervisory Committee

Approved for the Major Department

Martin Berzins
Chair/Director

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

The semiconductor industry is experiencing a shift from "computation-bound design" to "communication-bound design." Many future systems will use one or many chip multiprocessors (CMPs) and support shared memory, as CMP-based systems can provide high-performance, cost-effective computing for workloads with abundant thread-level parallelism. One of the biggest challenges in CMP designs is to employ efficient cache coherence protocols to maintain coherence.

Most commercial products implement directory-based protocols to maintain coherence. Directory-based protocols avoid the bandwidth and electrical limits of a centralized interconnect by tracking the global coherence state of cache lines via a directory structure. Directory-based protocols are preferred in future systems due to their ability to exploit arbitrary point-to-point interconnects. However, existing directory-based protocols are not optimum in specific contexts. For example, existing protocols cannot optimize the coherence traffic according to the sharing patterns shown in the applications or the varying latency, bandwidth needs of different coherence messages, which leads to suboptimal performance and extra power consumption.

To overcome these limitations of conventional directory-based protocols, we propose two cache coherence protocols that optimize the coherence traffic based on context knowledge. After automatically detecting a stable producer-consumer pattern in an application, our sharing pattern-aware coherence protocol uses *directory delegation* to delegate the "home directory" of a cache line to the producer node, thereby converting 3-hop coherence operations into 2-hop operations. Then, the producer employs *speculative updates* to push the data to where it might soon be consumed. When the producer correctly predicts when and where to send updates, these 3-hop misses become local misses, effectively eliminating the

impact of remote memory latency. Our interconnect-aware protocol can exploit a heterogenous interconnect comprised of wires with varying latency, bandwidth, and energy characteristics. By intelligently mapping critical messages to wires optimized for delay and noncritical messages to wires optimized for low power, our interconnect-aware protocol can achieve performance improvement and power reduction at the same time.

We demonstrate the performance advantage of the proposed mechanisms through architecture-level simulation. The producer-consumer sharing-aware protocol reduces the average remote miss rate by 40%, reduces network traffic by 15%, and improves performance by 21% on seven benchmark programs that exhibit producer-consumer sharing using a cycle-accurate simulator of a future 16-node SGI multiprocessor. The interconnect-aware protocol yields a performance improvement of 11% and energy reduction of 22% on a set of scientific and commercial benchmarks for a future 16-core CMP system.

CONTENTS

ABSTRACT . . . . **iv**

LIST OF FIGURES . . . . **ix**

LIST OF TABLES . . . . **xi**

CHAPTERS

1. INTRODUCTION **1**

1.1 Cache Coherence Protocol Design Space 2
1.1.1 Low-latency Coherence Misses 3
1.1.2 No Reliance on Bus Interconnects 4
1.1.3 Low Power Consumption 5
1.1.4 Simple Programming Model 5
1.1.5 Low Bandwidth Usage 6
1.2 Context of Cache Coherence Protocols 7
1.3 Thesis Statement 10
1.4 Contributions 11
1.5 Outline 12

2. BACKGROUND AND RELATED WORK **13**

2.1 Caching in Multiprocessor Systems 13
2.2 Common Coherence States 17
2.2.1 MESI Protocol 17
2.2.2 MOESI Protocols 18
2.3 Common Coherence Protocols 20
2.3.1 Bus-based Protocols 20
2.3.2 Ring-based Protocols 21
2.3.3 Token-based Protocols 22
2.3.4 Directory-based Protocols 23
2.4 Related Work 25
2.4.1 Sharing Pattern-aware Protocols Related 25
2.4.2 Interconnect-aware Protocols Related 27

3. SHARING PATTERN-AWARE COHERENCE PROTOCOLS **30**

3.1 Producer-consumer Sharing 31
3.2 Protocol Implementation 35
3.2.1 Remote Access Cache 36
3.2.2 Sharing Pattern Detection 36

3.2.3 Directory delegation 38
3.2.3.1 Delegation 39
3.2.3.2 Request forwarding 40
3.2.3.3 Undelegation 41
3.2.3.4 Discussion 42
3.2.4 Speculative updates 42
3.2.4.1 Delayed Intervention 43
3.2.4.2 Selective Updates 44
3.2.4.3 Summary of Control Flow 45
3.2.5 Verification 46
3.3 Evaluation 47
3.3.1 Simulator Environment 47
3.3.2 Results 48
3.3.3 Sensitivity Analysis 54
3.3.3.1 Equal Storage Area Comparison 54
3.3.3.2 Sensitivity to Intervention Delay Interval 56
3.3.3.3 Sensitivity to Network Latency 58
3.3.3.4 RAC and Delegate Cache 59
3.4 Summary 60

4. INTERCONNECT-AWARE COHERENCE PROTOCOLS . . . 62

4.1 Background 63
4.2 Wire Implementations 64
4.3 Optimizing Coherence Traffic 66
4.3.1 Protocol-dependent Techniques 67
4.3.1.1 Proposal I: Write for Shared Data 68
4.3.1.2 Proposal II: Read for Exclusive Data 69
4.3.1.3 Proposal III: NACK Messages 70
4.3.1.4 Proposal IV: Unblock and Write Control 70
4.3.1.5 Proposal V: Signal Wires 71
4.3.1.6 Proposal VI: Voting Wires 71
4.3.2 Protocol-independent Techniques 72
4.3.2.1 Proposal VII: Synchronization Variables 72
4.3.2.2 Proposal VIII: Writeback Data 72
4.3.2.3 Proposal IX: Narrow Messagess 72
4.3.3 Implementation Complexity 73
4.3.3.1 Implementation Overhead 73
4.3.3.2 Overhead in Decision Process 74
4.3.3.3 Overhead in Cache Coherence Protocols 75
4.4 Results 75
4.4.1 Methodology 75
4.4.1.1 Simulator 75
4.4.1.2 Interconnect Power/Delay/Area Models 77
4.4.2 Results for Scientific Benchmarks 80
4.4.3 Results for Commercial Benchmarks 84
4.4.4 Sensitivity Analysis 87

4.4.4.1 Out-of-order/In-order Processors 87
4.4.4.2 Link Bandwidth 88
4.4.4.3 Routing Algorithm 89
4.4.4.4 Network Topology 89
4.5 Summary 91

5. CONCLUSIONS **92**

REFERENCES **94**

LIST OF FIGURES

1.1 Context knowledge for cache coherence protocols. 8
1.2 An example of 3-hop miss 9
2.1 Write invalidate and write update 16
2.2 Three characteristics of cached data. 19
3.1 Coherence operations at the producer 32
3.2 Coherence operations at the consumer 33
3.3 Modeled node architecture (new components are highlighted) 35
3.4 Format of delegate cache table entries 39
3.5 Directory delegation 39
3.6 Request forwarding 40
3.7 Directory undelegation 41
3.8 Flow of speculative updates 45
3.9 Application speedup 49
3.10 Application network messages 49
3.11 Application remote misses 50
3.12 Equal storage area comparison (larger L2 caches) 54
3.13 Equal storage area comparison (larger directory caches) 55
3.14 Sensitivity to intervention delay 56
3.15 Delayed intervention VS. updates issued by processor 57
3.16 Sensitivity to hop latency (Appbt) 58
3.17 Sensitivity to delegate cache sz (MG) 59
3.18 Sensitivity to RAC size (Appbt) 60
4.1 Examples of different wire implementations 66
4.2 Read exclusive request for a shared block in MESI protocol 68
4.3 Interconnect model for a 16-core CMP 77
4.4 Speedup of heterogeneous interconnect 80
4.5 Distribution of messages on the heterogeneous network 81

4.6 Distribution of L-message transfers across different proposals. 82
4.7 Improvement in link energy and ED^2. 84
4.8 Speedup in commercial applications. 86
4.9 Power reduction in commercial applications. 87
4.10 Speedup of heterogeneous interconnect when driven by OoO cores . . . 88
4.11 2D torus topology. 89
4.12 Heterogeneous interconnect speedup in 2D torus. 90

LIST OF TABLES

2.1 MESI state transitions 18
2.2 MOESI state transitions 26
3.1 System configuration. 47
3.2 Applications and data sets 48
3.3 Number of consumers in the producer-consumer sharing patterns 51
4.1 Power characteristics of different wire implementations) is assumed to be 0.15. The above latch spacing values are for a 5GHz network. 73
4.2 System configuration. 76
4.3 Area, delay, and power characteristics of wires 78
4.4 Energy consumed by arbiters, buffers, and crossbars 79

CHAPTER 1

INTRODUCTION

Improvements in semiconductor technology have led to a tremendous increase in the clock speed and transistor counts of single-chip devices. Higher clock speeds along with novel architecture techniques have improved microprocessor performance by orders of magnitude in the last decade [29]. An emerging trend in processor microarchitecture design, driven by both technology and marketing initiatives, is a move towards chip multiprocessors (CMPs). CMPs employ multiple independent processing cores on a single processing chip. The individual cores in a CMP are slower than the processors found in typical uniprocessor chips. However, they consume less power, generate less heat, and in aggregate have more potential compute power. In spite of dramatic improvements in a single chip's performance, higher performance is demanded to solve large-scale scientific and commercial problems.

To surpass the computational power offered by a single chip, computer designers have studied techniques to connect two or more microprocessors and make them cooperate on common computational tasks. These multiprocessors are designed with two major approaches: bus-based multiprocessors and distributed shared memory (DSM) multiprocessors. For small numbers of processors, the dominant architecture employs a shared bus that joins the processors and one or more main memory modules. Such systems are called symmetric multiprocessors (SMPs), as all processors have equal access to the centralized main memory.

DSM systems are designed to scale to larger numbers of processors. These systems provide the abstraction of one global memory, which is physically distributed among nodes. Programs access data by referencing an address in the global memory space. DSM architectures transparently fetch the data, regardless of its physical

location, to satisfy the request. This abstraction matches a programmer's intuition developed on SMPs. Programmers can develop working programs without worrying about data distribution and communication. However, DSM machines hide the underlying distributed memory organization too well at times. Programs which are naive about data distribution and communication are unlikely to perform well. In a DSM system, the access latency to data depends on the memory module in which the data resides. Remote memory access latency can be orders of magnitude higher than local memory access latency. Because of the resulting variability in memory access time, DSM machines are also referred as Non-Uniform Memory Access (NUMA) machines.

Caches are used in the multiprocessors to hide memory latencies. Multiprocessor caches are more complicated than uniprocessor caches because they introduce a new problem, *cache coherence* that arises when multiple processors cache and update the same memory location. Solving this problem in hardware requires a *cache coherence protocol* that enforces the desired memory model and guarantees a load of any memory location still returns the latest value written in that location. There are two major types of cache coherence protocols: *bus-based protocols* [33] for small systems, and *directory-based protocols* [83] for large-scale shared memory systems. Chapter 2 details bus-based and directory-based protocols, along with other recently proposed protocols like ring-based [112] and token-based protocols [95].

1.1 Cache Coherence Protocol Design Space

Commercial multiprocessors use a variety of bus-based protocols [33, 35, 36, 37] or directory-based protocols [62, 70, 79, 83, 85]. A number of other protocols have also been proposed in academia [40, 55, 79, 107]. We will explore the design space by identifying five desirable attributes of cache coherence protocols: low-latency coherence misses (Section 1.1.1), no reliance on bus interconnects (Section 1.1.2), low power consumption (Section 1.1.3), simple programming model (Section 1.1.4) and low bandwidth usage (Section 1.1.5). Unfortunately, no existing cache coherence protocol exhibits all five of these attributes. This limitation of

the current approaches motivates our proposal to employ context-aware cache coherence protocols that can improve these attributes within a given context. Our sharing-pattern aware protocol can improve performance and bandwidth efficiency (Chapter 3). Our interconnect-aware protocol can improve performance and reduce power consumption (Chapter 4).

1.1.1 Low-latency Coherence Misses

Cache misses in uniprocessors are typically categorized into three categories: compulsory, capacity and conflict misses [63]. Compulsory misses, also known as cold misses, refer to those cache misses caused by the first access to a particular memory line by a processor. Capacity misses occur when all lines that are referenced by a processor during the execution of a program do not fit in the cache, so some lines are replaced and later accessed again. Conflict misses occur in caches with less than full associativity when the collection of lines referenced by a program that maps to the same cache set does not fit in the set. Multiprocessors introduce a new category of cache misses: coherence misses. Coherence misses occur when different processors read and write to the same block.

Modern processors have large caches with high set associativity, which reduces the number of capacity and conflict misses. Therefore, coherence misses are often the performance limiters in shared memory multiprocessors. Barroso et al. [21] and Martin et al. [96] report that about one-half of all off-chip memory references are coherence misses for commercial workloads. To reduce the latency of coherence misses, a coherence protocol should ideally support direct cache-to-cache transfer. For example, bus-based protocols support fast cache-to-cache transfer by broadcasting all requests to locate the owner directly. In contrast, directory protocols indirectly find remote data by placing a directory lookup first, which incurs a 3 hop traversal for cache-to-cache misses.

Meanwhile, memory and interconnect latencies are not keeping up with the dramatic increases in processor performance. The coherence miss latency measured in processor cycles is increasing. On a machine that is several meters across, such

as the 10,000 processor SGI Altix "Columbia" system that SGI recently sold to NASA [1], even if the request and response move at the speed of light, with no processing delays, the latency of accessing remote data on a modern 4GHz processor is hundreds of processor cycles. Therefore, reducing the impact of remote coherence misses becomes more and more important.

1.1.2 No Reliance on Bus Interconnects

The interconnection network is a key component of a multiprocessor system. Early bus-based protocols rely on a bus or bus-like interconnect, which provides a total order of requests. An interconnect provides a total order if all messages are delivered to all destinations in the same order. For example, if message A is sent before message B, then all processors should receive message A before B. However, as the bus frequency continuously increases, the bandwidth and electrical limits of centralized bus designs make them inappropriate for current generation systems [14].

To overcome this limitation, some recent bus-based protocols use virtual bus switched interconnects that exploit high-speed point-to-point links [14]. These interconnects provide a total order by ordering all requests at the root switch. For example, the Sun UE10000 [35], which can accommodate up to 64 processors, uses four interleaved address buses to broadcast coherence transactions. This complex address network, designed to perform ordered broadcasts, significantly increases the final cost of the system. Moreover, the energy consumed by snoop requests and snoop bandwidth limitations make snoopy-based designs extremely challenging.

In contrast, directory-based protocols can take advantage of scalable point-to-point interconnection networks, such as a mesh or a torus [49]. These systems maintain a directory at the home node that resolves possibly conflicting requests by ordering requests on a per-cache-block basis. Unfortunately, traditional directory protocols must first send all requests to the home node, adding latency to the critical path of cache-to-cache misses.

1.1.3 Low Power Consumption

Power has always been the dominant design issue in the world of handheld and portable devices. Battery life and system cost constraints drive designers to consider power over performance in such systems. Power consumption has recently emerged as the major constraint in the design of microprocessors [57, 115, 27]. Chip designers now consider power consumption and dissipation limits early in the design cycle and evaluate the tradeoff between power and performance.

Although one obvious metric to characterize the power-performance efficiency of a microprocessor is MIPS (million instructions per second)/watts (W), there are strong arguments against it. Performance has typically been the primary driver of high end processors, so a design team may well choose a design point that has high performance level but low MIPS/W efficiency (within power budget). To solve this problem, a fused metric energy-delay product is often used. The extra delay factor ensures a greater emphasis on performance. For the highest performance server-class machines, it may be appropriate to weight the delay part even more, which would point to the use of energy-delay^2.

Since multiprocessors are at the high end of the performance spectrum, we use energy-delay^2 metric to evaluate the power-performance efficiency of the cache coherence protocols. This metric limits the effectiveness of some common cache coherence protocol optimizations. For example, prefetching is one technique used to reduce the number of coherence misses. Some researchers propose a two-level predictor to predict the coherence messages and prefetch the data, thus potentially eliminating all coherence overhead [80]. However, aggressive prefetching often generates excessive coherence traffic and incurs extra power consumption. Therefore, aggressive prefetching might reduce the power-performance efficiency of the protocol.

1.1.4 Simple Programming Model

The memory consistency model assumed throughout this dissertation is *sequential consistency* [82]. Sequential consistency is the simplest programming interface

for shared memory systems that impose program order among all memory operations. Informally, sequential consistency requires that when processes run concurrently on different machines, any valid interleaving of read and write operations is acceptable, as long as all processes see the same interleaving of operations. In this model, multiple processes appear to share a single logical memory, even though it may be physically distributed. Every process appears to issue and complete memory operations one at a time and atomically in program order. One advantage of sequential consistency is its portability. Any parallel program developed on a uniprocessor is guaranteed to generate the same result when run on a DSM system conforming to sequential consistency.

However, not all parallel programs are written assuming such a strict memory model. Many programs use synchronization primitives to guarantee that different threads cannot perform conflicting accesses to shared variables. For these programs, it is sufficient to enforce data consistency only at the synchronization points. Several weaker memory models [7, 53, 50] have been developed to provide an efficient DSM implementation to run these programs. For example, release consistency requires the programmer to label the program with release and acquire points, where the writes done by a thread must be visible to other threads. Since release consistency requires only some memory operations to be performed in program order, it often improves performance over sequentially consistent implementations. However, Gniady and Falsafi argue that sequential consistency implementations can perform as well as release consistency implementations if the hardware provides enough support for speculations [55]. We assume sequential consistency in our study, but the mechanisms proposed can be easily applied to other consistency models.

1.1.5 Low Bandwidth Usage

Bandwidth efficiency, also known as message efficiency, measures cache coherence protocols by the global traffic that they generate. Excessive messages can cause interconnect contention which reduces system performance. Existing protocols vary

widely in how they use bandwidth. SCI cache coherence protocols [69] have explicit features to reduce contention and hot-spotting in the memory system. Remote access cache [85] and S-COMA [107] cache remote data to reduce global traffic. In contrast, some protocols sacrifice bandwidth efficiency (e.g., using coarse sharing vectors) to reduce the overhead of memory directory [88].

Bandwidth efficiency is perhaps currently a less important attribute. Most systems shipped recently have a small to moderate number of processors, where bandwidth is not the system bottleneck. For example, of the 30,000 SGI Origin 200/2000 [83] systems shipped, less than 10 systems contained 256 or more processors (0.03%), and less than 250 systems had 128 or more processors (1%) [98]. Therefore, a cache coherence protocol should conserve bandwidth as much as possible, while not sacrificing other attributes to achieve this.

1.2 Context of Cache Coherence Protocols

As discussed in the previous section, the tradeoffs in coherence protocol design are only partially understood and change as technology advances. In this section, we start discussing the contexts in which the cache coherence protocols operate and demonstrate how to exhibit the desirable attributes of coherence protocols within a given context.

Computer systems are organized as a set of layers, as indicated in Figure 1.1. The concept of layers is important as it relieves users of the need to worry about technical details. After some constraints are enforced across these layers, development teams can work on different layers of a system independently. However, this transparency also brings inefficiency in the implementation, as designers are required to make assumptions based on worst-case scenarios. A major contribution of this dissertation is the observation that **context knowledge** can be used to optimize the cache coherence protocols. Context means situational information. Or as Abowd et al. state: "Context is any information that can be used to characterize the situation of an entity." [5] We perform two case studies that demonstrate

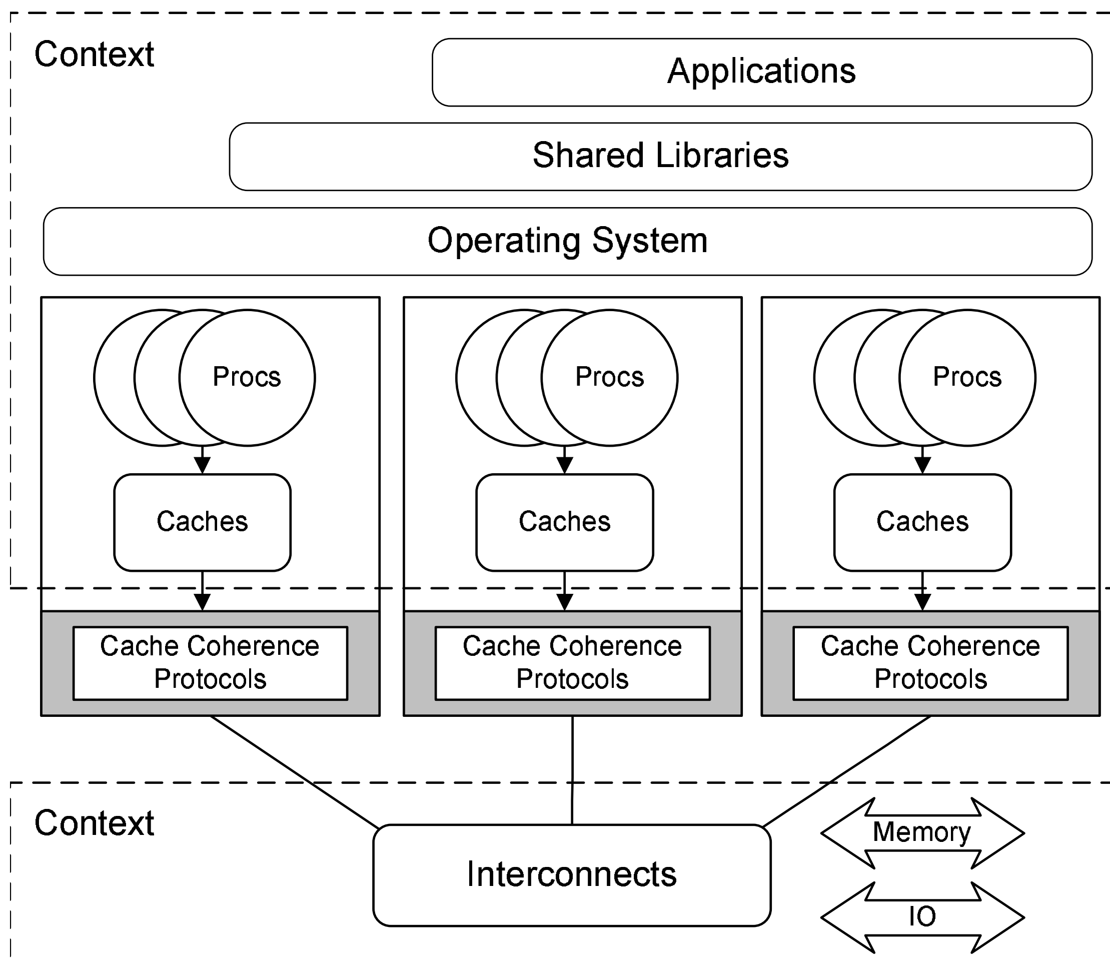

Figure 1.1. Context knowledge for cache coherence protocols.

context knowledge exposed to the cache coherence protocols can lead to protocol optimizations.

The first context we studied is the sharing patterns that can be observed and detected in the memory reference streams. Memory reference streams are generated by a mixture of application and operating system code. Many scientific applications show stable sharing patterns, which can be used to optimize cache coherence protocols. Producer-consumer sharing is one of most popular sharing patterns. In a conventional directory-based write-invalidate protocol, unless the producer is located on the home node, the producer requires at least three network latencies whenever it wishes to modify data: (1) a message from the producer to

the home node requesting exclusive access, (2) a message from the home node to the consumer(s) requesting that they invalidate their copy of the data, and (3) acknowledgments from the home node and consumers to the producer. Figure 1.2 illustrates this scenario. To address the indirection in 3-hop misses, we propose a novel *directory delegation* mechanism whereby the "home directory" of a particular cache line can be delegated to another node. During the period in which the directory ownership is delegated, the home directory forwards all requests for the cache line to the delegated home node. Other nodes that learn of the delegation can send requests directly to the delegated node, bypassing the home directory as long as the delegation persists. As the home directory accesses are removed from the critical path, 3-hop misses are converted to 2-hop misses. To further improve performance, we extend the delegation mechanism to enable a node (delegated home node) to speculatively forward newly written data to the nodes that it believes are likely to consume it in the near future. We employ remote access caches (RACs) to provide a destination for these speculative pushes. This mechanism is somewhat analogous to last write prediction [81] or could also be thought of as producer-driven pre-pushing, in contrast to the common idea of consumer-driven pre-fetching. When

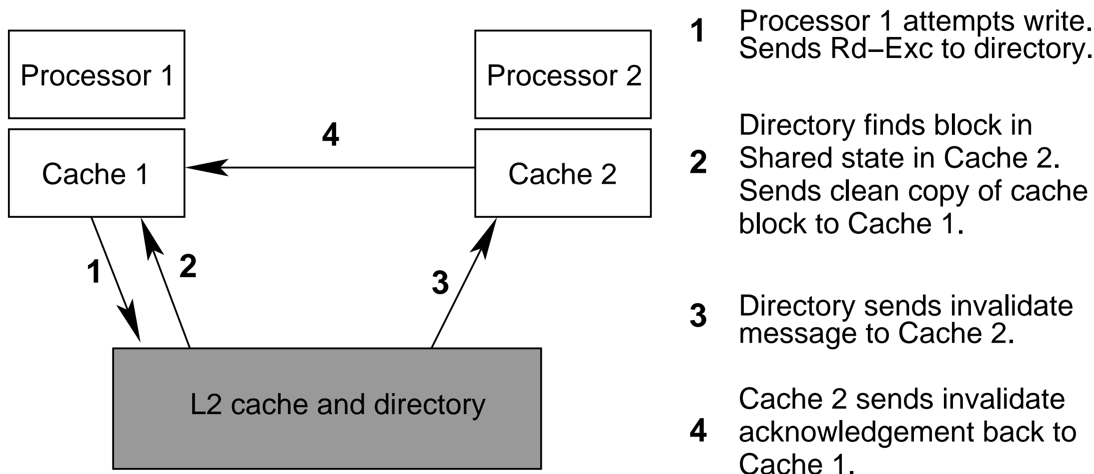

Figure 1.2. An example of 3-hop miss

the producer correctly predicts when and where to send updates, 2-hop misses become local misses, effectively eliminating the impact of remote memory latency.

Although scientific benchmarks show regular sharing patterns that can benefit from the use of adaptive coherence protocols, commercial benchmarks display less temporal and spatial locality, which require extensive storage [80, 92, 110] to track memory access histories. We study the varying latency, bandwidth needs of different coherence messages, and propose a heterogeneous interconnect to optimize the protocols. In the past, all protocol messages are indiscriminately sent on a homogeneous interconnect. As VLSI techniques enable a variety of wire implementations, architects might employ heterogeneous interconnects to reduce the performance and power overhead of directory-based protocols [39]. We propose a heterogeneous interconnect comprised of wires with varying latency, bandwidth, and energy characteristics, and advocate intelligently mapping coherence operations to the appropriate wires. For example, wires with varying latency and bandwidth properties can be designed by tuning wire width and spacing. Similarly, wires with varying latency and energy properties can be designed by tuning repeater size and spacing. In the same 3-hop misses transaction in Figure 1.2, the requesting processor may have to wait for data from the home directory (a 2-hop transaction) and acknowledgements from other sharers of the block (a 3-hop transaction). As the acknowledgements are on the critical path and have low bandwidth requirement, they can be mapped on wires optimized for delay, while the data transfer is not on the critical path and can be mapped on wires optimized for low power.

