© 2007 by Karin Strauss. All rights reserved.

CACHE COHERENCE IN EMBEDDED-RING MULTIPROCESSORS

BY

KARIN STRAUSS

B.Eng., Universidade de São Paulo, 2001
M.Eng., Universidade de São Paulo, 2002

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois

Abstract

Design complexity and limited power budget are causing the number of cores on the same chip to grow very rapidly. The wide availability of Chip Multiprocessors (CMPs) is enabling the design of inexpensive, shared-memory machines of medium size (32-128 cores). However, for machines of this size, none of the two traditional approaches to support cache coherence seems optimal. Snoopy schemes implemented with broadcast buses are difficult to efficiently scale beyond 8-32 cores. Directory-based schemes have the cost of maintaining a directory structure, as well as the fundamental latency disadvantage of adding at least one level of indirection to coherence transactions.

In this work, we propose to *logically embed* a ring in a point-to-point network topology. Snoop messages use the logical ring, while other messages can use any link in the network. The resulting design is simple and low cost. Perhaps the main drawback of the embedded-ring approach is that snoop requests may suffer long latencies or induce many snoop messages and operations. In this work, we address these issues and, as a result, provide simple and competitive cache coherence protocol designs.

To Luis, without whom I wouldn't have made it.

Acknowledgments

I would like to thank all the wonderful mentors and colleagues with whom I had the privilege to interact during my studies.

First of all, I would like to thank Prof. Josep Torrellas, who offered me a research assistantship from the beginning and supported me financially during the PhD program. Even more important, Josep taught me how to do solid research, how to write consistent papers, and how to give good presentations, among many other things.

I would also like to thank Dr. Xiaowei Shen, who was practically my co-advisor and who has accepted to be part of my committee, even though he is currently in a very busy assignment in China. I worked with Xiaowei on my third and fourth summer co-op assignments at IBM, and remotely during the last three years of my PhD. He taught me a lot about cache coherence. We discussed many multiprocessor issues, and those discussions inspired most of what is in this thesis, and other things that I should not mention here, well, for obvious reasons.

I am very grateful to the members of my PhD committee. Prof. Marc Snir, Prof. Sarita Adve, Prof. Sanjay Patel and Prof. Wen-Mei Hwu, together with Xiaowei and Josep, provided me invaluable feedback on my work and helped me make it more formal.

I would like to thank Intel and IBM for supporting me during my PhD program with PhD Fellowships. I also had great mentors at IBM, both during summers, while working for IBM in Yorktown Heights, and throughout the school year, while working on my studies back in Urbana-Champaign.

There are also a number of other people from University of Illinois at Urbana-Champaign

I would like to thank. I would like to start with my I-ACOMA group colleagues, José Renau, Jun Nakano, James Tuck, Luis Ceze, Radu Teodorescu, Smruti R. Sarangi, Paul Sack, Pablo Montesinos, Brian Greskamp, Abhishek Tiwari, Wonsun Ahn and Abdulla Muzahid and Wei Liu, for attending my talks and giving me suggestions, especially Paul, Brian, Luis and Pablo, who also had the patience to review my papers. I would also like to thank many of them for contributing to the development of SESC, our simulator, especially José Renau, who started this effort. I am also thankful to Pierre Salverda, Naveen Neelakantam, Lee Baugh, Alex Li and Nicholas Riley for attending practice talks and discussing my work with me. Pierre Salverda, Jerry Li and Luis Ceze made a wonderful study group for the qualifying exam. We leaned a lot about vector processors and architecture in general. We were also very good at making each other read more and more papers as the exam approached. I had a number of great instructors, including Prof. Vikram Adve, Prof. Craig Zilles and Prof. Sanjay Patel. Sheila Clark, our Research Information Specialist, is the most efficient person I have ever met. She helped me with a variety of things throughout my PhD program.

Next, I would like to thank Dr. José Moreira. He is the main reason I ended up pursuing a PhD. We met on a visit a few friends and I made during our last Engineering School vacations to the IBM T. J. Watson Research Center, which was fantastic. By the end of that year, when he visited University of São Paulo, José mentioned the summer co-op program at IBM. Luis Ceze and I expressed interest in that. In a couple of months, we got offers from IBM. José was our mentor and manager during our first co-op at IBM. He did it so well that, besides convincing us to apply to PhD programs, we kept going back to IBM during our PhDs almost every summer. His help and support have been crucial to our success. During that first year at IBM, we had the opportunity to meet and interact with extremely smart and competent people. Dr. Călin Caşcaval was the first person we worked with and, with him, we learned to "read the manual", among many other extremely useful things. Dr. José G. Castaños and Dr. George Almasi coached us for the GRE and taught us to write (good) software. Dr. Siddhartha Chatterjee instigated our curiosity about compilers. We had the

pleasure to interact with many other people whose names I will not list, but they know they are special to me.

At the Polytechnic School, in Brazil, I would like to thank Prof. Jaime Sichman, who gave me my first assistantship, Prof. André Hirakawa, who taught wonderful classes and made me want to learn more, Prof. Wilson Ruggiero, whose eyes shone when he talked about computers, who hired me as a research assistant in his lab (LARC - Laboratory of Computer Architecture and Networks) in University of São Paulo and who later introduced me to Dr. José Moreira. At LARC, I had the opportunity to work with wonderful people. Together we built amazing things, and had a lot of fun while doing that. The list of names is too long to enumerate here but they know what a great and fun team we made. However, one name stands out: Prof. Tereza Cristina Carvalho, who has always been a very good mentor and who has since became a very good friend as well.