1.3 Thesis Statement

Cache coherence protocols that are tuned for the contexts in which they are operating can significantly increase performance and reduce power consumption. In order to prove this thesis statement, we perform two case studies that demonstrate context knowledge exposed to the cache coherence protocols can lead to protocol optimizations.

Being aware of the directory access patterns in the applications, our protocols can dynamically delegate the "home directory" to the appropriate node to remove directory access from the critical path. Being aware of the data access patterns in the applications, our protocols can enable the delegated home node to speculatively forward newly written data to the nodes that are likely to consume it in the near future, thus eliminating remote misses. In this context, our sharing pattern-aware protocols can improve performance (by reducing the number of remote coherence misses) and improve bandwidth efficiency, while maintaining the simple programming model.

The protocols we propose can also exploit a heterogeneous interconnect, which is composed of wires with varying latency, bandwidth, and energy characteristics. Being aware of such a heterogeneous interconnect, our proposed protocols can send critical messages on fast wires and noncritical messages on power efficient wires. In this context, our interconnect-aware protocols can improve performance and reduce power consumption, while retaining other desirable attributes.

1.4 Contributions

In this dissertation, we motivate, present, and evaluate the idea of context-aware coherence protocols. The primary contributions of this dissertation are:

- Identify and quantify the performance inefficiency of producer-consumer sharing patterns on a conventional directory-based write-invalidate protocol. And design a simple hardware mechanism to detect producer-consumer sharing by using only information available to the directory controller.

- Propose a directory delegation mechanism whereby the home node of a cache line can be delegated to a producer node, thereby converting 3-hop coherence operations into 2-hop operations. Then extend the delegation mechanism to support speculative updates for data accessed in a producer-consumer pattern, which can convert 2-hop misses into local misses, thereby eliminating the remote memory latency. An important feature of the mechanisms proposed

is that they require no changes to the processor core or system bus interface and can be implemented entirely within an external directory controller. In addition, our optimizations are sequentially consistent.

- Identify the hop imbalance in coherence transactions. For example, when a processor wants to write in a directory-based protocol, it may have to wait for data from the home node (a 2-hop transaction) and for acknowledgments from other sharers of the block (a 3-hop transaction). Since the acknowledgments are on the critical path and have low bandwidth needs, they can be mapped on latency-optimized wires to improve performance. Similarly, the data block transfer is not on the critical path and can be mapped on power-optimized wires to reduce power consumption.

- Present a comprehensive list of techniques that allow coherence protocols to exploit a heterogeneous interconnect and evaluate the techniques to show their performance and power-efficiency potential.

1.5 Outline

This dissertation is organized into five chapters. The next chapter discusses the fundamentals of cache coherence protocols, and how these relate to context-aware protocols. Chapter 3 presents an adaptive cache coherence protocol that identifies and exploits producer-consumer sharing patterns in many applications by using *directory delegation* and *speculative updates*. Chapter 4 presents interconnect-aware protocols that significantly improve performance and reduce power consumption by intelligently mapping coherence traffic to the appropriate types of wires. Finally, Chapter 5 summarizes this dissertation and points out a few possible directions for future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter provides the background and terminology for understanding the cache coherence protocols described throughout this dissertation. The focus of this chapter is on recent researches, though it also covers some basics of cache coherence. We refer readers to Chapter 6 of Hennessy and Patterson [63] and Chapters 5-8 of Culler and Singh [46] on this topic for further background and introductory material.

We first discuss the problem of keeping the contents of caches coherent in shared memory multiprocessors. In Section 2.2, we describe the basic strategy and coherence states used in invalidation based cache coherence protocols. Section 2.3 describes four major types of cache coherence protocols: bus-based protocols, ring-based protocols, token-based protocols and directory-based protocols. Finally, in Section 2.4, we demonstrate the advantages of context-aware cache coherence protocols and present prior work on context-aware coherence protocols.

2.1 Caching in Multiprocessor Systems

Advances in semiconductor technology have led to a tremendous increase in clock speeds and transistor counts in single-chip devices. Higher clock speeds along with novel architectural techniques have improved microprocessor performance by orders of magnitude in the last decade [28]. Despite the dramatic improvements in a single processor's performance, supercomputers and enterprise servers continue to demand higher performance to solve large scientific and commercial problems [58].

To go beyond the performance offered by a single microprocessor, computer designers have studied techniques that connect multiple processors in some way so that they can cooperate on common computational tasks. As defined by Almansi

and Gottlieb [12], a multiprocessor is a computing system composed of multiple processors and memory modules connected together through an interconnection network.

A variety of programming abstractions and languages have been developed to program multiprocessors. Message passing and shared memory are the two predominant communication models for parallel multiprocessor architectures [116]. In the traditional message passing model, processors communicate by directly sending and receiving messages to one another. This paradigm allows programmers to optimize data movement among different nodes based on the characteristics of applications. However, the message passing model places all of the burden of data decomposition, synchronization, data motion on the programmers and compilers. Despite substantial effort, current compiler technology has not yet been able to automatically extract and exploit parallelism via the insertion of message passing calls on a wide range of programs [42]. This forces programmers to explicitly specify all communication required among the nodes in the system. Such specification is hard, if not impossible, for applications that exhibit dynamic communication behavior or fine grain sharing.

In contrast, many programmers find the shared memory model attractive. This model provides programmers with a simple abstraction. A programmer's intuition developed on single-processor systems still holds on shared memory multiprocessors, as a load of any memory location still returns the latest value written in that location. The underlying memory system will satisfy a processor's request to read or write a memory location by sending and receiving messages to the appropriate memory module on the processor's behalf. This abstraction makes it possible to run sequential programs directly on shared memory systems.

Like in uniprocessor systems, effective caching of memory is the key to reducing memory latency in shared memory multiprocessor systems. In fact, caching offers even more benefits in multiprocessors than in uniprocessors due to the higher memory latencies [106]. However, multiprocessor caches are more complicated than uniprocessor caching because they introduce coherence problems between the

different processor caches. When multiple copies of the same data are present in different caches at the same time, a processor wishing to modify the data needs to take some actions to notify other processors about the changes and prevent them from reading a stale copy of the data. Some shared memory multiprocessors avoid this cache coherence problem by restricting the kind of the addresses can be cached. For example, the Cray T3E multiprocessor [13] simplifies the coherence problem by only allowing local memory to be cached. Similarly, the IBM Cell processor [65] avoids coherence issues between the Synergistic Processor Elements (SPEs) by treating each SPEs' local memories as register files. These restrictions usually result in lower processor utilization and/or force programmers to be much more aware of locality and interprocessor communication.

Two properties are required to maintain coherence. First, changes to a data location must be made visible globally, which is called *write propagation*. Second, the changes to a location must be made visible in the same order to all processors, which is called *write serialization*. Depending on how other processor caches are notified of the changes, protocols can be classified as invalidation-based and update-based as shown in Figure 2.1. In Figure 2.1(a), only the memory has a valid copy of data block A. In Figure 2.1(b), both processors read A and store it in their private caches. The difference between the invalidation-based and update-based protocols becomes apparent when processor P1 issues a write. In an invalidation-based protocol, processor P1 must first obtain exclusive ownership of the cache line containing A by invalidating all remote copies before modifying its local copy of the data, as shown in Figure 2.1(c). The advantage of this is that only the first write, in a sequence of writes to the same cache line with no intervening read operations from other nodes, causes global actions. Figure 2.1(d) demonstrates the same scenario when a update-based protocol is employed. The processor writes to its copy of the cache block and propagates the change to remote replicas at the same time. Upon receiving an update massage, the remote caches update their contents accordingly. Although update-based protocols reduce the number of coherence misses by eliminating both true and false sharing misses, these protocols

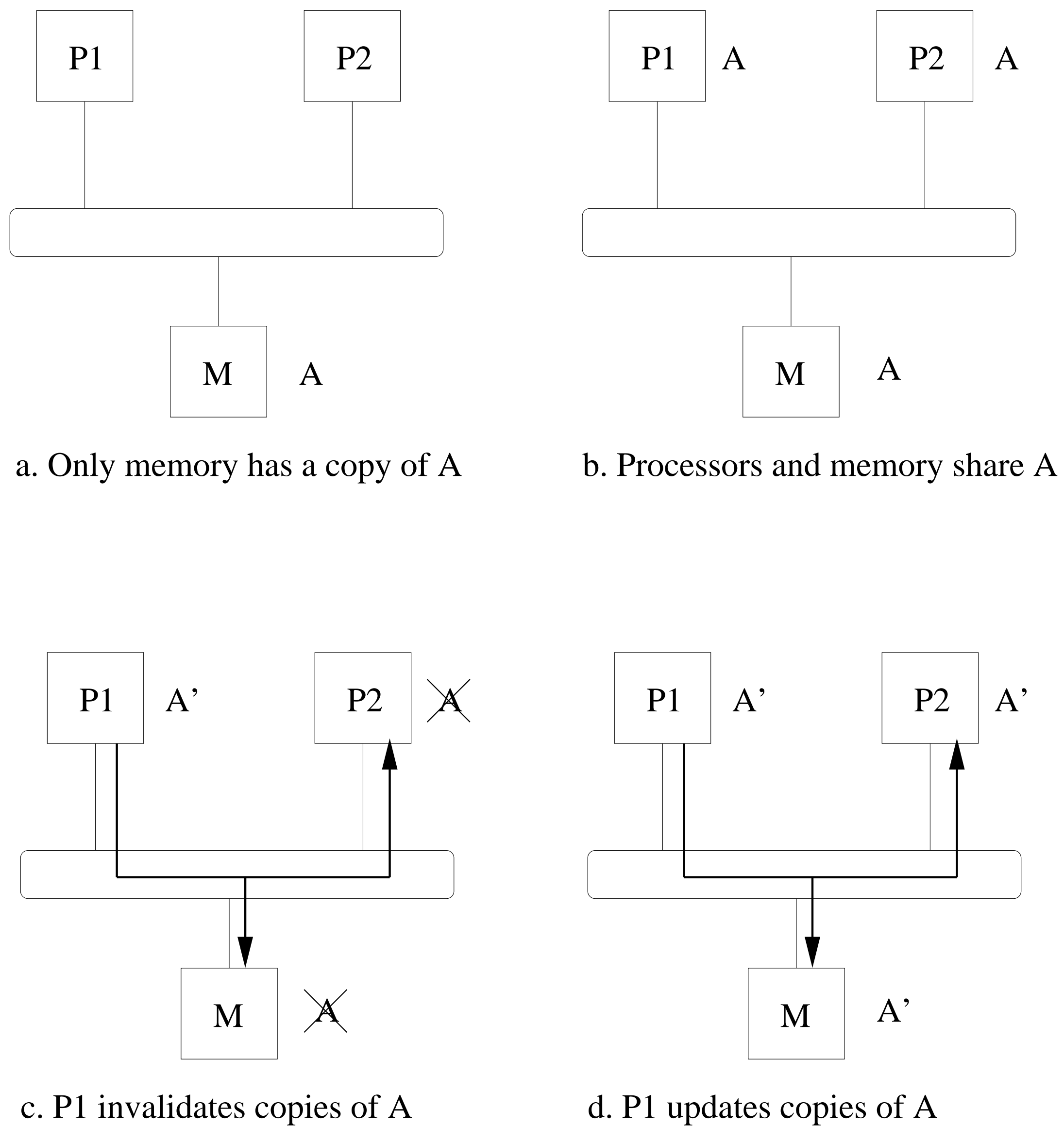

Figure 2.1. Write invalidate and write update

tend to generate an excessive number of update messages, which leads to serious performance problems by consuming precious interconnect bandwidth.

The comparison between update-based protocols and invalidation-based protocols was a subject of considerable debate in the past [59]. The trend towards larger cache line sizes, the increased bandwidth requirement of updated-based protocols, and the complexity of maintaining sequentially consistent updates, tilted

the balance in favor of invalidation-based protocols. Since all recent commercial shared memory multiprocessors [14, 35, 37, 76, 83, 102, 114] employ invalidation-based protocols, we consider invalidation-based protocols as the baseline in this dissertation. However, in Section 3.2.4, we describe how selectively sending update messages can improve the performance of invalidation-based protocols.

2.2 Common Coherence States

Coherence protocols use protocol states to track read and write permissions of blocks present in processor caches. We first describe the well-established MESI [14, 114, 83, 102] and MOESI states [37, 113]. Next, we discuss various mechanisms used for optimizing particular sharing patterns.

2.2.1 MESI Protocol

We first consider the MESI protocol. In the MESI protocol, a cache line is in one of four states: M(modified), E(exclusive), S(shared), or I(invalidate). A block in the INVALID or I state is not valid, and data must be fetched to satisfy any attempted access. When a block is not found in a cache, it is implicitly in the INVALID state with respect to that cache. The SHARED or S state signifies that this line is one of several copies in the system. It also implies no other replicas is in E or M states. A processor may read the block, but cannot write it. A block in the EXCLUSIVE or E state indicates this cache has the only copy of the line, but the line is clean. A processor may read, write and silently drop this line. A block in the MODIFIED or M state indicates this cache has the only valid copy of the cache line, and has made changes to that copy. A processor can read and write this line, but must write back the dirty data to main memory when replacing this line from the cache. Table 2.1 shows the basic operations of a processor in an abstract MESI coherence protocol. In this table, we refer to writeback operations due to cache line eviction as explicit writebacks (EWB), and writeback operations due to remote snoops as implicit writebacks (IWB).

Table 2.1. MESI state transitions

	PROCESSOR ACTIONS			REMOTE SNOOPS	
STATE	READ	WRITE	EVICTION	READ	WRITE
MODI	HIT	HIT	EWB → INV	SEND DATA IWB → SHRD	SEND DATA IWB → INV
EXCL	HIT	HIT	SILENT DROP → INV	ACK → SHRD	ACK → INV
SHRD	HIT	WRITE REQ	SILENT DROP → INV	ACK	ACK → INV
INV	READ REQ → SHRD	WRITE REQ → EXCL	NONE	NONE	NONE

2.2.2 MOESI Protocols

Sweazey and Smith [113] introduce another state OWNED or O to the traditional MESI protocols. Adding an O state allows read-only access to a block whose value is dirty. The block in the OWNER state is responsible for updating main memory before evicting a block. It is similar to MODIFIED state in that only a single block is allowed to be in the OWNED state at one time. However, blocks in other processors are allowed to be in the SHARED state at the same time when one block is in the OWNED state.

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 remote snoop request. In the traditional MESI protocol, the responder needs to send a response message to the requester and an implicit writeback message to update the memory. With the addition of the OWNED state, the processor can transition from MODIFIED to OWNED without writing back the changes to the main memory immediately. 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 systems where a cache-to-cache transfer is faster than the memory access, the processor with the block in the OWNED state can be given the responsibility of providing data.

Figure 2.2 discusses the relationship between the five states, as described by Sweazey [113]. Each cache line can be in one of the five states and specified with three characteristics: ownership, validity, and exclusiveness. All lines in the MODIFIED, EXCLUSIVE, SHARED, and OWNED states are considered “valid.” Lines in the MODIFIED and EXCLUSIVE states are considered “exclusive.” Lines in the EXCLUSIVE and OWNED state are regarded as having “ownership.” Main memory holds the ownership for all cache lines that are not owned by any cache. The exclusive owner of the cache line is responsible for keeping the data coherent, as well as serving future requests. For example, when a line in the EXCLUSIVE state transitions to the MODIFIED state, the ownership of the line is transferred from the main memory to this cache. Later, when this line downgrades from MODIFIED to OWNED, the cache no longer exclusively owns this cache line. Hence, other caches can have this line in the SHARED state.

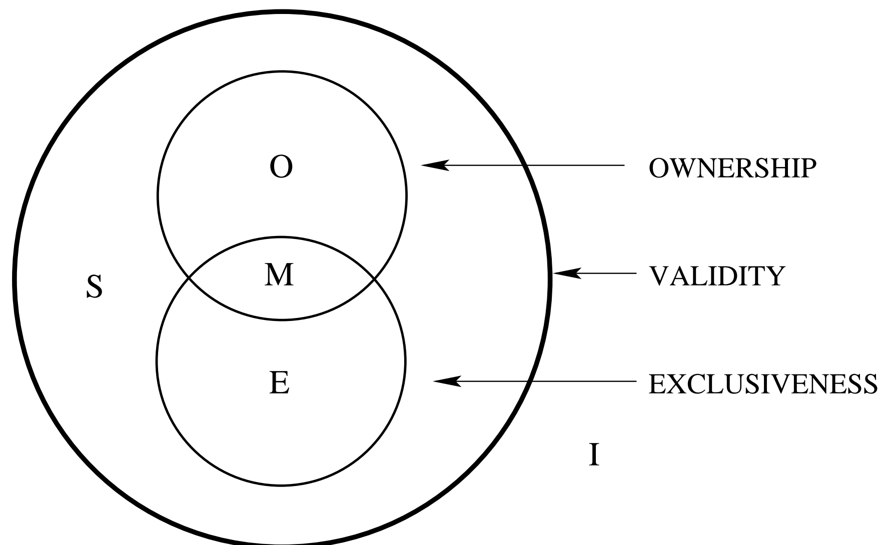

Figure 2.2. Three characteristics of cached data.

2.3 Common Coherence Protocols

Coherence protocols encode some permissions and other attributes of blocks in caches using a subset of the MODIFIED, OWNED, EXCLUSIVE, SHARED, and INVALID coherence states. A designer needs to choose one of several approaches for manipulating and controlling these states. Modern shared memory systems employ bus-based protocols [33, 35, 37], ring-based protocols [22, 114], token-based protocols [95, 97] or directory-based protocols [26, 54, 72, 83] based on the types of interconnection networks.

2.3.1 Bus-based Protocols

Many small-scale shared memory multiprocessors use bus-based protocols. In these systems, a shared bus is employed to interconnect all processors and memory modules [33]. Each processor also maintains a duplicate set of snoop tags, which are dedicated to monitoring snoop requests from the bus. These snoop tags filter memory traffic that does not affect the local caches, thus reducing contention for cache tags.

To increase effective system bandwidth, many evolutionary enhancements have been proposed to snoop protocol designs. Split transaction protocols [35, 37] allow the requesting processor to release the bus while waiting for its response. This mechanism significantly increases bus efficiency by pipelining multiple requests. Other systems implement virtual buses [36, 109] using distributed arbitration, point-to-point links, and dedicated switch chips. However, virtual-bus systems still rely on totally-ordered broadcasts for issuing requests. As bus frequency increases, the bandwidth and electrical limits of the centralized bus make it inappropriate for current generation systems [14].

The primary advantage of snoop-based multiprocessors is the low cache-to-cache miss latency. Since a request is sent directly to all other processors and memory modules in the system, the current owner quickly knows that it should send a response. If cache-to-cache misses have lower latency than memory access, replying with data from the owning processor cache whenever possible can reduce the average

miss latency. Bus-based protocols are relatively simple, although this advantage is much less pronounced as the advent of split transactions and virtual buses. As we have mentioned before, the major disadvantage of bus-based protocols is scalability. First, the bus is a mutually exclusive resource and only one processor can transmit at any given time. Second, all processors must participate in an arbitration phase before accessing the bus. Third, the bus clock cycle must be long enough so that signals can propagate throughout the entire bus. Moreover, in broadcast-based protocols, the bandwidth requirement increases with the number of processors, which becomes prohibitive in the large systems.

2.3.2 Ring-based Protocols

Point-to-point unidirectional connections have emerged as a very promising interconnection technology. Point-to-point connections have only one transmitter and one receiver and their transmission speed is much higher than buses [14]. Additionally, signals on point to point links can be pipelined: a new transmission can be started before the previous one has reached the receiver. Therefore, point-to-point connections are much more technologically scalable than bus connections, and their delivered bandwidth is expected to benefit continuously from improvements in circuit technology.

An unidirectional ring is one of the simplest forms of point-to-point interconnection. In this design, coherence transactions are serialized by sending snoop messages along the ring. Barroso and Dubois [22] evaluate the performance of a logically-embedded slotted ring. The Hector multiprocessor [117], developed at the University of Toronto, uses a hierarchy of unidirectional slotted rings. Alternatively, IBM's Power4 [114] can embed a logical ring in whatever network topology is available. In this design, snoop messages are ordered on the logical ring, while other messages can use any link in the network. Ring-based interconnect is simple and low cost. Moreover, it places no constraints on the network topology or timing.

One main drawback of this approach is that snoop requests may suffer long latencies or entail many snoop messages and operations. For example, a scheme

where each node snoops the request before forwarding it to the next node in the ring induces long request latencies. Likewise, a scheme where each node immediately forwards the request and then performs the snoop is likely to induce many snoop messages and snoop operations. To solve this problem, Strauss et al. [112] propose flexible snooping algorithms, a family of adaptive forwarding and filtering snooping algorithms. In these algorithms, depending on certain conditions, a node receiving a snoop request may either forward it to another node and then perform the snoop, or snoop and then forward it, or simply forward it without snooping. However, despite of all these optimizations, ring-based protocols are not highly scalable, so they are most likely to be employed in small systems with less than eight nodes.

2.3.3 Token-based Protocols

Martin et al. [94, 95] observe that simple token counting rules can ensure that a 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.

Token coherence can simultaneously capture the best aspects of bus-based protocols and directory-based protocols. These two dominant approaches to coherence have a different set of attractive attributes. Bus-based 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. The correctness substrate of token coherence provides a foundation for implementing many performance policies. These performance policies focus on making the system fast and bandwidth-efficient without any correctness responsibilities.

Token coherence prevents starvation using persistent requests. 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. While conceptually appealing, this scheme has some potentially difficult implementation issues. One of them is that persistent requests need an arbiter to avoid live-lock. A simplified implementation uses a single centralized arbiter. The substrate directs persistent requests to the arbiter, queuing multiple requests in a dedicated virtual network or at the arbiter itself. However, a single centralized arbiter is likely to be the bottleneck in the large systems. Martin presents a distributed-arbitration technique that reduces the latency of persistent requests by allowing direct communication of persistent requests between processors [94], though this technique still relies on point-to-point ordering in the interconnect to avoid reordering activation and deactivation messages.

Another drawback of token-based protocols is that every line needs token storage in main memory. Finally, in multiprocessors with multiple CMPs, the protocol must be extended with additional storage and states to allow a local cache in the CMP to supply data to another local cache. Some of these issues are addressed in [97].

2.3.4 Directory-based Protocols

Directory-based protocols target the systems where scalability is a first-order design constraint. Examples of systems that use directory-based protocols include Stanford's DASH [85] and FLASH [62], MIT's Alewife [77], SGI's Origin [83], HP's AlphaServer GS320 [54], Sequent's NUMA-Q [89], and Cray's X1 [45]. An emerging trend in processor microarchitecture design is a move towards chip multiprocessors (CMPs). Almost all new CMP systems also employ directory-based protocols, such as IBM's Power4 [114], Sun's Niagara [74], HP's Superdome [56], Intel's E8870 [26] and AMD's Opteron [72]. Since directory-based protocols are employed in most modern systems, the focus of this dissertation is on improving directory-based protocols.

Scalable shared memory multiprocessors are constructed based on scalable point-to-point interconnection networks, such as fat trees or meshes [49]. Main memory is physically distributed in order to ensure that memory bandwidth scales with the number of processors. In these designs, totally ordered message delivery becomes unfeasible. Since the state of a block in the caches can no longer be determined implicitly by placing a request on a shared bus, a directory-based protocol associates each cache line of data with an entry in a directory that tracks the global coherence state of the cache line. Upon receiving a request, the home directory uses the directory information to respond directly with the data and/or forward the request to other processors. In typical protocols, the directory uses DirN, also known as Full-Map, to exactly identify the sharers. This amount of information is proportional to the product of the number of memory blocks and the number of processors. For larger multiprocessors, more efficient methods are used to scale the directory structure. Researchers have proposed to encode a superset of sharers [8, 61, 88] or to form a linked list of sharers [69].

The directory also provides a sequential point for each block which handles conflicting requests and eliminates various protocol races. Since directory-based protocols avoid the use of centralized buses, multiple messages for the same block can be active in the system at the same time. The order between these messages is determined by the order in which the requests are processed by the directory. Many directory-based protocols use transient or pending states to delay subsequent requests to the same block by queuing or negatively acknowledging requests at the directory controller while a previous request for the same block is still active in the system. A simple directory protocol might employ only one busy state and enter the busy state anytime a request reaches the directory. A more optimized protocol often employs many busy states and allows transitions between different busy states, although it also brings tremendous challenge in proving the correctness properties of the protocol.

Directory-based protocols can exploit arbitrary point-to-point interconnects and dramatically improve scalability. Since snoop messages are only sent to processors

that might have copies of a cache block, the traffic in the system grows linearly with the number of processors instead of quadratically as in bus-based protocols. However, directory-based protocols entail directory access, which is on the critical path of cache-to-cache misses. With the dominance of cache-to-cache misses in many important commercial workloads, these high-latency cache-to-cache misses can significantly impact system performance.

2.4 Related Work

We argue that future cache coherence protocols need to be aware of the context in which they are operating. Hence, all coherence traffic can be tuned based on the hints given by the context. Compared with traditional protocols, context-aware coherence protocols can significantly improve performance and reduce power consumption through specific optimizations.

As defined in Section 1.2, context means situational information. We focus on two kinds of context in this dissertation. The first one is the sharing patterns displayed in the memory reference streams. We describe the protocol which can automatically detect and optimize the producer-consumer sharing patterns in Chapter 3. The second context is the varying latency, bandwidth needs of different coherence messages. We propose a heterogeneous interconnect consisting of wires with different characteristics, and describe our interconnect-aware cache coherence protocols in Chapter 4. In this section, we provide a survey of the ideas that are most closely related to our context-aware protocols, and discuss how other proposals can benefit from being aware of their operating context.

2.4.1 Sharing Pattern-aware Protocols Related

Our sharing pattern-aware cache coherence protocols are similar in nature to those adaptive cache coherence protocols. Adaptive cache coherence protocols that optimize various sharing patterns at runtime have been proposed for migratory data [44, 111], pairwise sharing [68], and wide sharing [71].

Migratory sharing patterns are common in many multiprocessor workloads. They result from data blocks that are read and written by many processors in turn [60]. Shared data guarded by lock-based synchronization tend to exhibit migratory sharing patterns. In traditional MOESI coherence protocols, these read-then-write sequences generate a read miss followed by an upgrade miss. An optimization for the migratory sharing allows a processor to respond an external read request with a read-exclusive response and transitioning to INVALID. When the requesting processor receives the read-exclusive response, it transitions immediately to the EXCLUSIVE state to avoid a potential upgrade miss. This policy performs well in workloads dominated by read-then-write patterns. However, it substantially penalizes other sharing patterns, like widely shared data [71]. To find a balance, most cache coherence protocols in commercial systems [26, 1] employ a less aggressive optimization for the migratory sharing, as illustrated in Table 2.2. Upon receiving a READ request, the home directory performs the optimization only when the requested line is in the INVALID state. In this case, the home directory will grant the requesting processor the exclusive ownership.

Table 2.2. MOESI state transitions

	PROCESSOR ACTIONS			REMOTE SNOOPS	
STATE	READ	WRITE	EVICTION	READ	WRITE
MODI	HIT	HIT	EWB → INV	SEND DATA → OWNED	SEND DATA IWB → INV
OWN	HIT	HIT	EWB → INV	SEND DATA	SEND DATA IWB
					→ INV
EXCL	HIT	HIT	SILENT DROP → INV	ACK → SHRD	ACK → INV
SHRD	HIT	WRITE REQ	SILENT DROP → INV	ACK	ACK → INV
INV	READ REQ (SHRD RPLY) → SHRD -OR- READ REQ (EXCL RPLY) → EXCL	WRITE REQ → EXCL	NONE	NONE	NONE

Prefetching is often used in adaptive cache coherence protocols to hide long miss latencies [30, 105], though aggressive use of prefetch tends to increase network traffic and pollute the local cache. Recent research has focused on developing producer-initiated communication mechanisms in which data are sent directly to the consumer's cache [4, 75, 119] under weak/released consistency modes.

A different approach is to use speculative coherence operations. Prediction in the context of shared memory was first proposed by Mukherjee and Hill, who show that it is possible to use address-based two-level predictors at the caches and directories to trace and predict coherence messages [103]. Subsequently, Lai and Falsafi improve upon these predictors by using smaller history tables and show how coalescing messages from different nodes can accelerate reads [80]. Alternatively, Kaxiras and Goodman propose instruction-based prediction for migratory sharing, wide sharing, and producer-consumer sharing [71].

Many research have proposed exploiting coherence prediction to convert 3-hop misses into 2-hop misses. Acacio et al. [6] and Martin et al. [92] propose mechanisms for predicting sharers. Lebeck and Wood propose dynamic self-invalidation [84], whereby processors speculatively flush blocks based on access history to reducing the latency of invalidations by subsequent writers. Lai and Falsafi propose a two-level adaptive predictor to more accurately predict when a cache line should be self-invalidated [81]. All of these techniques do a good job of predicting when a processor is done with a particular cache line, but they are all integrated on the processor die. The predictor used in our sharing pattern-aware protocols is simpler, and likely less accurate. It can, however, be used in conjunction with unmodified commercial processors, since all changes are made at the external directory controller. As a result, our design can be adopted in near-term system designs such as the SGI Altix.

2.4.2 Interconnect-aware Protocols Related

Most scientific benchmarks show regular sharing patterns that can benefit from the use of adaptive coherence protocols. However, commercial benchmarks display

less temporal and spatial locality, which require extensive storage [80, 92] to track memory access histories. We propose a heterogeneous interconnect comprised of wires with varying latency, bandwidth, and energy characteristics, and design interconnect-aware coherence protocols which intelligently map coherence operations to the appropriate wires.

To the best of our knowledge, only three other bodies of work have attempted to exploit different types of interconnects at the microarchitecture level. Beckmann and Wood [23, 24] propose speeding up access to large L2 caches by introducing transmission lines between the cache controller and individual banks. Nelson et al. [104] propose using optical interconnects to reduce intercluster latencies in a clustered architecture where clusters are widely-spaced in an effort to alleviate power density. Citron et al. [43] examine entropy within data being transmitted on wires and identify opportunities for compression. Unlike our proposed technique, they employ a single interconnect to transfer all data.

A recent paper by Balasubramonian et al. [17] introduces the concept of a heterogeneous interconnect and applies it for register communication within a clustered architecture. A subset of load/store addresses are sent on low-latency wires to prefetch data out of the L1D cache, while noncritical register values are transmitted on low-power wires. A heterogeneous interconnect similar to the one proposed by Balasubramonian et al. [17] has been applied to a different problem domain. The nature of cache coherence traffic and the optimizations they enable are very different from that of register traffic within a clustered microarchitecture. We have also improved upon the wire modeling methodology in [17] by modeling the latency and power for all the network components including routers and latches. Our power modeling also takes into account the additional overhead incurred due to the heterogeneous network, such as additional buffers within routers.

Recent studies [66, 81, 84, 95] have proposed several protocol optimizations that can also benefit from being aware of the context in which they are operating. For example, given a heterogeneous interconnect, dynamic self-invalidation scheme proposed by Lebeck et al. [84] can transfer the self-invalidate [81, 84] messages

through power-efficient wires, as these messages are unlikely on the critical path. Similarly, in a system based on token coherence protocols, the low-bandwidth token messages [95] are often on the critical path, and thus can be transferred on latency optimized L-Wires. A recent study by Huh et al. [66] reduces the amount of false sharing by exploiting incoherent data. For cache lines suffering from false sharing, only the sharing states need to be propagated and such messages are a good match for low-bandwidth wires.

CHAPTER 3

SHARING PATTERN-AWARE COHERENCE PROTOCOLS

Producer-consumer sharing is common in scientific applications. For example, The work in successive over relaxation (SOR) is divided across processors who need to communicate with their neighbors between program phases. Consequently, data along the boundary between two processors exhibit stable producer-consume sharing pattern. However, applications with producer-consumer sharing don't perform well on modern shared memory systems due to an excessive number of coherence message exchanges. In this chapter, we demonstrate how a sharing pattern-aware coherence protocol can optimize the producer-consumer sharing. We start with identifying and quantifying the performance inefficiency of producer-consumer sharing on real cc-NUMA systems. In Section 3.2, we propose two mechanisms that improve the performance of the producer-consumer sharing by eliminating remote misses and reducing the amount of communication required to maintain coherence. We first present a simple hardware mechanism for detecting producer-consumer sharing. We then describe a *directory delegation* mechanism whereby the "home node" of a cache line can be delegated to a producer node, thus converting 3-hop coherence operations into 2-hop operations. We then extend the delegation mechanism to support *speculative updates* for data accessed in a producer-consumer pattern, which can convert 2-hop misses into local misses, thereby eliminating the remote memory latency. Both mechanisms can be implemented without changes to the processor. In Section 3.3, we evaluate our directory delegation and speculative update mechanisms on seven benchmark programs that exhibit producer-consumer sharing patterns using a cycle-accurate execution-driven simulator of a future 16-

node SGI multiprocessor. We find that the mechanisms proposed in this paper reduce the average remote miss rate by 40%, reduce network traffic by 15%, and improve performance by 21%. Finally, we summarize this chapter in Section 3.4.

3.1 Producer-consumer Sharing

Most enterprise servers and many of the Top500 supercomputers are shared memory multiprocessors [2]. Remote misses have a significant impact on shared memory performance, and their significance is growing as network hop latency increases as measured in processor cycles [58]. The performance impact of remote memory accesses can be mitigated in a number of ways, including higher performance interconnects, sophisticated processor-level latency hiding techniques, and more effective caching and coherence mechanisms. This chapter focuses on the latter.

Previous research has demonstrated the value of adaptive protocols that identify and optimize for migratory sharing [44, 111]. In this chapter, we present a novel adaptive mechanism that identifies and optimizes for producer-consumer sharing. An important feature of the mechanisms described herein is that they *require no changes to the processor core or system bus interface* and can be implemented entirely within an external directory controller. In addition, *our optimizations are sequentially consistent.* Further, *we identify producer-consumer sharing with a very simple mechanism* that does not require large history tables or similar expensive structures.

Typical cc-NUMA systems maintain coherence using directory-based distributed write-invalidate protocols. Each cache line of data has a “home node” that tracks the global coherence state of the cache line, e.g., which nodes have a cached copy of the data, via a directory structure [85]. Before a processor can modify a cache line of data, it must invalidate all remote copies and be designated as its *owner*. Thus, only the first write in a sequence of writes to a cache line with no intervening read operations from other nodes causes global actions. This behavior leads to good

performance for situations where data is used exclusively by a single processor or mostly read-shared.

However, write invalidate protocols are inefficient for some sharing patterns. For example, consider producer-consumer sharing, where a subset of threads access a shared data item, one of which modifies the data while the others read each modification. Figure 3.1 illustrates the communication induced by producer-consumer sharing involving one producer node and one consumer node, where data is initially cached in read-only mode on each node.

Unless the producer is located on the home node, a conventional directory-based write-invalidate protocol requires at least three network latencies whenever a producer wishes to modify data: (1) a message from the producer to the home node requesting exclusive access, (2) a message from the home node to the consumer(s) requesting that they invalidate their copy of the data, and (3) acknowledgments from the home node and consumers to the producer. Figure 3.1 illustrates this scenario. Similarly, when the first consumer accesses the data after its copy has been invalidated, it incurs a 3-hop miss unless it or the producer is located on the home node: (1) a message from the consumer to the home node requesting a read-only copy of the data, (2) a message from the home node to the producer requesting that the producer downgrade its copy to SHARED mode and write back

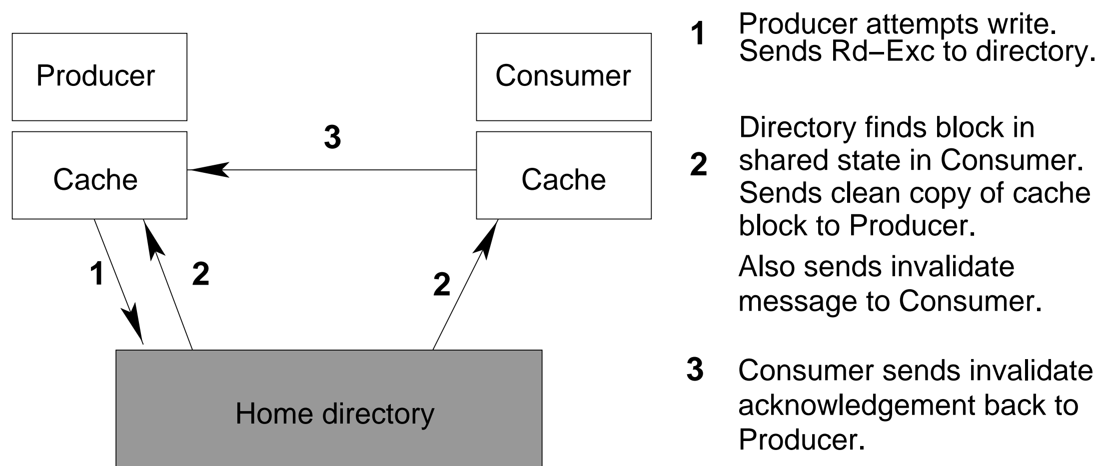

Figure 3.1. Coherence operations at the producer

the new contents of the cache line, and (3) a message from the producer to the consumer providing the new contents of the data and the right to read. Figure 3.2 illustrates this scenario.

One performance problem illustrated here is the third network hop required when the producer does not reside on the home node. To address this problem, we propose a novel *directory delegation* mechanism whereby the "home node" of a particular cache line of data can be delegated to another node. During the period in which the directory ownership is delegated, the home node forwards requests for the cache line to the delegated home node. Other nodes that learn of the delegation can send requests directly to the delegated node, bypassing the home node as long as the delegation persists. When used appropriately, directory delegation can convert a number of 3-hop coherence operations into 2-hop operations.

To improve producer-consumer sharing performance, we extend the delegation mechanism to enable the producer (delegated home node) to speculatively forward newly written data to the nodes that it believes are likely to consume it in the near future. Our update mechanism is a performance optimization on top of a conventional write invalidate protocol, analogous to prefetching or last write prediction [81], and thus does not affect the (sequential) consistency semantics of the underlying coherence protocol. When the producer correctly predicts when and

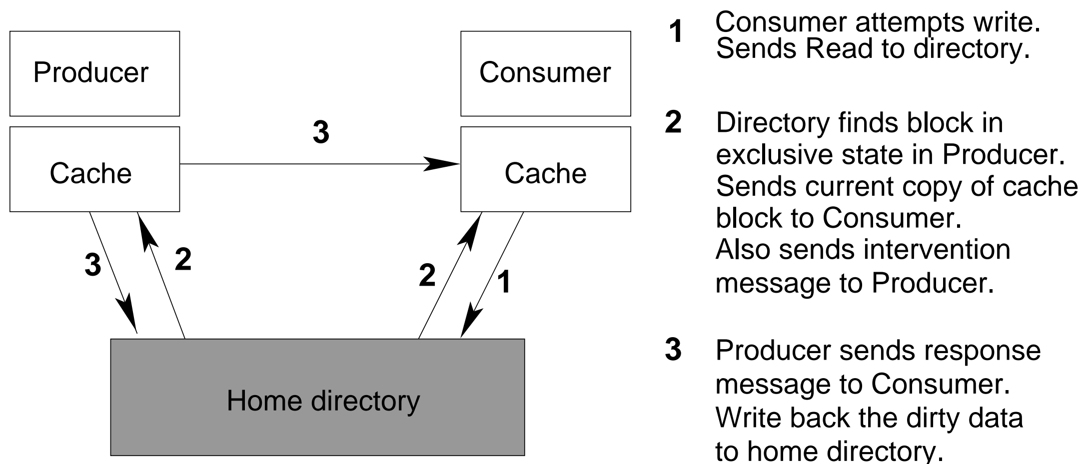

Figure 3.2. Coherence operations at the consumer

where to send updates, 2-hop misses become local misses, effectively eliminating the impact of remote memory latency.

Although update protocols have the potential to eliminate remote misses by pre-pushing data to where it will soon be consumed, they have a tendency to generate excessive amounts of coherence traffic [32, 59]. These extra coherence messages are typically due to sending updates to nodes that no longer are consuming the data. This extra coherence traffic can lead to serious performance problems by consuming precious interconnect bandwidth and memory controller occupancy. To mitigate this problem, we only perform speculative updates when the producer has been delegated control of a cache line and a stable producer-consumer sharing pattern has been observed. For the benchmarks we examine, the speculative push mechanism generates less network traffic than even a tuned write invalidate protocol.

A goal of our work is to design mechanisms that can be implemented in near future systems. We concentrate on mechanisms that require only modest hardware overhead, require no changes to the processor, do not add significant pressure to the interconnect, and do not significantly increase directory controller occupancy. We do not consider designs that require large history tables, assume the ability to "push" data into processor caches, perform updates that are not highly likely to be consumed, or make other assumptions that make it difficult to apply our ideas to commercial systems. In Section 3.4 we discuss ways that our work can be extended by relaxing some of these restrictions.

Using a cycle-accurate execution-driven simulator of a future-generation 16-node SGI multiprocessor, we examine the performance of seven benchmarks that exhibit varying degrees of producer-consumer sharing. We find that our proposed mechanisms eliminate a significant fraction of remote misses (40%) and interconnect traffic (15%), leading to a mean performance improvement of 21%. The reduction in both coherence traffic and remote misses plays a significant role in the observed performance improvements. The hardware overhead required for the optimizations described above is less than the equivalent of 40KB of SRAM per node, including

the area required for the delegate cache, remote access cache, and directory cache extensions used to detect producer-consumer sharing.

3.2 Protocol Implementation

In this section we describe the design of our various novel coherence mechanisms (producer-consumer sharing predictor, directory delegation mechanism, and speculative update mechanism). Figure 3.3 shows our modeled node architecture with new components highlighted. We begin with a brief discussion of remote access caches and their value in our work (Section 3.2.1). We then describe the simple hardware mechanism that we employ to identify producer-consumer sharing patterns. Then, in Section 3.2.3 we describe the protocol extensions and hardware needed to support directory delegation. In Section 3.2.4 we describe the protocol

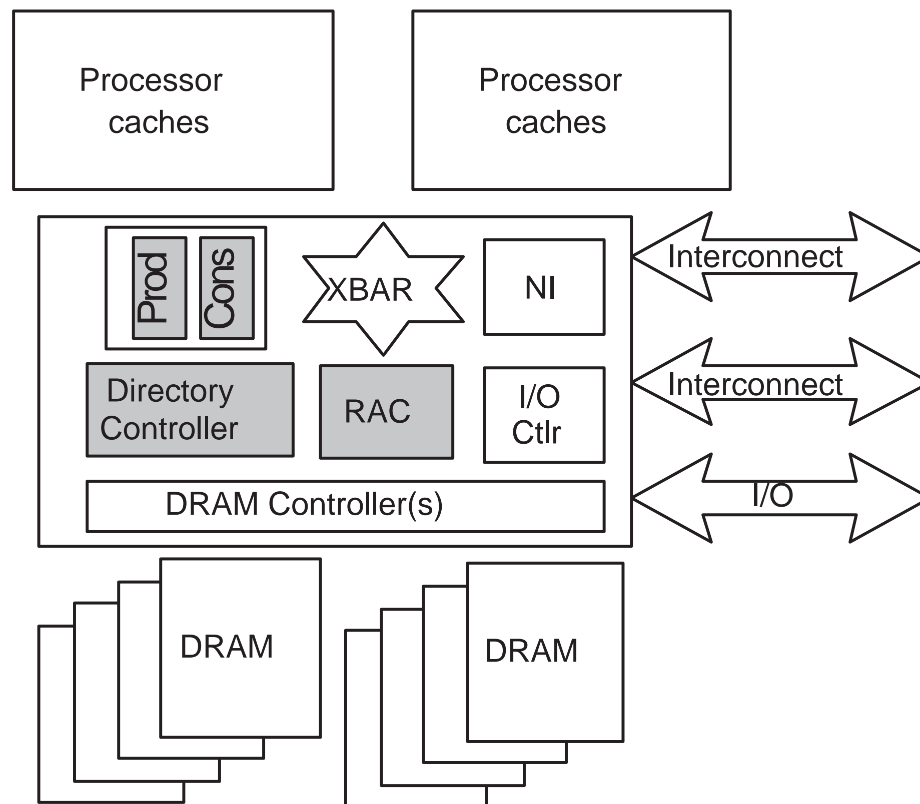

Figure 3.3. Modeled node architecture (new components are highlighted)

extensions and hardware required to implement our speculative update mechanism. Finally, in Section 3.2.5, we describe how we verified the correctness of our protocol extensions using the Murphi model checker.

3.2.1 Remote Access Cache

Traditional CC-NUMA machines can only store remote data in their processor caches. Remote access caches (RACs) eliminate unnecessary remote misses induced by the small size and associativity of processor caches by acting as victim caches for remote data [85]. However, the large caches in modern processors have largely eliminated remote conflict and capacity misses, so modern shared memory multiprocessors do not include RACs.

We propose to augment each node's "hub" with a RAC for three reasons. First, the RAC can be used as a traditional victim cache for remote data. Second, the RAC gives us a location into which we can *push* data at a remote node; researchers often assume the ability to push data into processor caches, but this capability is not supported by modern processors. Third, we employ a portion of the RAC as a surrogate "main memory" for cache lines that have been delegated to the local node. In particular, for each cache line delegated to the local node, we pin the corresponding cache line in the local RAC.

3.2.2 Sharing Pattern Detection

To exploit delegation and speculative updates, we first must identify cache lines that exhibit a stable producer-consumer sharing pattern. In this section we describe the producer-consumer pattern detector that we employ.

We define producer-consumer sharing as a repetitive pattern wherein one processor updates a block (producer) and then an arbitrary number of processors read the block (consumer(s)), which corresponds to the following regular expression:

$$...(W_i)(R_{\forall j:j\neq i})^+(W_i)(R_{\forall k:k\neq i})^+... \tag{3.1}$$

In the expression above, R_i and W_i represent read and write operations by processor i, respectively.

Coherence predictors can be classified into two categories: instruction-based [71] and address-based [103]. Instruction-based mechanisms require less hardware overhead, but require tight integration into the processor core. Since one of our goals is to require no changes to the processor core, we focus on address-based predictors. A problem with address-based predictors implemented outside of the core is they can only observe addresses of accesses that miss in the processor caches. Also, they typically require extensive storage to track access histories, e.g., Lai et al. [80] add one history entry per memory block to trace sharing patterns, which is roughly a 10% storage overhead.

To address these problems, we extend each node's directory controller to track block access histories. Since the directory controller coordinates all accesses to memory blocks homed by that node, it has access to the global access history of each block that it manages. To minimize space overhead, we only track the access histories of blocks whose directory entries reside in the directory cache, not all blocks homed by a node. Typically, a directory cache only contains a small number of entries, e.g., 8k entries on SGI Altix systems, which corresponds to only a fraction of the total memory homed on a particular node. However, the blocks with entries in the directory cache are the most recently shared blocks homed on that node. We found that tracking only their access histories detects the majority of the available producer-consumer sharing patterns, which corroborates results reported by Martin et al. [92].

To detect producer-consumer sharing, we augment each directory cache entry with three fields: *last writer*, *reader count*, and a *write-repeat counter*. The last writer field (four bits) tracks the last node to perform a write operation. The reader count (two bits, saturating) remembers the number of read requests from unique nodes since the last write operation. The write-repeat counter (two bits, saturating) is incremented each time two consecutive write operations are performed by the same node with at least one intervening read. Our detector marks a memory block as being producer-consumer whenever the write-repeat counter saturates. These

extra eight bits, which increase the size of each directory cache entry by 25%, are not saved if the directory entry is flushed from the directory cache.

The detector logic we employ is very simple, which limits its space overhead but results in a conservative predictor that misses some opportunities for optimization, e.g., when more than one node writes to a cache line. An important area of future work is to experiment with more sophisticated producer-consumer sharing detectors. However, as discussed in Section 3.3, even this simple detector identifies ample opportunities for optimization.

3.2.3 Directory delegation

Once the predictor identifies a stable producer-consumer sharing pattern, we delegate directory management to the producer node, thereby converting 3-hop misses into 2-hop misses. To support delegation, we augment the directory controller on each node with a *delegate cache*. The delegate cache consists of two tables: a *producer table* that tracks the directory state of cache lines delegated to the local node and a *consumer table* that tracks the delegated home node of cache lines being accessed by the local node. The producer table is used to implement delegated directory operations. Its entries include the directory information normally tracked by the home node (DirEntry), a valid bit, a tag, and an age field. The number of cache lines that can be delegated to a particular node at a time is limited by the size of the producer table. The consumer table allows consumers to send requests directly to the delegated (producer) node, bypassing the default home node. Its entries consist of a valid bit, a tag, and the identity of the new delegated home node for the corresponding cache line. The number of cache lines that a node will recognize as having a new (remote) home node is limited by the size of the consumer table. The format of delegate cache entries are shown in Figure 3.4.

When a node needs to send a message to a cache line's home, e.g., to acquire a copy or request ownership, it first checks the local delegate cache. If it finds a corresponding entry in the producer table, it handles the request locally. If it finds

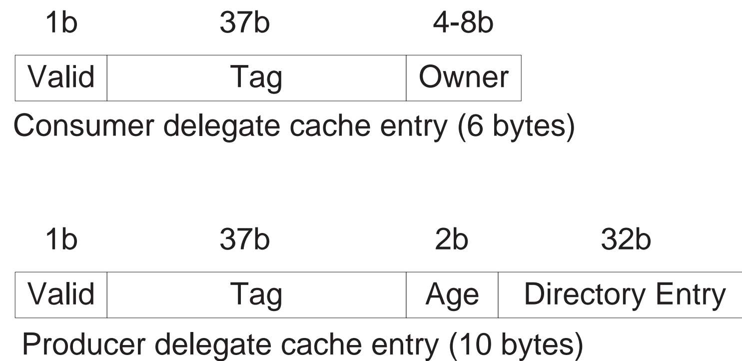

Figure 3.4. Format of delegate cache table entries

an entry in the consumer table, it forwards the request to the delegated home node recorded in the entry. Otherwise, it sends the request to the default home node.

The following subsections describe how we initiate directory delegation (Section 3.2.3.1), how coherence requests are forwarded during delegation (Section 3.2.3.2), and how we undelegate nodes (Section 3.2.3.3).

3.2.3.1 Delegation

Figure 3.5 illustrates the transition diagram for directory delegation. After the home detects a stable producer-consumer sharing pattern, it marks the producer as

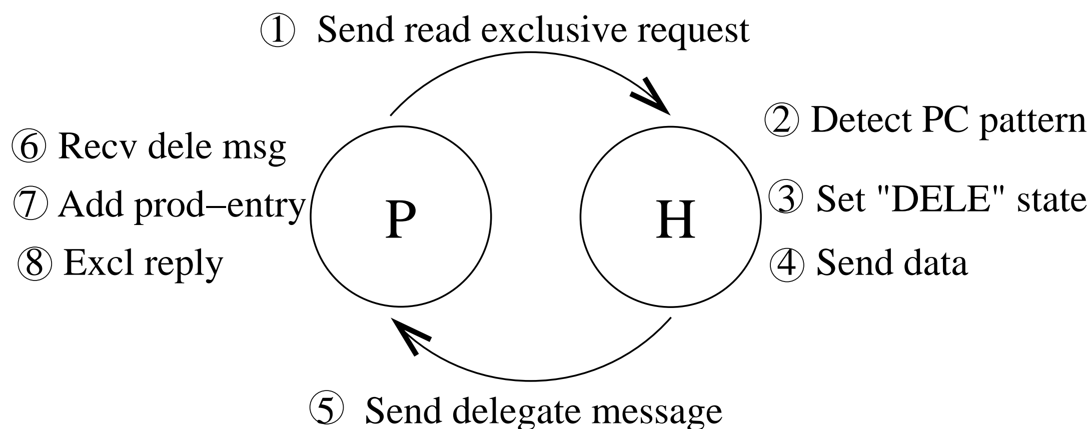

Figure 3.5. Directory delegation

the new owner of the cache line, changes the directory state to DELE, and sends the producer a DELEGATE message that includes the current directory information. Upon receiving this message, the producer adds an entry to its producer delegate table, treats the message as an exclusive reply message, and pins the corresponding RAC entry so that there is a place to put the data should it be flushed from the processor caches.

3.2.3.2 Request forwarding

Figure 3.6 shows the transition diagram for request forwarding. While the cache line is in the DELE state on the original home node, coherence messages sent to the original home node are forwarded to the delegated home node. The original home node also replies to the requester to notify it that the producer is acting as the home node until further notice, in response to which the requesting node adds an entry in its consumer table in the delegate cache. Subsequent coherence

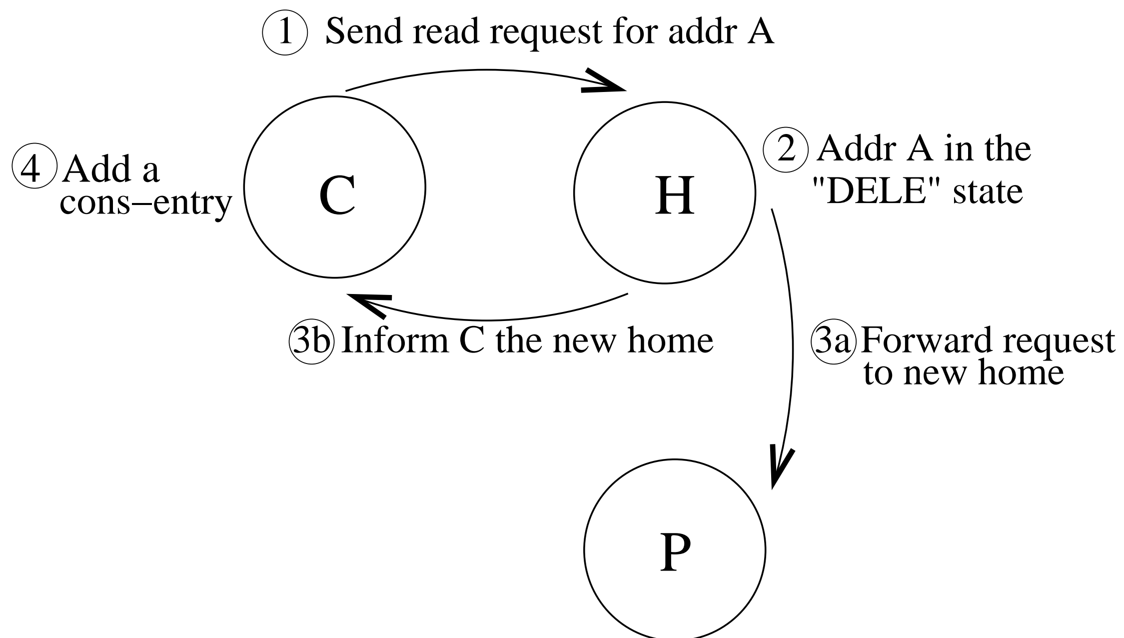

Figure 3.6. Request forwarding

requests from this node are sent directly to the delegated home node until the line is undelegated.

Entries in the consumer table are *hints* as to which node is acting as the home node for a particular cache line, so it can be managed with any replacement policy. If a consumer table entry becomes stale or is evicted, the consumer can still locate the acting home node, but will require extra messages to do so. In our design, the consumer table is four-way set associative and uses random replacement.

3.2.3.3 Undelegation

Figure 3.7 shows the transition diagram for directory undelegation. In our current design, a node undelegates control over a cache line for three reasons:

1. the delegated home node runs out of space in its producer table and needs to replace an entry,
2. the delegated home node flushes the corresponding cache line from its local caches, or
3. another node requests an exclusive copy.

To undelegate a cache line, the producer node invalidates the corresponding entry in its producer table and sends an UNDELE message to the original home node, including the current directory state (DirEntry) and cache block contents (if dirty). When the original home node receives the UNDELE message, it changes the local state of the cache line from DELE to UNOWNED or SHARED, depending on

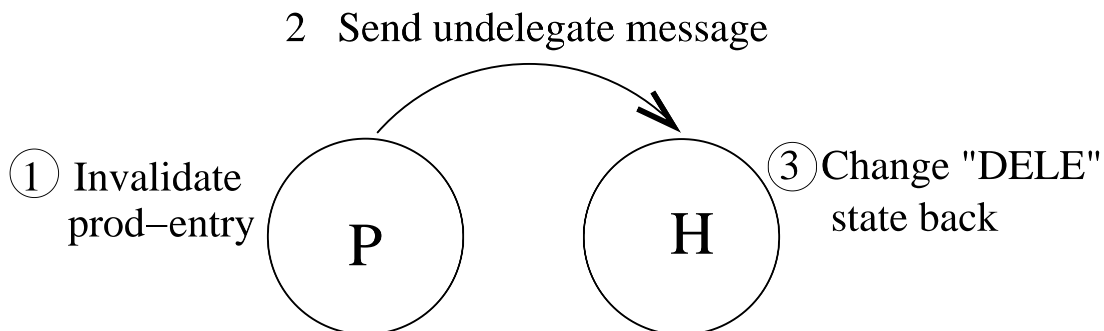

Figure 3.7. Directory undelegation

the contents of DirEntry. If the reason for undelegation is a request for exclusive ownership by another node, the UNDELE message includes the identity of this node and the original home node can handle the request.

3.2.3.4 Discussion

There are a number of race conditions and corner cases that need to be handled carefully to make delegation, forwarding, and undelegation work correctly. For example, it is possible for a request by the producer to arrive at the original home node while it is in the process of delegating responsibility to the producer. Or, the producer may receive a coherence request for a line for which it has undelegated responsibility and for which it no longer has any information in its producer table. To handle these kinds of races, we employ the mechanism used by SGI to simplify its coherence protocol implementations: NACK and retry. In the first case the original home node NACKs the request from the producer, causing it to retry the operation. When the producer retries the operation it will most likely find that it has become the acting home node, otherwise its requests will be NACKed until the delegation operation succeeds. Similarly, if the producer receives a coherence operation for a line that it undelegated, it NACKs and indicates that it is no longer the acting home node, which causes the requesting node to remove the corresponding entry from its consumer table and re-send the request to the default home node. In Section 3.2.5, we describe how we formally verify the correctness of our delegation, forwarding, and undelegation protocols.

3.2.4 Speculative updates

On cc-NUMA systems, applications with significant producer-consumer sharing suffer from frequent remote read misses induced by the invalidations used to ensure consistency. To reduce or eliminate these remote read misses, we extend our directory delegation mechanism to support selective updates. To ensure sequential consistency and simplify verification, we implement our update mechanism as an optimization applied to a conventional write invalidate protocol, rather than replacing

the base coherence protocol with an update protocol. Reads and writes continue to be handled via a conventional directory-based write invalidate protocol. However, after a cache line has been identified as exhibiting producer-consumer sharing and been delegated, producers send speculative update messages to the identified consumers shortly after each invalidation. In essence, we support a producer-driven "speculative push", analogous to a consumer-driven prefetch mechanism. These pushes are a performance optimization and do not impact correctness or violate sequential consistency.

To support speculative updates, we must overcome two challenges. The first is determining when to send updates and what data to send *without modifying the processor*. The second is limiting the use of updates to situations where we have high confidence that the updates will be consumed. We address these challenges in the following subsections.

3.2.4.1 Delayed Intervention

Modern processors directly support only invalidate protocols. Before performing a write, processor requests exclusive access to a cache line. After becoming the exclusive owner, a processor is free to modify it as many times as it likes and can refrain from flushing those changes until another processor attempts to access the cache line.

To generate updates without modifying the processor, we employ a *delayed intervention* mechanism. When the producer wishes to modify a cache line, it requests exclusive access to the line. As in a normal write invalidate protocol, we invalidate all other nodes that have a shared copy. After the producer's hub grants the producer exclusive access to the data, it delays for a short period and then sends an intervention request to the processor, requesting that it downgrade the cache line to SHARED mode. In response to this intervention, the processor flushes the dirty contents of the cache line. The producer's hub writes the flushed data into the local RAC and then sends update messages to the predicted consumers. The mechanism via which we predict the consumer set is described below.

Our delayed intervention mechanism could be thought of as an extremely simple last write predictor, analogous to last touch predictor [81]. To avoid modifying the processor, we simply wait a short time and then predict that the current write burst has completed, rather than employing large tables to track past access histories. Lai et al. used last touch prediction to dynamically self-invalidate cache lines, whereas we employ last write prediction to dynamically downgrade from EXCL to SHARED mode and generate speculative updates. By self-invalidating, Lai et al. convert read misses from 3-hop misses to 2-hop misses, whereas we use updates to convert 3-hop misses into 0-hop (local) misses. The downside of an overly aggressive downgrade decision has less impact in our design, since the producer retains a SHARED copy of the block. For these reasons, a less accurate predictor suffices to achieve good performance improvements.

It is important that delayed intervention not occur too soon or too long after data is supplied to the processor. If too short of an interval is used, the processor may not have completed the current write burst and will suffer another write miss. Overly long intervals result in the data not arriving at consumers in time for them to avoid suffering read misses. For simplicity, we use a fixed (configurable) intervention delay of 50 processor cycles. In Section 3.3.3 we show that performance is largely insensitive to delay intervals between 5 and 5000 cycles. Developing a more adaptive intervention mechanism is part of our future plans.

3.2.4.2 Selective Updates

To achieve good performance, we must accurately predict which nodes will read an update before the cache line is modified again. An overly aggressive mechanism will generate many useless updates, while an overly conservative mechanism will not eliminate avoidable remote read misses. To limit the number of unnecessary updates, we only send updates for cache lines that exhibit producer-consumer behavior and only send updates to nodes that were part of the sharing vector, i.e., the nodes that consumed the last update. Due to the way producer-consumer

sharing works, these are the nodes most likely to consume the newly written data. We track these nodes as follows.

In a traditional invalidated-based protocol, when the state changes from SHARED to EXCL, the sharing vector in the corresponding directory is replaced by the NodeID of the exclusive owner. To track the most recent consumer set, we add an ownerID field to the directory entry and use the old sharing vector to track the nodes to send updates, only overwriting it when a new read request is received.

3.2.4.3 Summary of Control Flow

Figure 3.8 shows the control flow of our speculative update operation. The producer node starts with a shared copy of the cache line that has already been

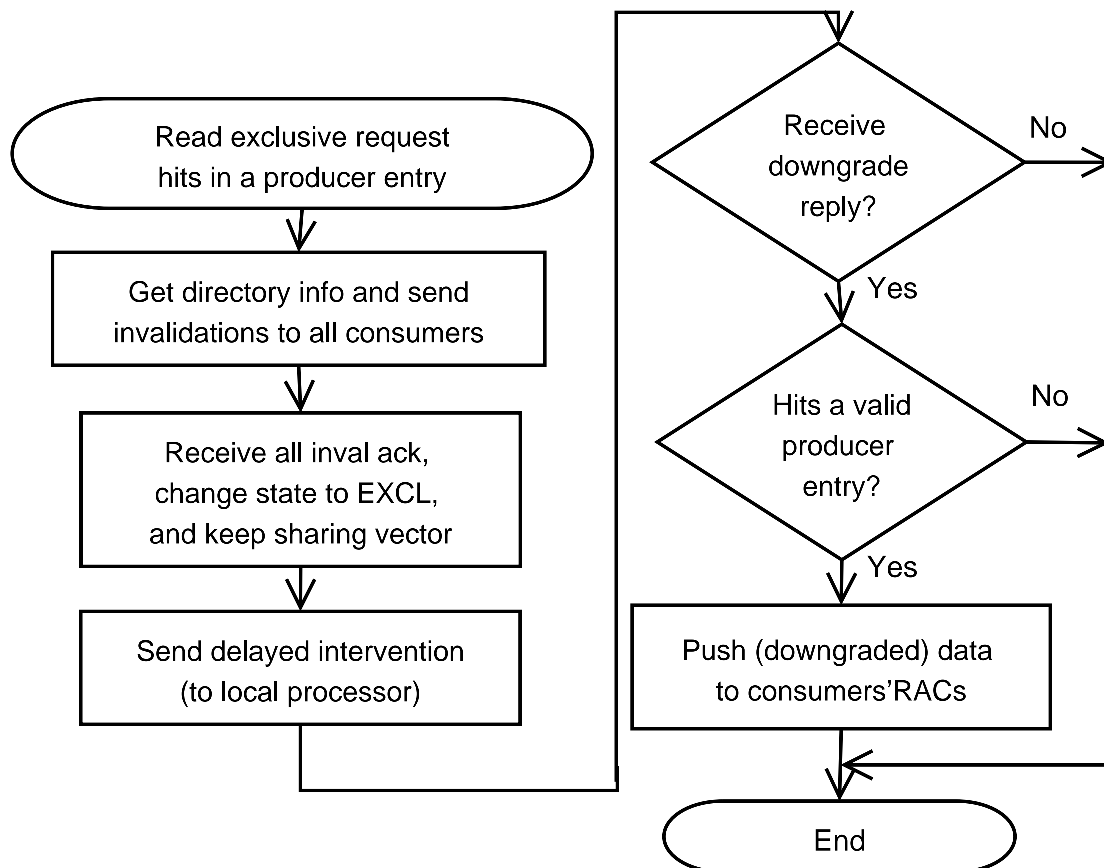

Figure 3.8. Flow of speculative updates

delegated. When the producer writes the data, the local coherence engine loads the directory information from the producer entry in to the local delegate cache. After invalidating all shared copies, the producer's hub changes the state to EXCL, without overwriting the sharing vector, and gives the requesting processor exclusive access. Several cycles later, the hub issues a delayed intervention on the local system bus, which causes the producer processor to downgrade the cache line and write back its new contents. After receiving the new data, the hub sends an update to each node listed in the sharing vector. Upon receipt of an update, a consumer places the incoming data in the local RAC. If the consumer processor has already requested the data, the update message is treated as the response.

3.2.5 Verification

We formally verified that the mechanisms described above do not violate sequential consistency, introduce deadlocks or livelocks, or have other race conditions or corner cases that violate correctness. We applied the standard method for debugging cache coherence protocols: we built a formal model of our protocols and performed an exhaustive reachability analysis of the model for a small configuration size [99, 41, 38] using explicit-state model checking with the Murphi [47] model checker.

We extended the DASH protocol model provided as part of the Murphi release, ran the resulting model through Murphi, and found that none of the invariants provided in the DASH model were violated by our changes. We modeled the delegate cache as described in Section 3.2.3, and also the additional request message types and reply message types required for various transitions described in Section 3.2.3.1, Section 3.2.3.2, and Section 3.2.3.3.

In out implementation, the home directory can be delegated to any node which currently holds an exclusive copy of the data. Note that the precondition of this rule "any node holds an exclusive copy" is weaker than the original precondition described in Section 3.2.3: "the node holds an exclusive copy and repeatedly produces the data." However, this weakening makes the rule more permissive. In

other words, all invariants that are true in this abstract model must be true in the original model. Our abstract model was verified using the default configuration, which consists of one home cluster and three remote clusters.

Moreover, we applied invariant checking to our simulator to bridge the gap between the abstract model and the simulated implementation and again found that no invariants are violated. More specifically, we tested both Murphi's "single writer exists" and "consistency within the directory" invariants at the completion of each transaction that incurs a L2 miss.

3.3 Evaluation

3.3.1 Simulator Environment

We use a cycle-accurate execution-driven simulator, UVSIM, in our performance study. UVSIM models a hypothetical future-generation SGI Altix architecture that we are investigating along with researchers from SGI. Table 3.1 summarizes the major parameters of our simulated system. The simulated interconnect is based on SGI's NUMALink-4, which uses a fat-tree structure with eight children on each nonleaf router. The minimum-sized network packet is 32 bytes and we model a network hop latency of 50nsecs (100 CPU cycles). We do not model contention within the routers, but do model hub port contention.

Table 3.1. System configuration.

Parameter	**Value**
Processor	4-issue, 48-entry active list, 2GHz
L1 I-cache	2-way, 32KB, 64B lines, 1-cycle lat.
L1 D-cache	2-way, 32KB, 32B lines, 2-cycle lat.
L2 cache	4-way, 2MB, 128B lines, 10-cycle lat.
System bus	16B CPU to system, 8B system to CPU
	max 16 outstanding L2C misses, 1GHZ
DRAM	4 16-byte-data DDR channels
Hub clock	500 MHz
DRAM	200 processor cycles latency
Network	100 processor cycles latency per hop

3.3.2 Results

We model a 16-processor system. Table 3.2 presents the input data sets of the applications we use in this study. We consider a mix of scientific applications that exhibit varying degrees of producer-consumer sharing. Barnes and Ocean (contig) are from the SPLASH-2 benchmark suite [120]; EM3D is a shared memory implementation of the Split-C benchmark; LU, CG, MG and Appbt are NAS Parallel Benchmarks (NPB) [15]. We use Omni's OpenMP versions of the NPBs [3].

All results reported in this chapter are for the parallel phases of these applications. Data placement is done by SGI's first-touch policy, which tends to be very effective in allocating data to processors that use them.

Figure 3.9 - 3.11 present the execution time speedup, network message reduction and remote miss reduction for each application for a range of machine configurations. All results are scaled relative to the baseline system runs. For each application, we also show the results for a system with a 32K RAC and no delegation or update mechanisms, a system with 32-entry delegate tables and a 32K RAC, and a system with 1K-entry delegate tables and a 1M RAC. Other than the baseline and RAC-only systems, the results include both directory delegation and selective updates. We omit results for delegation-only, as we found that the benefit of turning 3-hop misses into 2-hop misses roughly balanced out the overhead of delegation, which resulted in performance within 1% of the baseline system for most applications.

Table 3.2. Applications and data sets

Application	**Problem size**
Barnes	16384 nodes, 123 seed
Ocean	258*258 array, 1e-7 error tolerance
Em3D	38400 nodes, degree 5, 15% remote
LU	16*16*16 nodes, 50 testes
CG	1400 nodes, 15 iteration
MG	32*32*32 nodes, 4 steps
Appbt	16*16*16 nodes, 60 timesteps

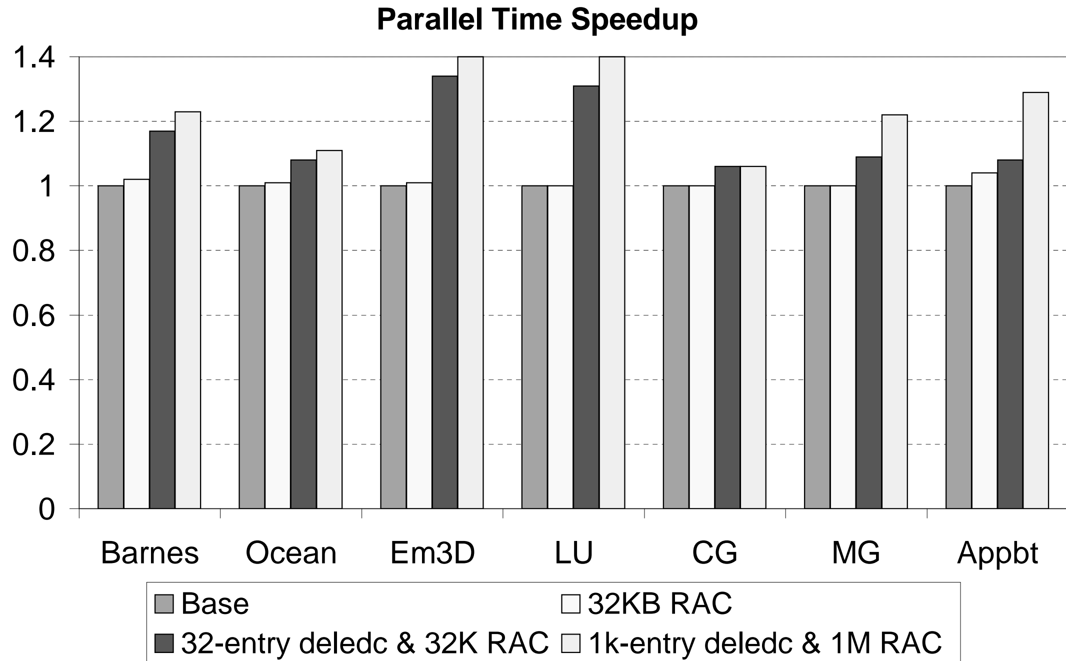

Figure 3.9. Application speedup

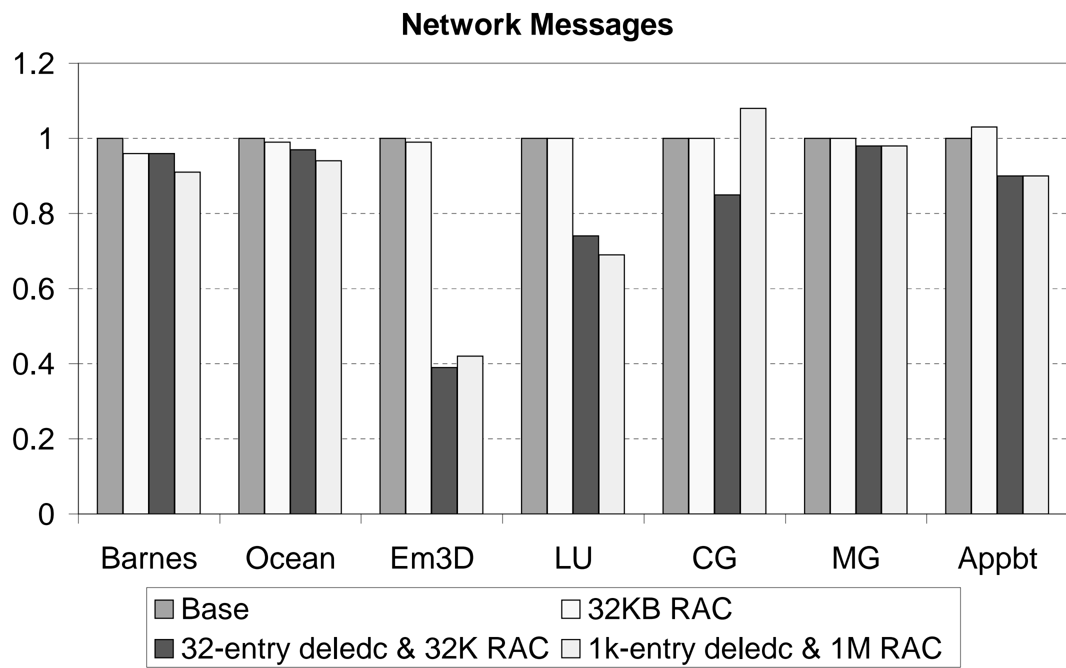

Figure 3.10. Application network messages

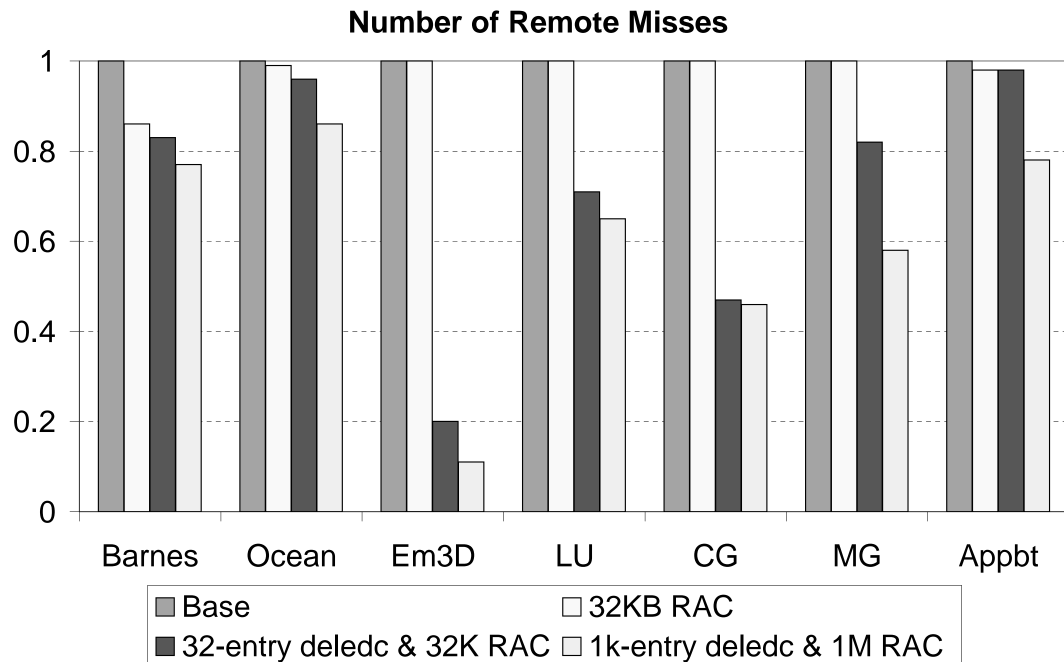

Figure 3.11. Application remote misses

Barnes simulates the interaction of a system of bodies in three dimensions using the Barnes-Hut hierarchical N-body method. The main data structure is an octree with leaves containing information on each body, and internal nodes representing space cells. During each iteration, processors traverse the octree to calculate forces between the bodies and rebuild the octree to reflect the movement of bodies. Although the communication patterns are dependent on the particle distribution, which changes with time, barnes exhibits stable producer-consumer sharing patterns during each phase. The octree data structure inherently results in there being a significant number of consumers per producer, as shown in Table 3.3. Therefore, the benefits of selective updates within each phase are substantial. Even a small delegate cache and RAC (32-entries and 32K respectively) eliminates roughly 20% of the baseline version's remote misses, which leads to a 17% performance improvement. This performance improvement grows to 23% for the larger delegate cache/RAC configuration.

Table 3.3. Number of consumers in the producer-consumer sharing patterns

Application	Number of Consumers (%)				
	1	**2**	**3**	**4**	**4+**
Barnes	13.9	6.8	9.4	8.1	61.7
Ocean	97.7	1.8	0.5	0	0
Em3D	67.8	32.2	0	0	0
LU	99.4	0	0	0.4	0.1
CG	0.1	0.2	0	0	99.7
MG	78.3	11.4	3.7	2.6	3.9
Appbt	0	0.3	6.7	1.4	91.6
Average	51.0	7.5	2.9	1.8	36.7

Ocean models large-scale ocean movements based on eddy and boundary currents. Processors communicate with their immediate neighbors, so nodes along processor allocation boundaries exhibit single producer single consumer sharing. The small number of consumers per producer limits the potential benefits of our speculative update mechanism, as illustrated by the modest remote miss reduction and performance benefits (8% for the small RAC/delegate cache configuration and 11% for the larger one).

Em3d models the propagation of electromagnetic waves through objects in three dimensions. It includes two configuration parameters that govern the extent of producer-consumer sharing: *distribution span* indicates how many consumers each producer will have while *remote links* controls the probability that the producer and consumer are on the different nodes. We use a *distribution span* of 5 and *remote links* probability of 15%.

Selective updates improve performance by 33-40% due largely to a dramatic reduction in coherence message traffic (60%) and an even larger reduction in remote misses (80-90%). The benefits of eliminating remote misses are obvious, but there are several other phenomenon at work that help eliminate network traffic. First, delegating the home directory to the producer node converts 3-hop misses in to 2-hop misses. A less obvious source of traffic reduction comes from eliminating NACKs induced when barrier synchronization is implemented on top of a write

invalidate protocol. After a barrier is crossed, a large number of nodes often attempt to read the same invalidated cache line simultaneously, which causes congestion at the data's home directory. To avoid queueing and potential deadlocks, the directory NACKs requests for a particular cache line if it is BUSY performing another operation on that line. In em3d, NACK messages caused by this "reload flurry" phenomenon represent a nontrivial percentage of network traffic and are largely removed by speculative updates.

LU solves a finite difference discretization of the 3D compressible Navier-Stokes equations through a block-lower block-upper approximate factorization of the original difference scheme. The LU factored form is solved using successive over-relaxation (SOR). A 2D partitioning of the grid onto processors divides the grid repeatedly along the first two dimensions, alternately x and then y, which results in vertical columns of data being assigned to individual processors. Boundary data exhibit stable producer-consumer sharing. Again, even a small delegate cache and RAC are able to achieve good performance improvements (31% speedup, 26% traffic reduction, and 30% remote miss reduction), while a larger configuration achieves a 40% speedup, 30% traffic reduction, and 35% remote miss reduction.

CG uses the conjugate gradient method to compute an approximation of the smallest eigenvalue of a large sparse symmetric positive definite matrix. Multiple issues limit the speedup of CG to 6%. First, CG exhibits producer-consumer sharing only during some phases. Second, the sparse matrix representation used in CG exhibits a high degree of false sharing. Our simple cache line-grained producer-consumer pattern detector avoids designating such cache lines as good candidates for selective updates, which limits the potential performance benefits. Third, and most important, remote misses are not a major performance bottleneck, so even removing roughly 60% of them does not improve performance dramatically.

MG is a simplified multigrid kernel that solves four iterations of a V-cycle multigrid algorithm to obtain an approximate solution to a discrete Poisson equation. The V-cycle starts with the finest grid, proceeds through successive levels to the coarsest grid, and then walks back up to the finest grid. At the finest grid size,

boundary data exhibits producer-consumer sharing. At coarse grid sizes, two pieces of dependent data are most likely on different processors, so more data exhibits producer consumer sharing. In fact, a 32-entry delegate cache is too small to hold all cache blocks identified as exhibiting producer-consumer sharing, which limits the number of remote misses that can be removed to 20% and the performance improvement to 9%. Increasing the delegate cache size to include 1K-entry tables increases the performance benefit to 22%. Note that the larger configuration results in almost the same network traffic as the baseline system, largely to due being overly aggressive in sending out updates. As expected, eliminating remote misses has a larger impact on performance than eliminating network traffic, so the larger configuration achieves better performance despite being less effective at eliminating network traffic.

Appbt is a three-dimensional stencil code in which a cube is divided into subcubes that are assigned to separate processors. Gaussian elimination is then performed along all three dimensions. Like MG, Appbt exhibits significant producer-consumer sharing along subcube boundaries. Like MG, the small RAC/delegate cache is able to capture only a fraction of the performance benefit of the large configuration, 8% compared to the 24% speedup achieved by the large configuration.

Overall, the geometric mean speedup of our speculative update mechanism across the seven benchmark programs using the small RAC/delegate cache is 13%, while network traffic is reduced by an arithmetic mean of 17% and 29% of remote misses are eliminated. If we increase the size of the delegate cache tables to 1K entries and the RAC size to 1MB, mean speedup increases to 21%, with a 15% reduction in message traffic and 40% reduction in remote misses.

In general, the primary benefit of speculative updates comes from removing remote misses, especially for em3d and MG. However, in some cases eliminating a significant number of remote misses does not translate directly into large runtime improvements.

3.3.3 Sensitivity Analysis

3.3.3.1 Equal Storage Area Comparison

We first compare our proposed system to systems with equal silicon area. To support 32-entry delegate tables and a 32KB RAC requires roughly 40KB of extra SRAM per node, plus a small amount of control logic and wire area. This estimate is derived as follows. A 32-entry delegate table requires 320 bytes. Extending the directory cache to support the sharing pattern predictor adds 8 bits to each directory entry (a 4-bit last writer field, a 2-bit reader counter, and a 2-bit write-repeat counter), which for a 8192-entry (32KB) directory cache represents an extra 8KB of storage. We do not save these extra bits when a directory entry is flushed, so there is no extra main memory overhead.

In Figure 3.12 we present the performance of a system with 1MB of L2 cache and no extensions, a system with 1MB of L2 cache extended with a 32-entry delegate cache and a 32KB RAC, and an equal silicon area system with 1.04MB of L2 cache and no extensions. Note that a 1.04MB cache involves adding silicon to the

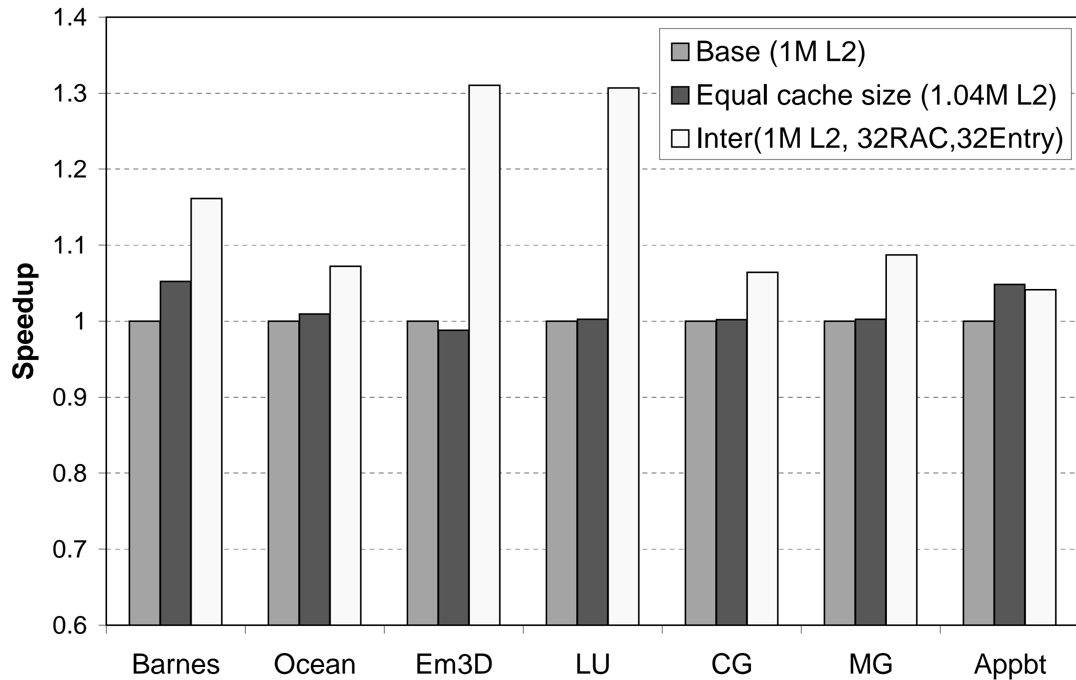

Figure 3.12. Equal storage area comparison (larger L2 caches)

processor die, but we include it to compare the value of building "smarter" versus "larger" caches.

For most benchmarks adding a 32-entry delegate cache and a 32KB RAC yields significantly better performance than simply building a larger L2 cache. The exception is Appbt. Recall that Appbt requires a large RAC to hold all producer-consumer data; a small RAC such as the one modeled here suffers excessive RAC misses, which limits the performance improvement of delegation and updates.

Similarly, additional silicon area can also be used to increase the size of the directory cache as illustrated in Figure 3.13. To handle a coherent data access, the home node must consult the corresponding directory entry to determine the data's sharing state. If the directory entry is not present in the directory cache, the home node need to load the directory entry from the local memory. Altix machines have a 32KB directory cache by default, which can cover 8192 directory entries. To perform an equal silicon area comparison, we model a system with a 1MB L2 cache and a 64KB directory cache versus a system with a 1MB L2 cache, a 32KB

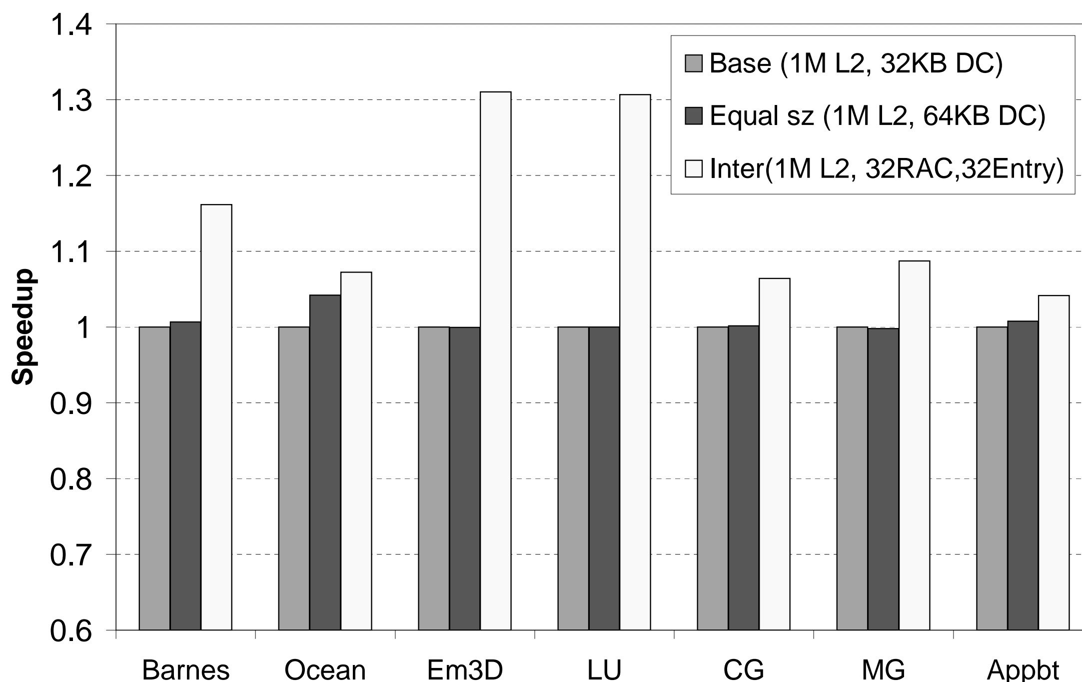

Figure 3.13. Equal storage area comparison (larger directory caches)

RAC, and support for delegation/updates. Both designs involve adding SRAM only outside the processor core. Figure 3.13 shows that doubling the size of directory cache improves the performance by less than 1%, and our proposed design yields significantly better performance for all benchmarks.

3.3.3.2 Sensitivity to Intervention Delay Interval

Figure 3.14 presents the execution time of each application as we vary the delay interval from 5 to 500M cycles normalized to the performance with a 5-cycle delay. A delay interval of 5 cycles results in 1%-5% worse performance than a 50-cycle delay interval, because some write bursts last longer than 5 cycles, which causes extra write misses and updates. As we increase the delay interval beyond 50 cycles, performance degrades at different rates for different applications. Some applications can tolerate a higher delay interval, e.g., MG performance does not deteriorate until the delay interval exceeds 50K cycles, because the average latency between producer writes and consumer reads is large. A delay interval of 50 cycles works well for all

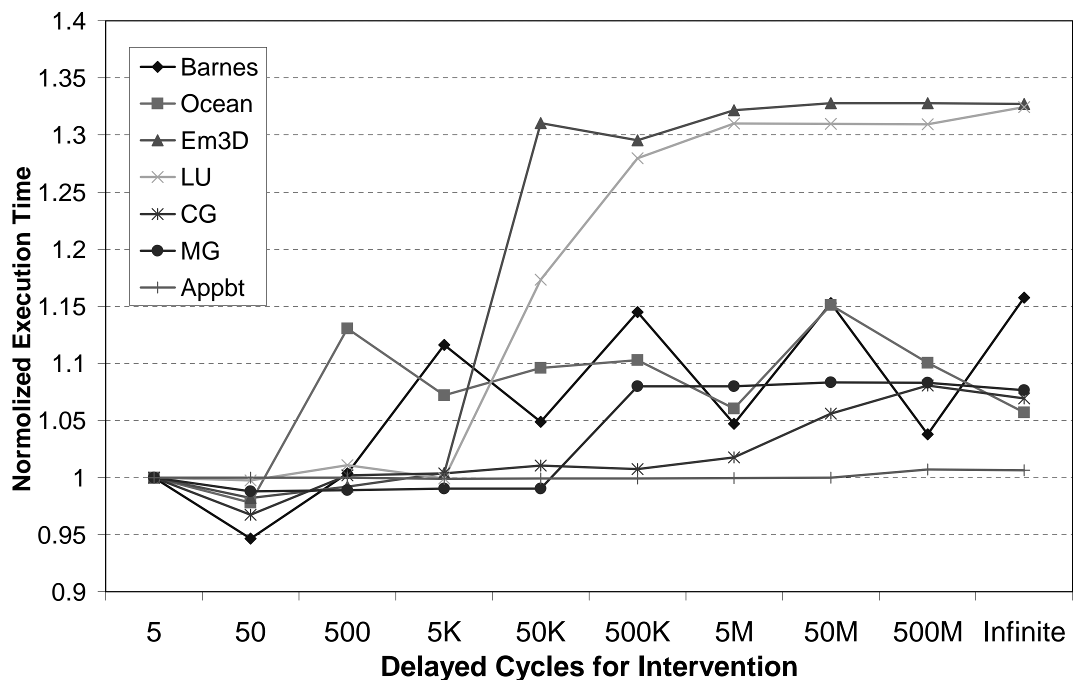

Figure 3.14. Sensitivity to intervention delay

of the benchmarks because it is long enough to capture most write bursts, but short enough to ensure that updates arrive early enough to eliminate consumer read misses.

The performance of Ocean and Barnes varies inconsistently as we vary the delay interval. This phenomenon is caused by a nonobvious interaction between the choice of delay interval and the timing of synchronization in these applications. When the choice of delay interval happens to cause updates to occur at the same time that the application is synchronizing between threads, e.g., as occurs for both Ocean and Barnes with a delay interval of 5M cycles, performance suffers.

As described in Section 3.2.4, the delayed intervention mechanism is employed to generate updates without modifying the processor core. In the following study, we remove this constraint by assuming processors have the ability to push the data directly to remote caches. When a producer writes to a cache line which shows stable producer consumer sharing, it sends update messages to consumers' caches immediately after the write instruction graduates from the pipeline. Figure 3.15 shows the

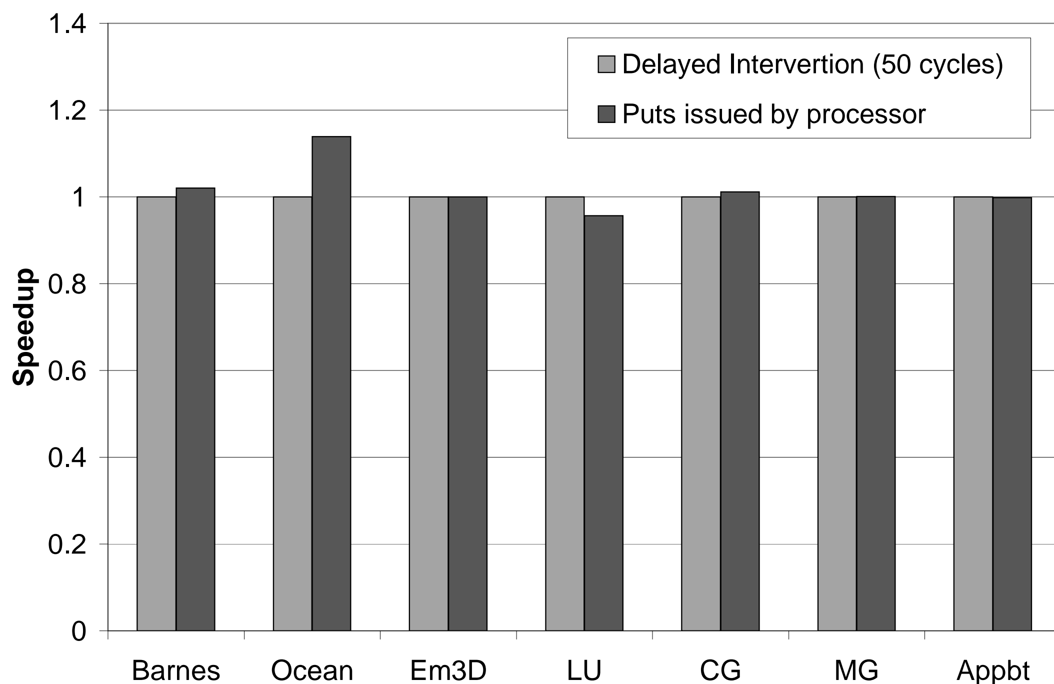

Figure 3.15. Delayed intervention VS. updates issued by processor

performance benefits of processor-issued updates compared with updates generated by the delayed intervention mechanism. The average performance speedup of processor-issued updates over baseline is less than 2%, with most applications within 0.5%. Two exceptions are Ocean and LU. Processor-issued updates improve the performance of Ocean by 12%. This is because updates issued by processors are sent earlier, thus reducing the chance of conflicting with following synchronization messages. In contrast, processor-issued updates reduce the performance of LU by 4% due to excessive long write bursts in LU. Processor-issued updates aggressively downgrade cache lines from exclusive state to shared state, which incurs extra update misses in the long write bursts.

3.3.3.3 Sensitivity to Network Latency

To investigate the extent to which remote miss latency impacts performance, we vary the network hop latencies from 25nsecs to 200nsecs. Figure 3.16 plots the execution time (left y-axis) and speedup (right y-axis) for Appbt, whose performance is representative. We consider only the baseline CC-NUMA system and a system enhanced with a 32K RAC and 32-entry delegate cache tables. Every time network

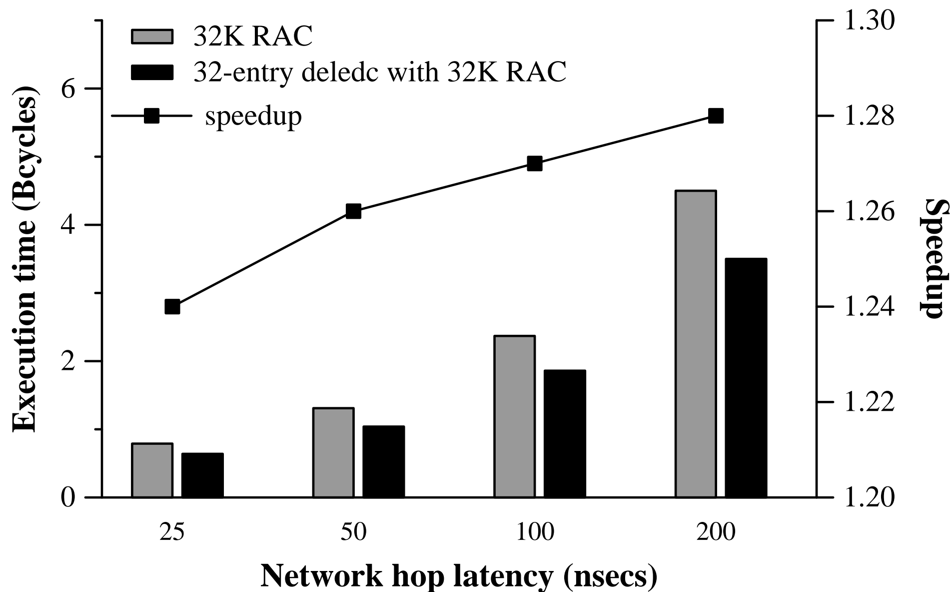

Figure 3.16. Sensitivity to hop latency (Appbt)

hop latency doubles, execution time nearly doubles. Therefore, the value of the mechanisms proposed herein increases as average remote miss latencies increase, albeit only gradually (increasing from a 24% speedup to 28% as we increase hop latency from 25nsecs to 200nsecs).

3.3.3.4 RAC and Delegate Cache

For most benchmarks, a small delegate cache and RAC (32-entry and 32KB) result in significant performance benefits. Two exceptions are MG and Appbt, which are limited by the size of the delegate cache and RAC, respectively. Figure 3.17 shows MG's sensitivity to the delegate cache size. MG has a substantial amount of producer-consumer data, so increasing the delegate cache size improves performance. For Appbt, the performance benefit of delegation/updates is limited by the size of the RAC; increasing the RAC size eliminates this bottleneck and improves performance, as shown in Figure 3.18.

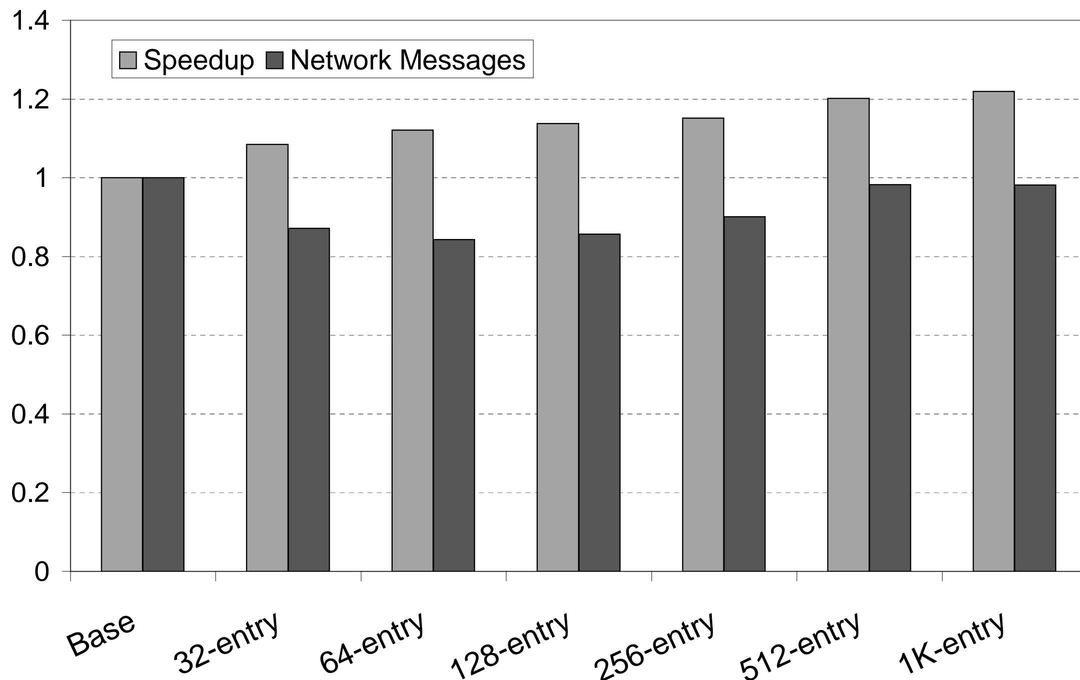

Figure 3.17. Sensitivity to delegate cache sz (MG)

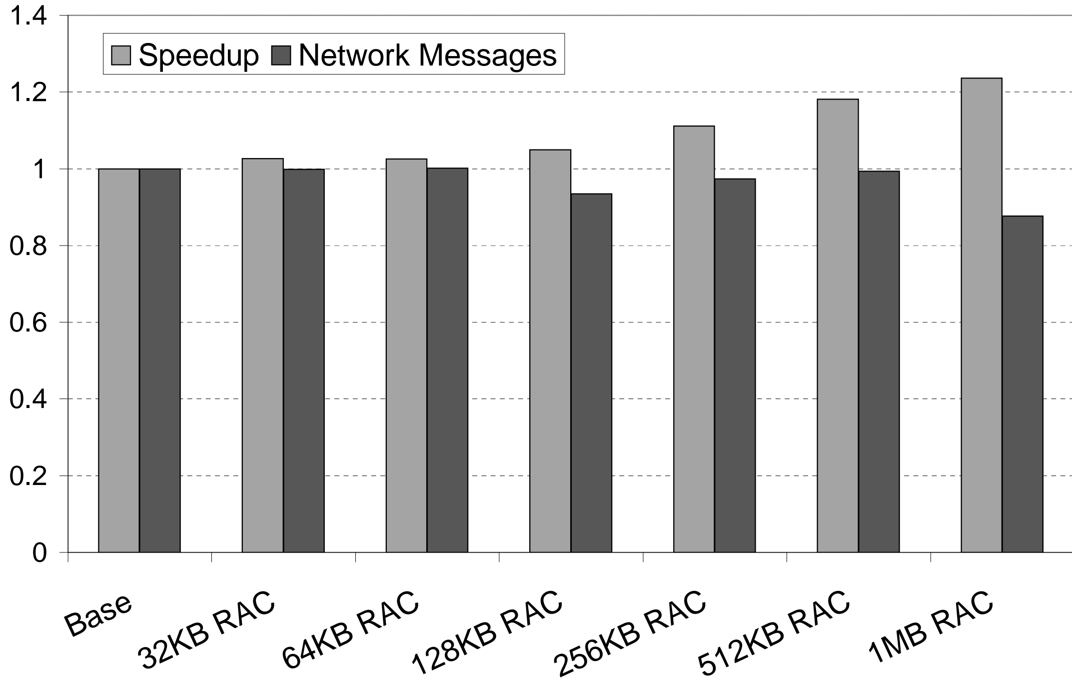

Figure 3.18. Sensitivity to RAC size (Appbt)

3.4 Summary

Our work focuses on the design and analysis of mechanisms that improve shared memory performance by eliminating remote misses and coherence traffic. In this chapter we propose two novel mechanisms, *directory delegation* and *speculative updates*, that can be used to improve the performance of applications that exhibit producer-consumer sharing. After detecting instances of producer-consumer sharing using a simple directory-based predictor, we delegate responsibility for the data's directory information from its home node to the current producer of the data, which can convert 3-hop coherence operations into 2-hop operations. We also present a speculative update mechanism wherein shortly after modifying a particular piece of data, producers speculatively forward updates to the nodes that most recently accessed it.

On a collection of seven benchmark programs, we demonstrate that speculative updates can significantly reduce the number of remote misses suffered and amount of network traffic generated. We consider two hardware implementations, one that

requires very little hardware overhead (a 32-entry delegate cache and a 32KB RAC per node) and one that requires modest overhead (a 1K-entry delegate cache and a 1MB RAC per node). On the small configuration, delegation/updates reduce execution time by 13% by reducing the number of remote misses by 29% and network traffic by 17%. On the larger configuration, delegation/updates reduce program execution time by 21% by reducing the number of remote misses by 40% and network traffic by 15%. Finally, we show that the performance benefits derive primarily from eliminating remote misses, and only secondarily from reducing network traffic.

There are many ways to extend and improve the work reported herein. To minimize the amount of hardware needed, thereby making it easier to adopt our ideas in near future products, we employ a very simplistic producer-consumer sharing pattern and “last write” predictor. Using a simple analytical model, not presented here for space reasons, we found that as network latency grows, the achievable speedup is limited to *1/(1-accuracy).* Thus, we plan to investigate the value of more sophisticated predictors, e.g., one that can detect producer-consumer behavior in the face of false sharing and multiple writers. In addition, we plan to investigate the potential performance benefits of non-sequentially consistent versions of our mechanisms, e.g., ones that issued updates in place of invalidates rather than after invalidates, or that support multiple writers [32].

CHAPTER 4

INTERCONNECT-AWARE COHERENCE PROTOCOLS

In this chapter, we demonstrate how our interconnect-aware coherence protocols optimize the coherence traffic in a Chip Multi-Processors (CMP). Improvements in semiconductor technology have made it possible to include multiple processor cores on a single die. CMPs are an attractive choice for future billion transistor architectures due to their low design complexity, high clock frequency, and high throughput. In a typical CMP architecture, the L2 cache is shared by multiple cores and data coherence is maintained among private L1s. Coherence operations entail frequent communication over global on-chip wires. In future technologies, communication between different L1s will have a significant impact on overall processor performance and power consumption. On-chip wires can be designed to have different latency, bandwidth, and energy properties. Likewise, coherence protocol messages have different latency and bandwidth needs. We propose an interconnect composed of wires with varying latency, bandwidth, and energy characteristics, and advocate intelligently mapping coherence operations to the appropriate wires. In this section, we present a comprehensive list of techniques that allow coherence protocols to exploit a heterogeneous interconnect and evaluate a subset of these techniques to show their performance and power-efficiency potential. Most of the proposed techniques can be implemented with a minimum complexity overhead.

The chapter is organized as follows. We describe the background in Section 4.1. Section 4.2 reviews techniques that enable different wire implementations and the design of a heterogeneous interconnect. Section 4.3 describes the proposed in-

novations that map coherence messages to different on-chip wires. Section 4.4 quantitatively evaluates these ideas and we conclude in Section 4.5.

4.1 Background

One of the greatest bottlenecks to performance in future microprocessors is the high cost of on-chip communication through global wires [64]. Power consumption has also emerged as a first order design metric and wires contribute up to 50% of total chip power in some processors [90]. Most major chip manufacturers have already announced plans [65, 73] for large-scale chip multiprocessors (CMPs). Multithreaded workloads that execute on such processors will experience high on-chip communication latencies and will dissipate significant power in interconnects. In the past, only VLSI and circuit designers were concerned with the layout of interconnects for a given architecture. However, with *communication* emerging as a larger power and performance constraint than *computation*, it may become necessary to understand and leverage the properties of the interconnect at a higher level. Exposing wire properties to architects enables them to find creative ways to exploit these properties. This chapter presents a number of techniques by which coherence traffic within a CMP can be mapped intelligently to different wire implementations with minor increases in complexity. Such an approach can not only improve performance, but also reduce power dissipation.

In a typical CMP, the L2 cache and lower levels of the memory hierarchy are shared by multiple cores [76, 114]. Sharing the L2 cache allows high cache utilization and avoids duplicating cache hardware resources. L1 caches are typically not shared as such an organization entails high communication latencies for every load and store. There are two major mechanisms used to ensure coherence among L1s in a chip multiprocessor. The first option employs a bus connecting all of the L1s and a snoopy bus-based coherence protocol. In this design, every L1 cache miss results in a coherence message being broadcast on the global coherence bus and other L1 caches are responsible for maintaining valid state for their blocks and responding to misses when necessary. The second approach employs a centralized

directory in the L2 cache that tracks sharing information for all cache lines in the L2. In such a directory-based protocol, every L1 cache miss is sent to the L2 cache, where further actions are taken based on that block's directory state. Many studies [6, 25, 66, 80, 86] have characterized the high frequency of cache misses in parallel workloads and the high impact these misses have on total execution time. On a cache miss, a variety of protocol actions are initiated, such as request messages, invalidation messages, intervention messages, data block writebacks, data block transfers, etc. Each of these messages involves on-chip communication with latencies that are projected to grow to tens of cycles in future billion transistor architectures [9].

Simple wire design strategies can greatly influence a wire's properties. For example, by tuning wire width and spacing, we can design wires with varying latency and bandwidth properties. Similarly, by tuning repeater size and spacing, we can design wires with varying latency and energy properties. To take advantage of VLSI techniques and better match the interconnect design to communication requirements, a heterogeneous interconnect can be employed, where every link consists of wires that are optimized for either latency, energy, or bandwidth. In this study, we explore optimizations that are enabled when such a heterogeneous interconnect is employed for coherence traffic. For example, when employing a directory-based protocol, on a cache write miss, the requesting processor may have to wait for data from the home node (a 2-hop transaction) and for acknowledgments from other sharers of the block (a 3-hop transaction). Since the acknowledgments are on the critical path and have low bandwidth needs, they can be mapped to wires optimized for delay, while the data block transfer is not on the critical path and can be mapped to wires that are optimized for low power.

4.2 Wire Implementations

We begin with a quick review of factors that influence wire properties. It is well-known that the delay of a wire is a function of its RC time constant (R is resistance and C is capacitance). Resistance per unit length is (approximately) inversely

proportional to the width of the wire [64]. Likewise, a fraction of the capacitance per unit length is inversely proportional to the spacing between wires, and a fraction is directly proportional to wire width. These wire properties provide an opportunity to design wires that trade off bandwidth and latency. By allocating more metal area per wire and increasing wire width and spacing, the net effect is a reduction in the RC time constant. This leads to a wire design that has favorable latency properties, but poor bandwidth properties (as fewer wires can be accommodated in a fixed metal area). In certain cases, nearly a three-fold reduction in wire latency can be achieved, at the expense of a four-fold reduction in bandwidth. Further, researchers are actively pursuing transmission line implementations that enable extremely low communication latencies [34, 48]. However, transmission lines also entail significant metal area overheads in addition to logic overheads for sending and receiving [23, 34]. If transmission line implementations become cost-effective at future technologies, they represent another attractive wire design point that can trade off bandwidth for low latency.

Similar trade-offs can be made between latency and power consumed by wires. Global wires are usually composed of multiple smaller segments that are connected with repeaters [16]. The size and spacing of repeaters influences wire delay and power consumed by the wire. When smaller and fewer repeaters are employed, wire delay increases, but power consumption is reduced. The repeater configuration that minimizes delay is typically very different from the repeater configuration that minimizes power consumption. Banerjee et al. [18] show that at 50nm technology, a five-fold reduction in power can be achieved at the expense of a two-fold increase in latency.

Thus, by varying properties such as wire width/spacing and repeater size/spacing, we can implement wires with different latency, bandwidth, and power properties. Consider a CMOS process where global intercore wires are routed on the 8X and 4X metal planes. Note that the primary differences between minimum-width wires in the 8X and 4X planes are their width, height, and spacing. We will refer to these mimimum-width wires as baseline *B-Wires* (either 8X-B-Wires or 4X-B-Wires). In

addition to these wires, we will design two more wire types that may be potentially beneficial (summarized in Figure 4.1). A low-latency *L-Wire* can be designed by increasing the width and spacing of the wire on the 8X plane (by a factor of four). A power-efficient *PW-Wire* is designed by decreasing the number and size of repeaters within minimum-width wires on the 4X plane. While a traditional architecture would employ the entire available metal area for B-Wires (either 4X or 8X), we propose the design of a heterogeneous interconnect, where part of the available metal area is employed for B-Wires, part for L-Wires, and part for PW-Wires. Thus, any data transfer has the option of using one of three sets of wires to effect the communication. A typical composition of a heterogeneous interconnect may be as follows: 256 B-Wires, 512 PW-Wires, 24 L-Wires. In the next section, we will demonstrate how these options can be exploited to improve performance and reduce power consumption. We will also examine the complexity introduced by a heterogeneous interconnect.

4.3 Optimizing Coherence Traffic

For each cache coherence protocol, there exist a variety of coherence operations with different bandwidth and latency needs. Because of this diversity, there are many opportunities to improve performance and power characteristics by employing a heterogeneous interconnect. The goal of this section is to present a comprehensive listing of such opportunities. We focus on protocol-specific optimizations in

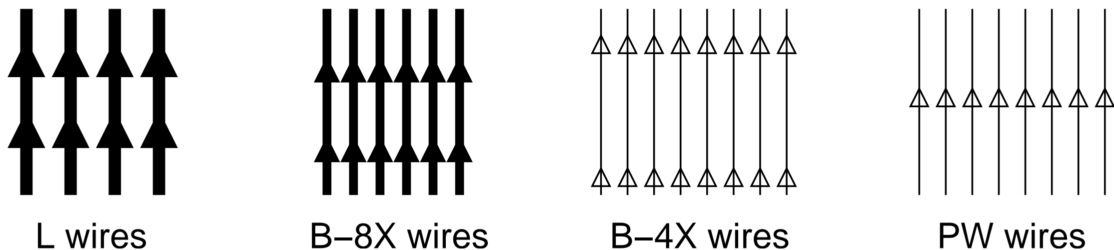

Figure 4.1. Examples of different wire implementations

Section 4.3.1 and on protocol-independent techniques in Section 4.3.2. We discuss the implementation complexity of these techniques in Section 4.3.3.

4.3.1 Protocol-dependent Techniques

We first examine the characteristics of operations in both directory-based and snooping bus-based coherence protocols and how they can map to different sets of wires. In a bus-based protocol, the ability of a cache to directly respond to another cache's request leads to low L1 cache-to-cache miss latencies. L2 cache latencies are relatively high as a processor core has to acquire the bus before sending a request to L2. It is difficult to support a large number of processor cores with a single bus due to the bandwidth and electrical limits of a centralized bus [26]. In a directory-based design [68, 83], each L1 connects to the L2 cache through a point-to-point link. This design has low L2 hit latency and scales better. However, each L1 cache-to-cache miss must be forwarded by the L2 cache, which implies high L1 cache-to-cache latencies. The performance comparison between these two design choices depends on the cache sizes, miss rates, number of outstanding memory requests, working-set sizes, sharing behavior of targeted benchmarks, etc. Since either option may be attractive to chip manufacturers, we will consider both forms of coherence protocols in our study.

Write-invalidate directory-based protocols have been implemented in existing dual-core CMPs [114] and will likely be used in larger scale CMPs as well. In a directory-based protocol, every cache line has a directory where the states of the block in all L1s are stored. Whenever a request misses in an L1 cache, a coherence message is sent to the directory at the L2 to check the cache line's global state. If there is a clean copy in the L2 and the request is a READ, it is served by the L2 cache. Otherwise, another L1 must hold an exclusive copy and the READ request is forwarded to the exclusive owner, which supplies the data. For a WRITE request, if any other L1 caches hold a copy of the cache line, coherence messages are sent to each of them requesting that they invalidate their copies. The requesting L1 cache

acquires the block in exclusive state only after all invalidation messages have been acknowledged.

Hop imbalance is quite common in a directory-based protocol. To exploit this imbalance, we can send critical messages on fast wires to increase performance and send noncritical messages on slow wires to save power. For the sake of this discussion, we assume that the hop latencies of different wires are in the following ratio: L-wire : B-wire : PW-wire :: 1 : 2 : 3

4.3.1.1 Proposal I: Write for Shared Data

In this case, the L2 cache's copy is clean, so it provides the data to the requesting L1 and invalidates all shared copies. When the requesting L1 receives the reply message from the L2, it collects invalidation acknowledgment messages from the other L1s before returning the data to the processor core.[1] Figure 4.2 depicts all generated messages.

The reply message from the L2 requires only one hop, while the invalidation process requires two hops – an example of hop imbalance. Since there is no

[1]Some coherence protocols may not impose all of these constraints, thereby deviating from a sequentially consistent memory model.

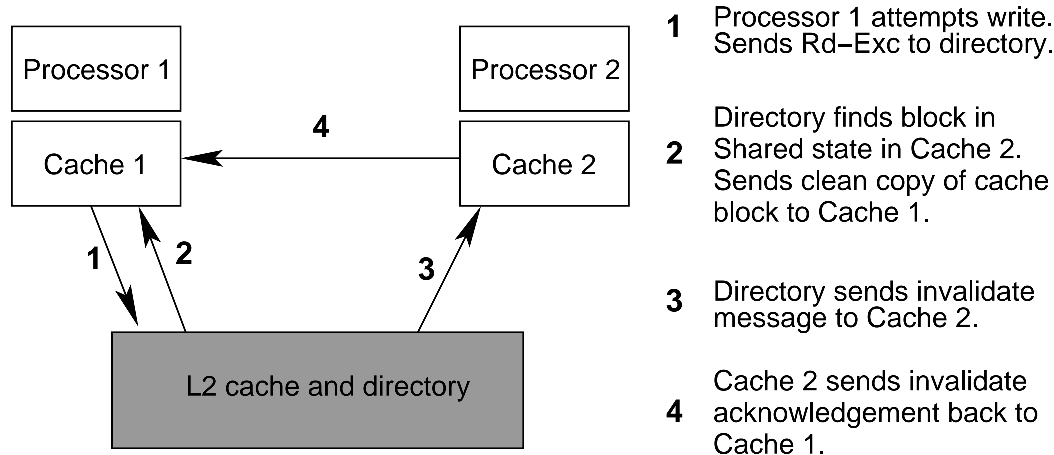

Figure 4.2. Read exclusive request for a shared block in MESI protocol

benefit to receiving the cache line early, latencies for each hop can be chosen so as to equalize communication latency for the cache line and the acknowledgment messages. Acknowledgment messages include identifiers so they can be matched against the outstanding request in the L1's MSHR. Since there are only a few outstanding requests in the system, the identifier requires few bits, allowing the acknowledgment to be transferred on a few low-latency L-Wires. Simultaneously, the data block transmission from the L2 can happen on low-power PW-Wires and still finish before the arrival of the acknowledgments. This strategy improves performance (because acknowledgments are often on the critical path) and reduces power consumption (because the data block is now transferred on power-efficient wires). While circuit designers have frequently employed different types of wires within a circuit to reduce power dissipation without extending the critical path, the proposals in this chapter represent some of the first attempts to exploit wire properties at the architectural level.

4.3.1.2 Proposal II: Read for Exclusive Data

In this case, the value in the L2 is likely to be stale and the following protocol actions are taken. The L2 cache sends a speculative data reply to the requesting L1 and forwards the read request as an intervention message to the exclusive owner. If the cache copy in the exclusive owner is clean, an acknowledgment message is sent to the requesting L1, indicating that the speculative data reply from the L2 is valid. If the cache copy is dirty, a response message with the latest data is sent to the requesting L1 and a write-back message is sent to the L2. Since the requesting L1 cannot proceed until it receives a message from the exclusive owner, the speculative data reply from the L2 (a single hop transfer) can be sent on slower PW-Wires. The forwarded request to the exclusive owner is on the critical path, but includes the block address. It is therefore not eligible for transfer on low-bandwidth L-Wires. If the owner's copy is in the exclusive clean state, a short acknowledgment message to the requestor can be sent on L-Wires. If the owner's copy is dirty, the cache block can be sent over B-Wires, while the low priority writeback to the L2 can

happen on PW-Wires. With the above mapping, we accelerate the critical path by using faster L-Wires, while also lowering power consumption by sending noncritical data on PW-Wires. The above protocol actions apply even in the case when a read-exclusive request is made for a block in the exclusive state.

4.3.1.3 Proposal III: NACK Messages

When the directory state is busy, incoming requests are often NACKed by the home directory, i.e., a negative acknowledgment is sent to the requester rather than buffering the request. Typically the requesting cache controller re-issues the request and the request is serialized in the order in which it is actually accepted by the directory. A NACK message can be matched by comparing the request id (MSHR index) rather than the full address, so a NACK is eligible for transfer on low-bandwidth L-Wires. If load at the home directory is low, it will likely be able to serve the request when it arrives again, in which case sending the NACK on fast L-Wires can improve performance. In contrast, when load is high, frequent backoff-and-retry cycles are experienced. In this case, fast NACKs only increase traffic levels without providing any performance benefit. In this case, in order to save power, NACKs can be sent on PW-Wires.

4.3.1.4 Proposal IV: Unblock and Write Control

Some protocols [97] employ unblock and write control messages to reduce implementation complexity. For every read transaction, a processor first sends a request message that changes the L2 cache state into a transient state. After receiving the data reply, it sends an unblock message to change the L2 cache state back to a stable state. Similarly, write control messages are used to implement a three-phase writeback transaction. A processor first sends a control message to the directory to order the writeback message with other request messages. After receiving the writeback response from the directory, the processor sends the data. This avoids a race condition in which the processor sends the writeback data while a request is being forwarded to it. Sending unblock messages on L-Wires can improve perfor-

mance by reducing the time cache lines are in busy states. Write control messages (writeback request and writeback grant) are not on the critical path, although they are also eligible for transfer on L-Wires. The choice of sending writeback control messages on L-Wires or PW-Wires represents a power-performance trade-off.

4.3.1.5 Proposal V: Signal Wires

We next examine techniques that apply to bus-based protocols. In a bus-based system, three wired-OR signals are typically employed to avoid involving the lower/slower memory hierarchy [46]. Two of these signals are responsible for reporting the state of snoop results and the third indicates that the snoop result is valid. The first signal is asserted when any L1 cache, besides the requester, has a copy of the block. The second signal is asserted if any cache has the block in exclusive state. The third signal is an inhibit signal, asserted until all caches have completed their snoop operations. When the third signal is asserted, the requesting L1 and the L2 can safely examine the other two signals. Since all of these signals are on the critical path, implementing them using low-latency L-Wires can improve performance.

4.3.1.6 Proposal VI: Voting Wires

Another design choice is whether to use cache-to-cache transfers if the data is in the shared state in a cache. The Silicon Graphics Challenge [51] and the Sun Enterprise use cache-to-cache transfers only for data in the modified state, in which case there is a single supplier. On the other hand, in the full Illinois MESI protocol, a block can be preferentially retrieved from another cache rather than from memory. However, when multiple caches share a copy, a “voting” mechanism is required to decide which cache will supply the data, and this voting mechanism can benefit from the use of low latency wires.

4.3.2 Protocol-independent Techniques

4.3.2.1 Proposal VII: Synchronization Variables

Synchronization is one of the most important factors in the performance of a parallel application. Synchronization is not only often on the critical path, but it also contributes a large percentage (up to 40%) of coherence misses [86]. Locks and barriers are the two most widely used synchronization constructs. Both of them use small integers to implement mutual exclusion. Locks often toggle the synchronization variable between zero and one, while barriers often linearly increase a barrier variable from zero to the number of processors taking part in the barrier operation. Such data transfers have limited bandwidth needs and can benefit from using L-Wires.

This optimization can be further extended by examining the general problem of cache line compaction. For example, if a cache line is comprised mostly of 0 bits, trivial data compaction algorithms may reduce the bandwidth needs of the cache line, allowing it to be transferred on L-Wires instead of B-Wires. If the wire latency difference between the two wire implementations is greater than the delay of the compaction/de-compaction algorithm, performance improvements are possible.

4.3.2.2 Proposal VIII: Writeback Data

Writeback data transfers result from cache replacements or external intervention messages. Since writeback messages are rarely on the critical path, assigning them to PW-Wires can save power without incurring significant performance penalties.

4.3.2.3 Proposal IX: Narrow Messagess

Coherence messages that include the data block address or the data block itself are many bytes wide. However, many other messages, such as acknowledgments and NACKs, do not include the address or data block and only contain control information (source/destination, message type, MSHR id, etc.). Such narrow messages can be always assigned to low latency L-Wires to accelerate the critical path.

4.3.3 Implementation Complexity

4.3.3.1 Implementation Overhead

In a conventional multiprocessor interconnect, a subset of wires are employed for addresses, a subset for data, and a subset for control signals. Every bit of communication is mapped to a unique wire. When employing a heterogeneous interconnect, a communication bit can map to multiple wires. For example, data returned by the L2 in response to a read-exclusive request may map to B-Wires or PW-Wires depending on whether there are other sharers for that block (**Proposal I**). Thus, every wire must be associated with a multiplexor and de-multiplexor.

The entire network operates at the same fixed clock frequency, which means that the number of latches within every link is a function of the link latency. Therefore, PW-Wires have to employ additional latches, relative to the baseline B-Wires. Dynamic power per latch at 5GHz and 65nm technology is calculated to be 0.1mW, while leakage power per latch equals 19.8μW [78]. The power per unit length for each wire is computed in the next section. Power overheads due to these latches for different wires are tabulated in Table 4.1. Latches impose a 2% overhead within B-Wires, but a 13% overhead within PW-Wires.

The proposed model also introduces additional complexity in the routing logic. The base case router employs a cross-bar switch and 8-entry message buffers at each input port. Whenever a message arrives, it is stored in the input buffer and routed to an allocator that locates the output port and transfers the message. In case of a heterogeneous model, three different buffers are required at each port

Table 4.1. Power characteristics of different wire implementations) is assumed to be 0.15. The above latch spacing values are for a 5GHz network.

Wire Type	Power/Length mW/mm	Latch Power mW/latch	Latch Spacing mm	Total Power mW/10mm
B-Wire	1.4221	0.119	5.15	14.46
B-Wire	1.5928	0.119	3.4	16.29
L-Wire	0.7860	0.119	9.8	7.80
PW-wire	0.4778	0.119	1.7	5.48

to store L, B, and PW messages separately. In our simulations we employ three 4-entry message buffers for each port. The size of each buffer is proportional to the flit size of the corresponding set of wires. For example, a set of 24 L-Wires employs a 4-entry message buffer with a word size of 24 bits. For power calculations we have also included the fixed additional overhead associated with these small buffers as opposed to a single larger buffer employed in the base case. In our proposed processor model, the dynamic characterization of messages happens only in the processors and intermediate network routers cannot re-assign a message to a different set of wires. While this may have a negative effect on performance in a highly utilized network, we chose to keep the routers simple and not implement such a feature. For a network employing virtual channel flow control, each set of wires in the heterogeneous network link is treated as a separate physical channel and the same number of virtual channels are maintained per physical channel. Therefore, the heterogeneous network has a larger total number of virtual channels and the routers require more state fields to keep track of these additional virtual channels. To summarize, the additional overhead introduced by the heterogeneous model comes in the form of potentially more latches and greater routing complexity.

4.3.3.2 Overhead in Decision Process

The decision process in selecting the right set of wires is minimal. For example, in **Proposal I**, an OR function on the directory state for that block is enough to select either B- or PW-Wires. In **Proposal II**, the decision process involves a check to determine if the block is in the exclusive state. To support **Proposal III**, we need a mechanism that tracks the level of congestion in the network (for example, the number of buffered outstanding messages). There is no decision process involved for **Proposals IV**, **V**, **VI** and **VIII**. **Proposals VII** and **IX** require logic to compute the width of an operand, similar to logic used in the PowerPC 603 [52] to determine the latency for integer multiply.

4.3.3.3 Overhead in Cache Coherence Protocols

Most coherence protocols are already designed to be robust in the face of variable delays for different messages. For protocols relying on message order within a virtual channel, each virtual channel can be made to consist of a set of L-, B-, and PW-message buffers. A multiplexor can be used to activate only one type of message buffer at a time to ensure correctness. For other protocols that are designed to handle message re-ordering within a virtual channel, we propose to employ one dedicated virtual channel for each set of wires to fully exploit the benefits of a heterogeneous interconnect. In all proposed innovations, a data packet is not distributed across different sets of wires. Therefore, different components of an entity do not arrive at different periods of time, thereby eliminating any timing problems. It may be worth considering sending the critical word of a cache line on L-Wires and the rest of the cache line on PW-Wires. Such a proposal may entail nontrivial complexity to handle corner cases and is not discussed further in this chapter.

In a snooping bus-based coherence protocol, transactions are serialized by the order in which addresses appear on the bus. None of our proposed innovations for bus-based protocols affect the transmission of address bits (address bits are always transmitted on B-Wires), so the transaction serialization model is preserved.

4.4 Results

In this section we present details of our experimental methodology and results. We describe the methodology in Section 4.4.1. In Section 4.4.2 and Section 4.4.3, we report the results for scientific applications and commercial applications respectively. Finally, we present the sensitivity analysis in Section 4.4.4.

4.4.1 Methodology

4.4.1.1 Simulator

We simulate a 16-core CMP with the Virtutech Simics full-system functional execution-driven simulator [91] and a timing infrastructure GEMS [93]. GEMS

can simulate both in-order and out-of-order processors. In most studies, we use the in-order blocking processor model provided by Simics to drive the detailed memory model (Ruby) for fast simulation. Ruby implements a one-level MOESI directory cache coherence protocol with migratory sharing optimization [44, 111]. All processor cores share a noninclusive L2 cache, which is organized as a non-uniform cache architecture (NUCA) [67]. Ruby can also be driven by an out-of-order processor module called Opal, and we report the impact of the processor cores on the heterogeneous interconnect in Section 4.4.4.1. Opal is a timing-first simulator that implements the performance sensitive aspects of an out of order processor but ultimately relies on Simics to provide functional correctness. We configure Opal to model the processor described in Table 4.2 and use an aggressive implementation of sequential consistency.

To test our ideas, we employ a workload consisting of all programs from the SPLASH-2 [120] benchmark suite. The programs were run to completion, but all experimental results reported in this chapter are for the parallel phases of these applications. We use default input sets for most programs except fft and radix.

Table 4.2. System configuration.

Parameter	**Value**
number of cores	16
clock frequency	5GHz
pipeline width	4-wide fetch and issue
pipeline stages	11
cache block size	64 Bytes
split L1 I & D cache	128KB, 4-way
shared L2 cache	8MBytes
4-way, 16-banks noninclusive NUCA	30 cycles
interconnect link latency	4 cycles for 8X-B-Wires
DRAM latency	400 cycles
memory bank capacity	1 GByte per bank
latency to mem controller	100 cycles

Since the default working sets of these two programs are too small, we increase the working set of fft to 1M data points and that of radix to 4M keys.

4.4.1.2 Interconnect Power/Delay/Area Models

This section describes details of the interconnect architecture and the methodology we employ for calculating the area, delay, and power values of the interconnect. We consider 65nm process technology and assume 10 metal layers, 4 layers in 1X plane and 2 layers, in each 2X, 4X, and 8X plane [78]. For most of our study we employ a crossbar based hierarchical interconnect structure to connect the cores and L2 cache (Figure 4.3), similar to that in SGI's NUMALink-4 [1]. The effect of other interconnect topologies is discussed in our sensitivity analysis. In the base case, each link in Figure 4.3 consists of (in each direction) 64-bit address wires, 64-byte data wires, and 24-bit control wires. The control signals carry source, destination, signal type, and Miss Status Holding Register (MSHR) id. All wires are fully pipelined. Thus, each link in the interconnect is capable of transferring 75 bytes in each direction. Error Correction Codes (ECC) account for another 13% overhead in addition to the above mentioned wires [101]. All the wires of the base case are routed as B-Wires in the 8X plane.

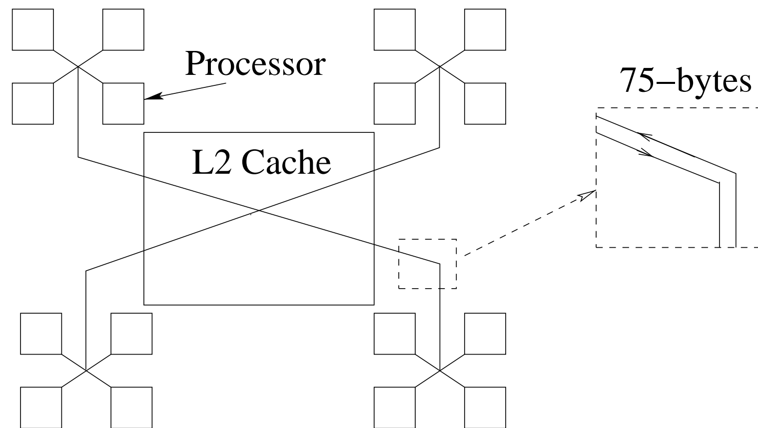

Figure 4.3. Interconnect model for a 16-core CMP

The proposed heterogeneous model employs additional wire types within each link. In addition to B-Wires, each link includes low-latency, low-bandwidth L-Wires and high-bandwidth, high-latency, power-efficient, PW-Wires. The number of L- and PW-Wires that can be employed is a function of the available metal area and the needs of the coherence protocol. In order to match the metal area with the baseline, each uni-directional link within the heterogeneous model is designed to be made up of 24 L-Wires, 512 PW-Wires, and 256 B-Wires (the base case has 600 B-Wires, not counting ECC). In a cycle, three messages may be sent, one on each of the three sets of wires. The bandwidth, delay, and power calculations for these wires are discussed subsequently.

Table 4.3 summarizes the different types of wires and their area, delay, and power characteristics. The area overhead of the interconnect can be mainly attributed to repeaters and wires. We use wire width and spacing (based on ITRS projections) to calculate the effective area for minimum-width wires in the 4X and 8X plane. L-Wires are designed to occupy four times the area of minimum-width 8X-B-Wires.

Our wire model is based on the RC models proposed in [18, 64, 100]. The delay per unit length of a wire with optimally placed repeaters is given by equation (4.1), where R_{wire} is resistance per unit length of the wire, C_{wire} is capacitance per unit length of the wire, and $FO1$ is the fan-out of one delay:

$$Latency_{wire} = 2.13\sqrt{R_{wire}C_{wire}FO1} \quad (4.1)$$

Table 4.3. Area, delay, and power characteristics of wires

Wire Type	Relative Latency	Relative Area	Dynamic Power (W/m) α = Switching Factor	Static Power W/m
B-Wire	$1x$	$1x$	2.65α	1.0246
B-Wire	$1.6x$	$0.5x$	2.9α	1.1578
L-Wire	$0.5x$	$4x$	1.46α	0.5670
PW-Wire	$3.2x$	$0.5x$	0.87α	0.3074

R_{wire} is inversely proportional to wire width, while C_{wire} depends on the following three components: (i) fringing capacitance that accounts for the capacitance between the side wall of the wire and substrate, (ii) parallel plate capacitance between the top and bottom layers of the metal that is directly proportional to the width of the metal, (iii) parallel plate capacitance between the adjacent metal wires that is inversely proportional to the spacing between the wires. The C_{wire} value for the top most metal layer at 65nm technology is given by equation (4.2) [100].

$$C_{wire} = 0.065 + 0.057W + 0.015/S(fF/\mu) \tag{4.2}$$

We derive relative delays for different types of wires by tuning width and spacing in the above equations. A variety of width and spacing values can allow L-Wires to yield a two-fold latency improvement at a four-fold area cost, relative to 8X-B-Wires. In order to reduce power consumption, we selected a wire implementation where the L-Wire's width was twice that of the minimum width and the spacing was six times as much as the minimum spacing for the 8X metal plane.

The total power consumed by a wire is the sum of three components (dynamic, leakage, and short-circuit power). Equations derived by Banerjee [18] are used to derive the power consumed by L- and B-Wires. These equations take into account optimal repeater size/spacing and wire width/spacing. PW-Wires are designed to have twice the delay of 4X-B-Wires. At 65nm technology, for a delay penalty of 100%, smaller and widely-spaced repeaters enable power reduction by 70% [18].

Crossbars, buffers, and arbiters are the major contributors for router power [118]. Table 4.4 shows the peak energy consumed by each component of the router for a single 32-byte transaction. The capacitance and energy for each of these

Table 4.4. Energy consumed by arbiters, buffers, and crossbars

Component	Energy/transaction (J)
Arbiter	6.43079e-14
Crossbar	5.32285e-12
Buffer read operation	1.23757e-12
Buffer write operation	1.73723e-12

components is based on analytical models proposed by Wang et al. [118]. We model a 5x5 matrix crossbar that employs a tristate buffer connector. As described in Section 4.3.3, buffers are modeled for each set of wires with word size corresponding to flit size.

4.4.2 Results for Scientific Benchmarks

For our simulations, we restrict ourselves to directory-based protocols. We model the effect of proposals pertaining to such a protocol: **I, III, IV, VIII, IX**. Proposal-II optimizes speculative reply messages in MESI protocols, which are not implemented within GEMS' MOESI protocol. Evaluations involving compaction of cache blocks (Proposal VII) is left as future work.

Figure 4.4 shows the execution time in cycles for SPLASH2 programs. The first bar shows the performance of the baseline organization that has one interconnect layer of 75 bytes, composed entirely of 8X-B-Wires. The second shows the

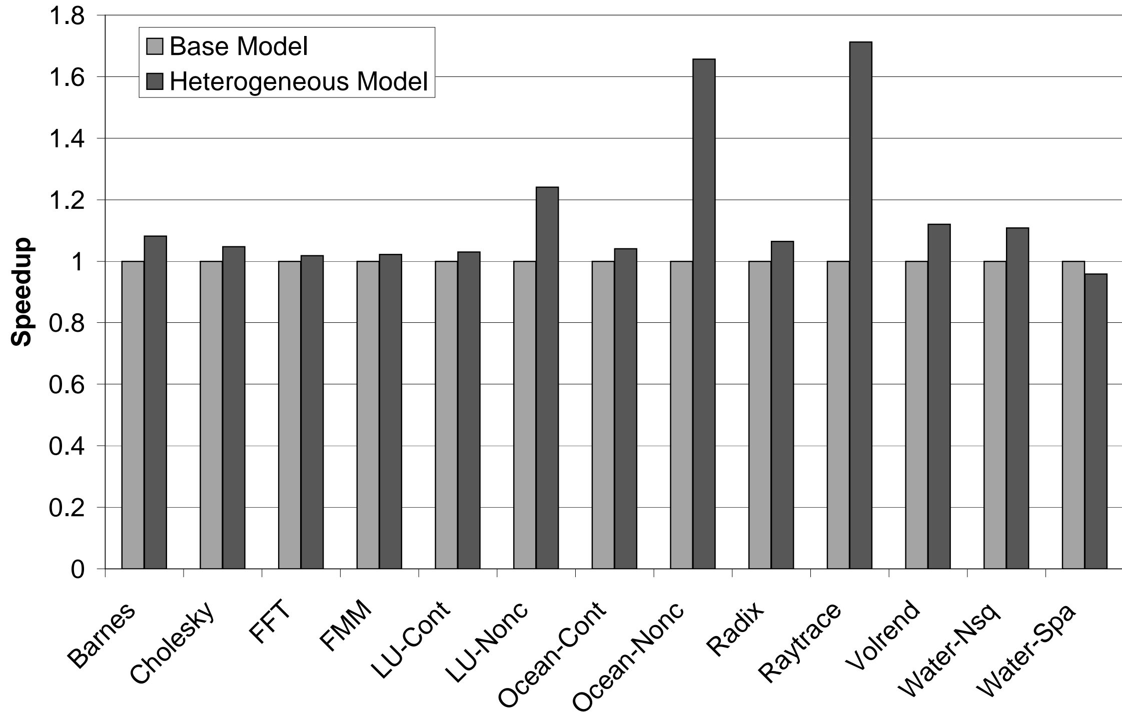

Figure 4.4. Speedup of heterogeneous interconnect

performance of the heterogeneous interconnect model in which each link consists of 24-bit L-wires, 32-byte B-wires, and 64-byte PW-wires. Programs such as LU-Noncontinuous, Ocean-Noncontinuous, and Raytracing yield significant improvements in performance. These performance numbers can be analyzed with the help of Figure 4.5 that shows the distribution of different transfers that happen on the interconnect. Transfers on L-Wires can have a huge impact on performance, provided they are on the program critical path. continuous, Ocean-Noncontinuous, Ocean-Continuous, and Raytracing experience the most transfers on L-Wires. But the performance improvement of Ocean-Continuous is very low compared to other benchmarks. This can be attributed to the fact that Ocean-Continuous incurs the most L2 cache misses and is mostly memory bound. The transfers on PW-Wires have a negligible effect on performance for all benchmarks. This is because PW-Wires are employed only for writeback transfers that are always off the critical

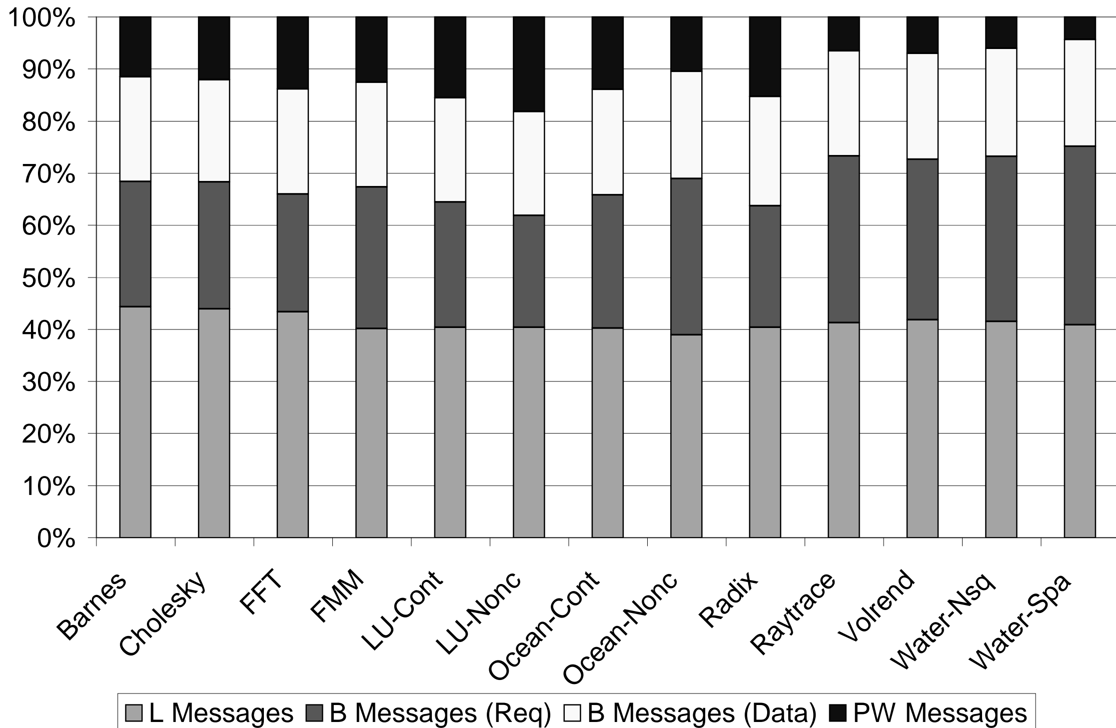

Figure 4.5. Distribution of messages on the heterogeneous network

path. On average, we observe a 11.2% improvement in performance, compared to the baseline, by employing heterogeneity within the network.

Proposals I, III, IV, and IX exploit L-Wires to send small messages within the protocol, and contribute 2.3, 0, 60.3, and 37.4 percent, respectively, to total L-Wire traffic. A per-benchmark breakdown is shown in Figure 4.6. Proposal-I optimizes the case of a read exclusive request for a block in shared state, which is not very common in the SPLASH2 benchmarks. We expect the impact of Proposal-I to be much higher in commercial workloads where cache-to-cache misses dominate. Proposal-III and Proposal-IV impact NACK, unblocking, and writecontrol messages. These messages are widely used to reduce the implementation complexity of coherence protocols. In GEMS' MOESI protocol, NACK messages are used only to handle the race condition between two write-back messages, which are negligible in our study (causing the zero contribution of Proposal-III). Instead, the protocol implementation heavily relies on unblocking and writecontrol messages to maintain

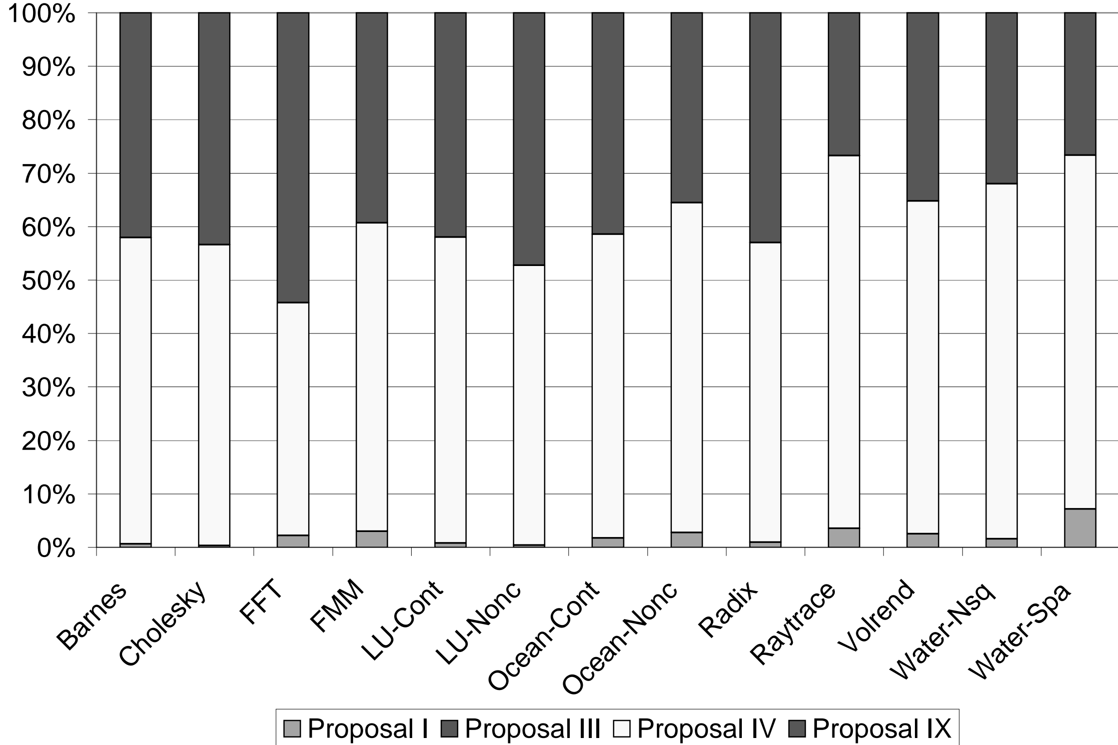

Figure 4.6. Distribution of L-message transfers across different proposals.

the order between read and write transactions, as discussed in Section 4.3.1. The frequency of occurrence of NACK, unblocking, and writecontrol messages depends on the protocol implementation, but we expect the sum of these messages to be relatively constant in different protocols and play an important role in L-wire optimizations. Proposal-IX includes all other acknowledgment messages eligible for transfer on L-Wires.

We observed that the combination of proposals I, III, IV, and IX caused a performance improvement more than the sum of improvements from each individual proposal. A parallel benchmark can be divided into a number of phases by synchronization variables (barriers), and the execution time of each phase can be defined as the longest time any thread spends from one barrier to the next. Optimizations applied to a single thread may have no effect if there are other threads on the critical path. However, a different optimization may apply to the threads on the critical path, reduce their execution time, and expose the performance of other threads and the optimizations that apply to them. Since different threads take different data paths, most parallel applications show nontrivial workload imbalance [87]. Therefore, employing one proposal might not speedup all threads on the critical path, but employing all applicable proposals can probably optimize threads on every path, thereby reducing the total barrier to barrier time.

Figure 4.7 shows the improvement in network energy due to the heterogeneous interconnect model. The first bar shows the reduction in network energy and the second bar shows the improvement in the overall processor $Energy \times Delay^2$ (ED^2) metric. Other metrics in the $E - D$ space can also be computed with data in Figure 4.4 and Figure 4.7. To calculate ED^2, we assume that the total power consumption of the chip is 200W, of which the network power accounts for 60W. The energy improvement in the heterogeneous case comes from both L and PW transfers. Many control messages that are sent on B-Wires in the base case are sent on L-Wires in the heterogeneous case. As shown in Table 4.3, the energy consumed by an L-Wire is less than the energy consumed by a B-Wire. But due to the small sizes of these messages, the contribution of L-messages to the total energy savings

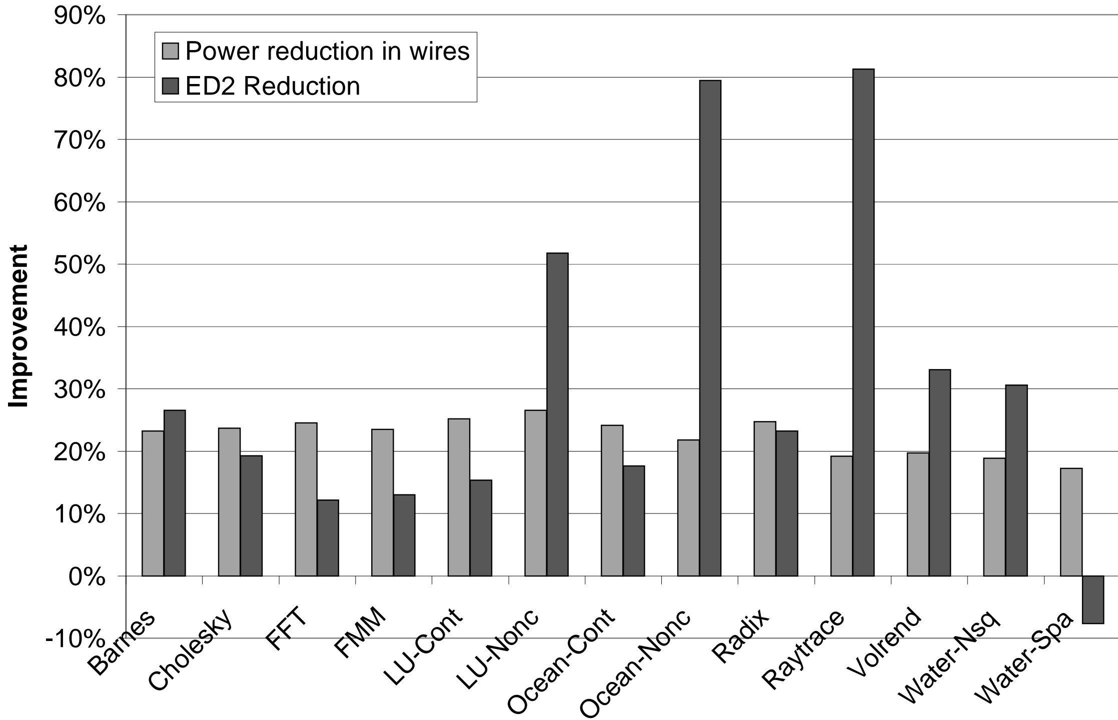

Figure 4.7. Improvement in link energy and ED^2.

is negligible. Overall, the heterogeneous network results in a 22% saving in network energy and a 30% improvement in ED^2.

4.4.3 Results for Commercial Benchmarks

Our commercial benchmarks consist of three workloads: a decision support benchmark (TPC-H), a static web serving workload (Apache), and a dynamic web serving workload (SPECweb). These workloads execute on a simulated 16-processor SPARC multiprocessor running Suse Linux 7.3. The simulated system has 4GBs of main memory.

- **Decision support:** TPC-H is a decision support benchmark released by the Transaction Processing Council. TPC-H consists of 8 tables, 22 read-only queries and 2 batch update statements, which simulate the activities of a wholesale supplier. We use MYSQL 3.23 as the database management systems with the total table size around 500 MB. We select nine queries from the TPC-

H DSS workload based on the categorization in [108]: seven scan-dominated queries (Q1, Q3, Q5, Q6, Q12, Q14, Q19), one join-dominated query (Q16), and one query exhibiting mixed behavior (Q10).

- **Static Web Content Serving:** Web servers such as Apache are an important enterprise server application. We use Apache 1.3.20-77 configured to use pthread locks and minimal logging as the web server. The web requests are generated by using SURGE [20, 10]. We use a repository of 10,000 files (totalling 300 MB), and disable Apache logging for high performance. We simulate 160 clients each with zero thinking time between requests. The system is warmed by half million requests.

- **Dynamic Web Content Serving:** SPECweb99 is a SPEC benchmark for evaluating a system's ability to act as a web server for static and dynamic pages. The benchmark runs a multithreaded HTTP load generator on a number of driving client systems that perform static, dynamic GETs and POSTs on a variety of pages. The default transaction mix consists of 70% static GETs, 12.45% standard dynamic GETs, 12.6% dynamic GETs with custom ad-rotation, 4.8% dynamic POSTs and 0.15% dynamic GETs calling CGI code. We simulate 50 simultaneous connections with a repository of 300 MB and warmup the system for 1200 seconds.

In this study, we employ the same methodology described in Section 4.4.1. We model the effect of proposals pertaining to the MOESI directory-based protocol: **I, III, IV, VIII, IX**.

Figure 4.8 shows the normalized execution time for commercial programs. The first bar shows the performance of the baseline organization that has one interconnect layer of 75 bytes, composed entirely of 8X-B-Wires. The second shows the performance of the heterogeneous interconnect model in which each link consists of 24-bit L-wires, 32-byte B-wires, and 64-byte PW-wires. The performance improvement of heterogeneous interconnect on scan-dominated TPC-H queries varies from 15% (Q3) to 26% (Q1, Q12, Q14). The join-bound query (Q10) and the

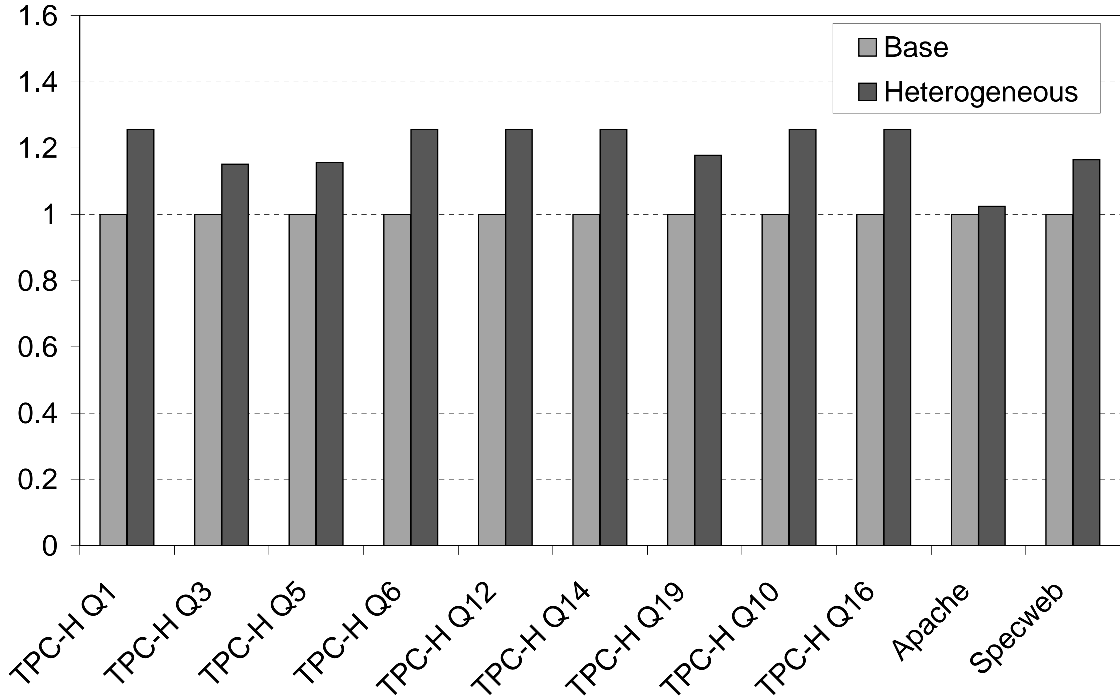

Figure 4.8. Speedup in commercial applications.

mixed behavior query (Q19) both yield 16% improvements in performance. Cain et al. report that TPC-H has a significant number of coherence misses due to false sharings [31]. These coherence misses can benefit from proposal **I** and proposal **IV**. SPECweb99 also shows significant number of false sharings [66], which leads to good performance improvement (17%). But the performance improvement of Apache is very low (2%) compared with other applications. This can be attributed to the fact that Apache incurs lots of L2 cache misses and is mostly memory bound.

Figure 4.9 shows the reduction in network energy due to the heterogeneous interconnect model. The energy improvement in the heterogeneous case comes form both L and PW transfers, though the contribution from L wires is negligible. Unlike the variations in the performance improvement plot, the power reduction in all applications are very predictable. Overall, the heterogeneous network results in a average 25% saving in network energy.

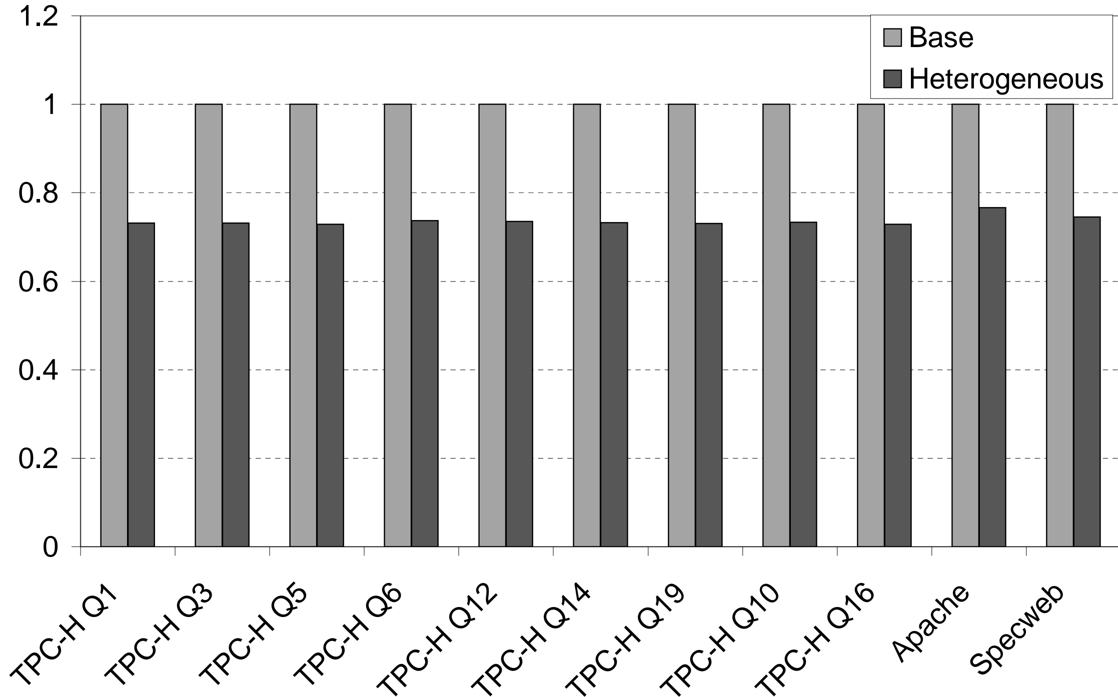

Figure 4.9. Power reduction in commercial applications.

4.4.4 Sensitivity Analysis

In this subsection, we discuss the impact of processor cores, link bandwidth, routing algorithm, and network topology on the heterogeneous interconnect. Due to the negative impact of workload variability in commercial workloads [11], we confine our sensitivity analysis to SPLASH2 applications.

4.4.4.1 Out-of-order/In-order Processors

To test our ideas with an out-of-order processor, we configure Opal to model the processor described in Table 4.2 and only report the results of the first 100M instructions in the parallel sections.[2]

Figure 4.10 shows the performance speedup of the heterogeneous interconnect over the baseline. All benchmarks except Ocean-Noncontinuous demonstrate dif-

[2]Simulating the entire program takes nearly a week and there exist no effective toolkits to find the representative phases for parallel benchmarks. LU-Noncontinuous and Radix were not compatible with the Opal timing module.

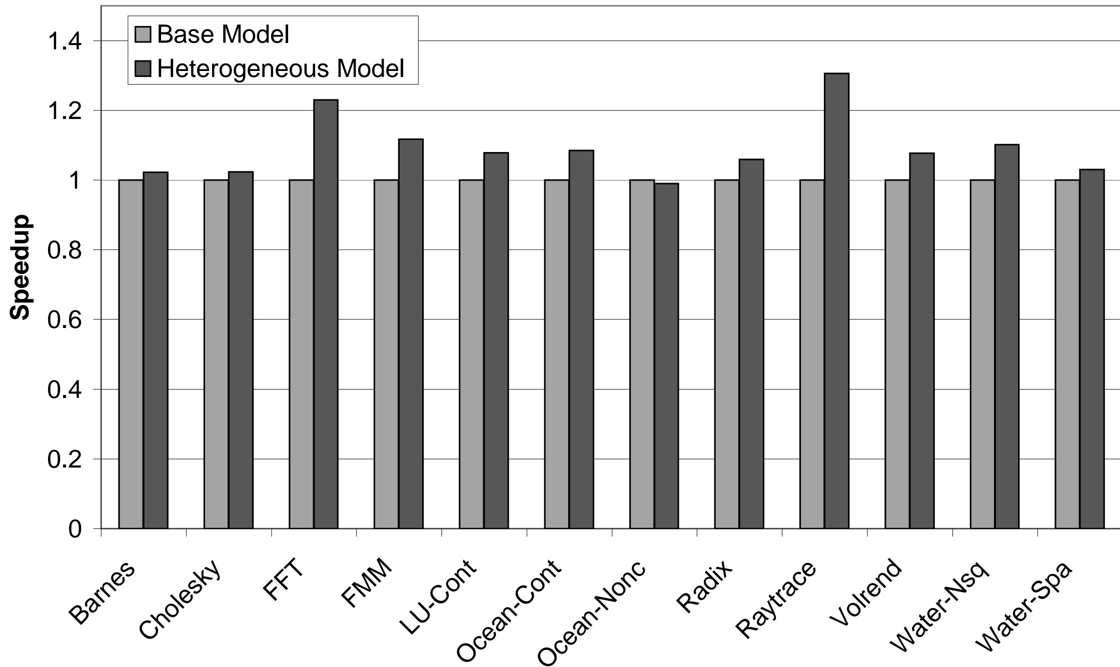

Figure 4.10. Speedup of heterogeneous interconnect when driven by OoO cores

ferent degrees of performance improvement, which leads to an average speedup of 9.3%. The average performance improvement is less than what we observe in a system employing in-order cores (11.2%). This can be attributed to the greater tolerance that an out-of-order processor has to long instruction latencies.

4.4.4.2 Link Bandwidth

The heterogeneous network poses more constraints on the type of messages that can be issued by a processor in a cycle. It is therefore likely to not perform very well in a bandwidth-constrained system. To verify this, we modeled a base case where every link has only 80 8X-B-Wires and a heterogeneous case where every link is composed of 24 L-Wires, 24 8X-B-Wires, and 48 PW-Wires (almost twice the metal area of the new base case). Benchmarks with higher network utilizations suffered significant performance losses. In our experiments raytracing has the maximum messages/cycle ratio and the heterogeneous case suffered a 27% performance loss, compared to the base case (in spite of having twice the metal

area). The heterogeneous interconnect performance improvement for Ocean Noncontinuous and LU Noncontinuous is 12% and 11%, as against 39% and 20% in the high-bandwidth simulations. Overall, the heterogeneous model performed 1.5% worse than the base case.

4.4.4.3 Routing Algorithm

Our simulations thus far have employed adaptive routing within the network. Adaptive routing alleviates the contention problem by dynamically routing messages based on the network traffic. We found that deterministic routing degraded performance by about 3% for most programs for systems with the baseline and with the heterogeneous network. Raytracing is the only benchmark that incurs a significant performance penalty of 27% for both networks.

4.4.4.4 Network Topology

Our default interconnect thus far was a two-level tree based on SGI's NUMALink-4 [1]. To test the sensitivity of our results to the network topology, we also examine a 2D-torus interconnect resembling that in the Alpha 21364 [19]. As shown in Figure 4.11, each router connects to four links that connect to 4 neighbors in the torus, and wraparound links are employed to connect routers on the boundary.

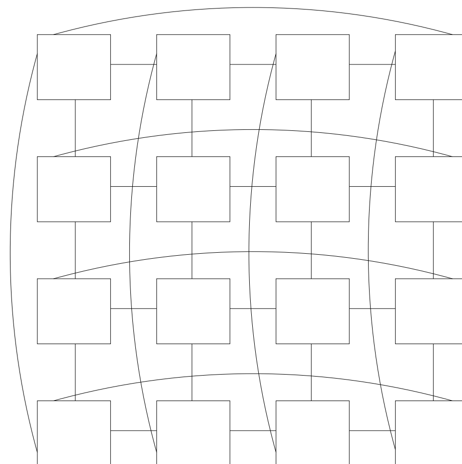

Figure 4.11. 2D torus topology.

Our proposed mechanisms show much less performance benefit (1.3% on average) in the 2D torus interconnect than in the two-level tree interconnect (Figure 4.12). The main reason is that our decision process in selecting the right set of wires calculates hop imbalance at the coherence protocol level without considering the physical hops a message takes on the mapped topology. For example, in a 3-hop transaction as shown in Figure 4.2, the 1-hop message may take four physical hops while the 2-hop message may also take four physical hops. In this case, sending the 2-hop message on the L-Wires and the 1-hop message on the PW-Wires will actually lower performance.

This is not a first-order effect in the two-level tree interconnect, where most hops take four physical hops. However, the average distance between two processors in the 2D torus interconnect is 2.13 physical hops with a standard deviation of 0.92 hops. In an interconnect with such high standard deviation, calculating hop imbalance based on protocol hops is inaccurate. For future work, we plan to

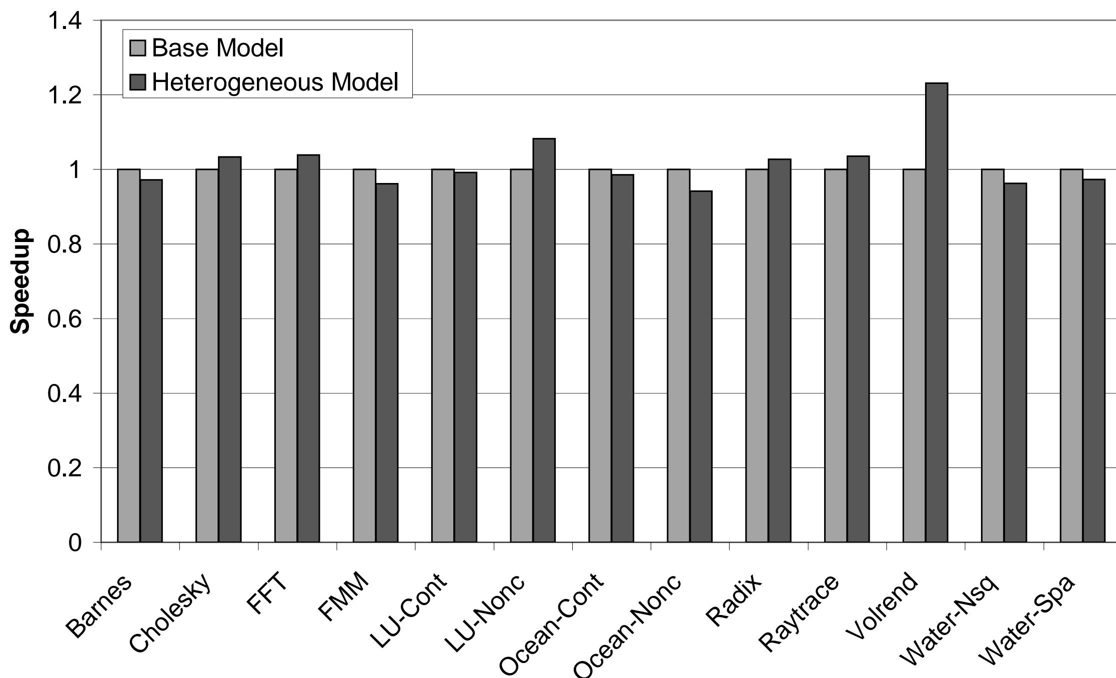

Figure 4.12. Heterogeneous interconnect speedup in 2D torus.

develop a more accurate decision process that considers source id, destination id, and interconnect topology to dynamically compute an optimal mapping to wires.

4.5 Summary

Coherence traffic in a chip multiprocessor has diverse needs. Some messages can tolerate long latencies, while others are on the program critical path. Further, messages have varied bandwidth demands. On-chip global wires can be designed to optimize latency, bandwidth, or power. We advocate partitioning available metal area across different wire implementations and intelligently mapping data to the set of wires best suited for its communication. This chapter presents numerous novel techniques that can exploit a heterogeneous interconnect to simultaneously improve performance and reduce power consumption.

Our evaluation of a subset of the proposed techniques shows that a large fraction of messages have low bandwidth needs and can be transmitted on low latency wires, thereby yielding a performance improvement of 11.2%. At the same time, a 22.5% reduction in interconnect energy is observed by transmitting noncritical data on power-efficient wires. The complexity cost is marginal as the mapping of messages to wires entails simple logic.

For future work, we plan to strengthen our decision process in calculating the hop imbalance based on the topology of the interconnect. We will also evaluate the potential of other techniques listed in this chapter. There may be several other applications of heterogeneous interconnects within a CMP. For example, in the *Dynamic Self Invalidation* scheme proposed by Lebeck et al. [84], the self-invalidate [81, 84] messages can be effected through power-efficient PW-Wires. In a processor model implementing token coherence, the low-bandwidth token messages [95] are often on the critical path and thus, can be effected on L-Wires. A recent study by Huh et al. [66] reduces the frequency of false sharing by employing incoherent data. For cache lines suffering from false sharing, only the sharing states need to be propagated and such messages are a good match for low-bandwidth L-Wires.

CHAPTER 5

CONCLUSIONS

In this dissertation, we propose *context-aware coherence protocols* that allow the coherence protocols to tune coherence traffic based on the hints given by the contexts. Compared with traditional protocols, context-aware coherence protocols can improve performance and reduce power consumption.

We first propose a cache coherence protocol which is aware of the producer-consumer sharing and propose two novel mechanisms, *directory delegation* and *speculative updates*, that can be used to improve the performance of applications that exhibit producer-consumer sharing. After detecting instances of producer-consumer sharing using a simple directory-based predictor, we delegate responsibility for the data's directory information from its home node to the current producer of the data, which can convert 3-hop coherence operations into 2-hop operations. We also present a speculative update mechanism wherein shortly after modifying a particular piece of data, producers speculatively forward updates to the nodes that most recently accessed it.

On a collection of seven benchmark programs, we demonstrate that speculative updates can significantly reduce the number of remote misses suffered and amount of network traffic generated. We consider two hardware implementations, one that requires very little hardware overhead (a 32-entry delegate cache and a 32KB RAC per node) and one that requires modest overhead (a 1K-entry delegate cache and a 1MB RAC per node). On the small configuration, delegation/updates reduce execution time by 13% by reducing the number of remote misses by 29% and network traffic by 17%. On the larger configuration, delegation/updates reduce program execution time by 21% by reducing the number of remote misses by

40% and network traffic by 15%. Finally, we show that the performance benefits derive primarily from eliminating remote misses, and only secondarily from reducing network traffic.

Furthermore, we propose coherence protocols that are aware of the availability of heterogeneous interconnects. Coherence traffic in a chip multiprocessor has diverse needs. Some messages can tolerate long latencies, while others are on the program critical path. Similarly, messages have varied bandwidth demands. On-chip global wires can be designed to optimize latency, bandwidth, or power. We advocate partitioning available metal area across different wire implementations and intelligently mapping data to the set of wires best suited for its communication. We present numerous novel techniques that can exploit a heterogeneous interconnect to simultaneously improve performance and reduce power consumption.

Our evaluation of a subset of the proposed techniques shows that a large fraction of messages have low bandwidth needs and can be transmitted on low latency wires, thereby yielding a performance improvement of 11.2%. At the same time, a 22.5% reduction in interconnect energy is observed by transmitting noncritical data on power-efficient wires. The complexity cost is marginal as the mapping of messages to wires entails simple logic.

Overall, we have conducted some of the first research on the context-aware cache coherence protocols. We have demonstrated the benefits of context-aware cache protocols in the context of "producer-consumer sharing" and "heterogeneous interconnect."

REFERENCES

[1] "http://www.sgi.com/products/servers/altix/configs.html".

[2] "http://www.top500.org/".

[3] "http://phase.hpcc.jp/Omni/benchmarks/NPB/".

[4] 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 3rd International Symposium on High Performance Computer Architecture*, 1997.

[5] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles. Towards a Better Understanding of Context and Context-Awareness. In *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing*, pages 304–307, 1999.

[6] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato. The Use of Prediction for Accelerating Upgrade Misses in CC-NUMA Multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques*, 2002.

[7] S. Adve and M. Hill. Weak Ordering: A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, 1990.

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

[9] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock Rate Versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, 2000.

[10] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, M. D. Hill, D. A. Wood, and D. J. Sorin. Simulating a $2M Commercial Server on a $2K PC. *Computer*, 36(2):50–57, 2003.

[11] A. R. Alameldeen and D. A. Wood. Variability in Architectural Simulations of Multi-Threaded Workloads. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, page 7, 2003.

[12] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Addison-Wesley, 1988. ISBN 0-8053-0177-1.

[13] E. Anderson, J. Brooks, C. Grassl, and S. Scott. Performance of the CRAY T3E Multiprocessor. In *Proceedings of the 11th Annual International Conference on Supercomputing*, pages 1–17, 1997.

[14] 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 Symposium on High Performance Interconnects HOT Interconnects*, page 65, 2002.

[15] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1994.

[16] H. Bakoglu. *Circuits, Interconnections, and Packaging for VLSI*. Addison-Wesley, 1990.

[17] R. Balasubramonian, N. Muralimanohar, K. Ramani, and V. Venkatachalapathy. Microarchitectural Wire Management for Performance and Power in Partitioned Architectures. In *Proceedings of the 11th International Symposium of High Performance Computer Architecture*, 2005.

[18] K. Banerjee and A. Mehrotra. A Power-optimal Repeater Insertion Methodology for Global Interconnects in Nanometer Designs. *IEEE Transactions on Electron Devices*, 49(11):2001–2007, 2002.

[19] P. Bannon. Alpha 21364: A Scalable Single-Chip SMP. *Microprocessor Forum*, October 1998.

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

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

[22] L. A. Barroso and M. Dubois. The Performance of Cache-Coherent Ring-based Multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 268–277, 1993.

[23] B. Beckmann and D. Wood. TLC: Transmission Line Caches. In *Proceedings of MICRO-36*, December 2003.

[24] B. Beckmann and D. Wood. Managing Wire Delay in Large Chip-Multiprocessor Caches. In *Proceedings of MICRO-37*, 2004.

[25] 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. *SIGARCH Compututer Architecture News*, pages 294–304, 1999.

[26] F. Briggs, M. Cekleov, K. Creta, M. Khare, S. Kulick, A. Kumar, L. P. Looi, C. Natarajan, S. Radhakrishnan, and L. Rankin. Intel 870: A Building Block for Cost-Effective, Scalable Servers. *IEEE Micro*, 22(2):36–47, 2002.

[27] D. M. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P. W. Cook. Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors. *IEEE Micro*, 20(6):26–44, 2000.

[28] D. Burger and J. R. Goodman. Billion Transistor Architectures. *IEEE Computer*, (9):46–50, 1997.

[29] D. Burger and J. R. Goodman. Billion-Transistor Architectures: There and Back Again. *IEEE Computer*, 37(3):22–28, 2004.

[30] G. Byrd and M. Flynn. Producer-consumer Communication in Distributed Shared Memory Multiprocessors. In *Proceedings of the IEEE, Vol.87, Iss.3*, 1999.

[31] H. W. Cain and M. H. Lipasti. Memory Ordering: A Value-Based Approach. *IEEE Micro*, 24(6):110–117, 2004.

[32] J. Carter, J. Bennett, and W. Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems. *ACM Transactions on Computer Systems*, 13(3):205–243, 1995.

[33] M. Cekleov, D. Yen, P. Sindhu, and J.-M. Frailong. SPARCcenter 2000: Multiprocessing for the 90's, Digest of Papers. In *IEEE Computer Society Press*, pages 345–53, 1993.

[34] R. Chang, N. Talwalkar, C. Yue, and S. Wong. Near Speed-of-Light Signaling Over On-Chip Electrical Interconnects. *IEEE Journal of Solid-State Circuits*, 38(5):834–838, 2003.

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

[36] A. Charlesworth. The Sun Fireplane SMP Interconnect in the Sun Fire 3800-6800. In *Proceedings of the 9th Symposium on High Performance Interconnects*, page 37, 2001.

[37] A. Charlesworth. The Sun Fireplane Interconnect. *IEEE Micro*, 22(1):36–45, 2002.

[38] X. Chen, Y. Yang, G. Gopalakrishnan, and C.-T. Chou. Reducing Verification Complexity of a Multicore Coherence Protocol Using Assume/Guarantee. In *Proceedings of FMCAD*, pages 81–88, 2006.

[39] L. Cheng, N. Muralimanohar, K. Ramani, R. Balasubramonian, and J. B. Carter. Interconnect-Aware Coherence Protocols for Chip Multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 339–351, 2006.

[40] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing Replication, Communication, and Capacity Allocation in CMPs. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 357–368, 2005.

[41] C.-T. Chou, P. K. Mannava, and S. Park. A Simple Method for Parameterized Verification of Cache Coherence Protocols. In *Proceedings of FMCAD*, pages 382–398, 2004.

[42] N. Chrisochoides, I. Kodukula, and K. Pingali. Compiler and Run-Time Support for Semi-structured Applications. In *Proceedings of the 11th Annual International Conference on Supercomputing*, pages 229–236, 1997.

[43] D. Citron. Exploiting Low Entropy to Reduce Wire Delay. *IEEE Computer Architecture Letters*, vol.2, 2004.

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

[45] Cray Research, Inc. *CRAY T3D System Architecture Overview*, Cray Research HR-04033 edition, 1993.

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

[47] D. L. Dill. The Murphi Verification System. In *Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102, pages 390–393, 1996.

[48] W. Dally and J. Poulton. *Digital System Engineering*. Cambridge University Press, Cambridge, UK, 1998.

[49] J. Duato, S. Yalamanchili, and N. Lionel. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers Inc., 2002.

[50] B. Falsafi, A. Lebeck, S. Reinhardt, I. Schoinas, M. Hill, J. Larus, A. Rogers, and D. Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of the 8th Annual International Conference on Supercomputing*, pages 380–389, 1994.

[51] M. Galles and E. Williams. Performance Optimizations, Implementation, and Verification of the SGI Challenge Multiprocessor. In *Proceeding of the 1st International Conference on System Science*, pages 134–143, 1994.

[52] G. Gerosa and et al. A 2.2 W, 80 MHz Superscalar RISC Microprocessor. *IEEE Journal of Solid-State Circuits*, 29(12):1440–1454, 1994.

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

[54] K. Gharachorloo, M. Sharma, S. Steely, and S. V. Doren. Architecture and design of AlphaServer GS320. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 13–24, 2000.

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

[56] G. Gostin, J.-F. Collard, and K. Collins. The Architecture of the HP Superdome Shared-memory Multiprocessor. In *Proceedings of the 19th Annual International Conference on Supercomputing*, pages 239–245, 2005.

[57] M. Gowan, L. Biro, and D. Jackson. Power Considerations in the Design of the Alpha 21264 Microprocessor. In *Proceedings of the 35th Design Automation Conference*, 1998.

[58] S. L. Graham, M. Snir, and C. A. Patterson. Getting Up to Speed: The Future of Supercomputing. In *Committee on the Future of Supercomputing, National Research Council*, 2004.

[59] H. Grahn, P. Stenström, and M. Dubois. Implementation and Evaluation of Update-based Cache Protocols under Relaxed Memory Consistency Models. *Future Generation Computer Systems*, 11(3):247–271, 1995.

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

[61] A. Gupta, W. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *Proceedings of the 19th International Conference on Parallel Processing*, volume I, pages 312–321, 1990.

[62] M. Heinrich and J. K. et al. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 274–285, 1994.

[63] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2002.

[64] R. Ho, K. Mai, and M. Horowitz. The Future of Wires. *Proceedings of the IEEE*, Vol.89, No.4, 2001.

[65] H. P. Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture*, pages 258–262, 2005.

[66] J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence Decoupling: Making Use of Incoherence. In *Proceedings of the 11th Symposium on Architecture Support for Programming Languages and Operating Systems*, pages 97–106, 2004.

[67] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A NUCA Substrate for Flexible CMP Cache Sharing. In *Proceedings of the 19th Annual International Conference on Supercomputing*, pages 31–40, 2005.

[68] Institute of Electrical and Electronics Engineers. *IEEE Standard for Scalable Coherent Interface: IEEE Std. 1596-1992*. 1993.

[69] D. James. Distributed Directory Scheme: Scalable Coherent Interface. *IEEE Computer*, 23(6):74–77, 1990.

[70] T. Joe and J. Hennessy. Evaluating the Memory Overhead Required for COMA Architectures. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 82–93, 1994.

[71] S. Kaxiras and J. R. Goodman. Improving CC-NUMA Performance Using Instruction-Based Prediction. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, 1999.

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

[73] P. Kongetira. A 32-Way Multithreaded SPARC Processor. In *Proceedings of Hot Chips 16*, 2004.

[74] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, 2005.

[75] D. A. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas. Data Forwarding in Scalable Shared-memory Multiprocessors. In *Proceedings of the 9th Annual International Conference on Supercomputing*, 1995.

[76] K. Krewell. UltraSPARC IV Mirrors Predecessor: Sun Builds Dualcore Chip in 130nm. *Microprocessor Report*, pages 1,5–6, 2003.

[77] J. Kubiatowicz and A. Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *Proceedings of the 7th Annual International Conference on Supercomputing*, 1993.

[78] R. Kumar, V. Zyuban, and D. Tullsen. Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads, and Scaling. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005.

[79] C.-C. Kuo, J. Carter, R. Kuramkote, and M. Swanson. AS-COMA: An Adaptive Hybrid Shared Memory Architecture. In *Proceedings of the 27th International Conference on Parallel Processing*, 1998.

[80] 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*, 1999.

[81] 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, 2000.

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

[83] 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, 1997.

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

[85] 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, 1990.

[86] K. M. Lepak and M. H. Lipasti. Temporally Silent Stores. In *Proceedings of the 10th Symposium on Architecture Support for Programming Languages and Operating Systems*, pages 30–41, 2002.

[87] J. Li, J. F. Martinez, and M. C. Huang. The Thrifty Barrier: Energy-Aware Synchronization in Shared-Memory Multiprocessors. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, page 14, 2004.

[88] T. Li and L. K. John. ADirpNB: A Cost-Effective Way to Implement Full Map Directory-Based Cache Coherence Protocols. *IEEE Transactions on Computers*, 50(9):921–934, 2001.

[89] T. Lovett and R. Clapp. STiNG: A CC-NUMA Compute System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308–317, 1996.

[90] N. Magen, A. Kolodny, U. Weiser, and N. Shamir. Interconnect Power Dissipation in a Microprocessor. In *Proceedings of System Level Interconnect Prediction*, 2004.

[91] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, 2002.

[92] M. Martin, P. Harper, D. Sorin, M. Hill, and D. 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*, 2003.

[93] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, 2005.

[94] M. M. Martin, M. D. Hill, and D. A. Wood. Token Coherence: A New Framework for Shared-Memory Multiprocessors. *IEEE Micro*, 23(6):108–116, 2003.

[95] 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*, 2003.

[96] 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 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–36, 2000.

[97] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. K. Martin, and D. A. Wood. Improving Multiple-CMP Systems Using Token Coherence. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture*, pages 328–339, 2005.

[98] J. R. Mashey. NUMAflex Modular Design Approach: A Revolution in Evolution. In *Computer Architecture News Group*, 2000.

[99] K. L. McMillan. Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking. In *Proceedings of the 11th Conference on Correct Hardware Design and Verification Methods*, pages 179–195, 2001.

[100] M. L. Mui, K. Banerjee, and A. Mehrotra. A Global Interconnect Optimization Scheme for Nanometer Scale VLSI With Implications for Latency, Bandwidth, and Power Dissipation. *IEEE Transactions on Electronic Devices*, Vol.51, No.2, 2004.

[101] S. Mukherjee, J. Emer, and S. Reinhardt. The Soft Error Problem: An Architectural Perspective. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture*, 2005.

[102] S. S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb. The Alpha 21364 Network Architecture. *IEEE Micro*, 22(1):26–35.

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

[104] N. Nelson, G. Briggs, M. Haurylau, G. Chen, H. Chen, D. Albonesi, E. Friedman, and P. Fauchet. Alleviating Thermal Constraints while Maintaining Performance Via Silicon-Based On-Chip Optical Interconnects. In *Proceedings of Workshop on Unique Chips and Systems*, 2005.

[105] K. J. Nesbit and J. E. Smith. Data Cache Prefetching Using a Global History Buffer. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, pages 96–105, 2004.

[106] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishin. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the 24th International Conference on Parallel Processing*, 1995.

[107] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An Argument for Simple COMA. In *Proceedings of the 1st Annual Symposium on High Performance Computer Architecture*, pages 276–285, 1995.

[108] M. Shao, A. Ailamaki, and B. Falsafi. DBmbench: Fast and Accurate Database Workload Representation on Modern Microarchitecture. In *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 254–267, 2005.

[109] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yuan, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvey, E. Hagersten, and B. Liencres. Gigaplane: A High Performance Bus for Large SMPs. In *Proceedings of Hot Interconnects IV*, pages 41–52, 1996.

[110] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial Memory Streaming. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 252–263, 2006.

[111] P. Stenström, M. Brorsson, and L. Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, 1993.

[112] K. Strauss, X. Shen, and J. Torrellas. Flexible Snooping: Adaptive Forwarding and Filtering of Snoops in Embedded-Ring Multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 327–338, 2006.

[113] 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, 1986.

[114] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. *IBM Server Group Whitepaper*, 2001.

[115] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez. Reducing Power in High-Performance Microprocessors. In *Proceedings of the 35th Design Automation Conference*, 1998.

[116] S.-Y. Tzou and D. Anderson. The Performance of Message Passing using Restricted Virtual Memory Mapping. *Software, Practice & Experience*, 21(3):251–267, 1991.

[117] Z. Vranesic, M. Stumm, D. Lewis, and R. White. Hector: A Hierarchically Structured Shared Memory Multiprocessor. *IEEE Computer*, 24(1):72–79, 1991.

[118] H. S. Wang, L. S. Peh, and S. Malik. A Power Model for Routers: Modeling Alpha 21364 and InfiniBand Routers. In *IEEE Micro, Vol 24, No 1*, 2003.

[119] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal Streaming of Shared Memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 222–233, 2005.

[120] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, pages 24–36, 1995.