I would like to thank my parents, who have always encouraged me to play with logic puzzles and educational toys, and supported me all the way through school. Cida, my second mother, has helped me a lot as well. I also attribute my success to my grandmother, Annelise, who was one of the first women to get a PhD in Medicine in Brazil and never seemed to notice that (and many other things!), who has always been involved in helping the community, and who has been a role model for me. I would like to thank my other grandmother, Marika, for having given me the genes that make me enjoy cooking and for having taught me (about a million times) how to cook her world famous risotto di funghi. Cooking is great stress-relief, with the very pleasant effect (most of the time, one hopes) of transforming raw ingredients into good food.

Finally, I would like to thank Luis Ceze. Without him, none of this would have happened. He has always been a great partner. His passion for science and technology kept fueling my own. He made my time in graduate school less stressful and much more enjoyable. He has helped me in so many different ways, it would be impossible to describe. I want to spend the rest of my life with him. I do.

Table of Contents

List of Tables . . . . **x**

List of Figures . . . . **xii**

List of Abbreviations . . . . **xiv**

List of Symbols . . . . **xv**

Chapter 1 Introduction and Motivation **1**

Chapter 2 Traditional Cache Coherence Approaches **4**
- 2.1 Collisions and Proper Serialization of Coherence Transactions 5
- 2.2 Shared Broadcast Bus 7
- 2.3 Point-to-Point Networks 8
  - 2.3.1 Unified Serialization Cache Coherence 9
  - 2.3.2 Distributed Serialization Cache Coherence 11

Chapter 3 Embedded-Ring Cache Coherence **14**
- 3.1 Basics of Embedded-Ring Cache Coherence 14
- 3.2 System and High-Level Protocol Overview 18
- 3.3 Enforcing Cache Coherence on Embedded-Ring Multiprocessors 21
  - 3.3.1 Cache Coherence Requirement 21
  - 3.3.2 Supporting the Invariant 23
  - 3.3.3 Protocol Operation 23
- 3.4 Correctness of Embedded-Ring Protocol 30
  - 3.4.1 Embedded-Ring Protocol Maintains Single Supplier Invariant 31
- 3.5 Forward Progress 36
- 3.6 Consistency Implications 37
- 3.7 Potential Optimizations 37
  - 3.7.1 Read Transactions Do Not Move Supplier Status 37
  - 3.7.2 Servicing Other Nodes Before Combined Response Arrives 38

Chapter 4 Flexible Snooping: Adaptive Forwarding and Filtering of Snoop Requests 39
4.1 Toward Flexible Snooping 39
4.1.1 Lazy and Eager Forwarding 39
4.1.2 Adaptive Forwarding and Filtering 41
4.2 Algorithms for Flexible Snooping 44
4.2.1 Design Space 44
4.2.2 Description of the Algorithms 45
4.2.3 Implementation of the Algorithms 47
4.3 Related Work 51

Chapter 5 Unconstrained Snoop Request Delivery 53
5.1 Benefits of Unconstrained Snoop Request Delivery 54
5.2 *UncoRq* in Action 56
5.3 Supporting Invariant I_1 with *UncoRq* 57
5.3.1 Overall Operation 59
5.4 Implementation Issues 60
5.4.1 Enforcement of Invariants C_2 and C_3 for *UncoRq* 60
5.4.2 Additional Optimization for *UncoRq* 63
5.4.3 Forward Progress 64
5.5 Related Work 66

Chapter 6 Evaluation of Embedded-Ring Cache Coherence 68
6.1 Experimental Setup 68
6.1.1 Simulation Infrastructure 68
6.1.2 Simulated Architectures 69
6.1.3 Predictors Used 71
6.1.4 Handling Write Snoop Requests 73
6.2 Evaluation 74
6.2.1 *Flexible Snooping* on Multi-CMP Configuration 74
6.2.2 *UncoRq* on Single-CMP Configuration 80
6.2.3 *Flexible Snooping* and *UncoRq* on Both Configurations 85
6.2.4 Comparison to Hierarchical Buses 88
6.2.5 Sensitivity Analysis 89

Chapter 7 Conclusions 101

Appendix A Exhaustive Enumeration of Message Interleavings 103
A.1 Regular Embedded-Ring Protocol 104
A.2 Embedded-Ring with Unconstrained Snoop Request Delivery 111

References 119

Author's Biography 123

List of Tables

3.1 Four different collision cases that need to be handled by embedded-ring cache coherence protocol. . . . 29

4.1 Comparing different snooping algorithms. . . . 40
4.2 Actions taken by each of the Flexible Snooping primitive operations. . . . 42
4.3 Proposed Flexible Snooping algorithm actions. . . . 45
4.4 Proposed Flexible Snooping algorithms characterization. . . . 46

5.1 Four different collision cases that need to be handled by *UncoRq*. . . . 59

6.1 Architectural parameters used for simulated processors. . . . 70
6.2 Architectural parameters used for Multi-CMP configuration. . . . 70
6.3 Architectural parameters used for Single-CMP configuration. . . . 71
6.4 Architectural parameters used for *Flexible Snooping* predictors. . . . 72
6.5 Architectural parameters used for prefetching predictors. . . . 73
6.6 Average data consumption latency for *Eager* and *UncoRq*, percent data consumption latency reduction and percentage of cache-to-cache transfers. . . . 83
6.7 Average data consumption latency for *UncoRq+Pref* and percent reduction of *UncoRq+Pref* compared to *UncoRq*. . . . 99

A.1 Rules derived from the baseline embedded-ring protocol operating principles, same-address FIFO order on ring links, and properties of the logical unidirectional ring. . . . 104
A.2 Message interleavings at nodes involved in a collision and their feasibility for baseline embedded-ring protocol. . . . 106
A.3 Feasibility of interleaving combinations between two nodes involved in a collision for baseline embedded-ring protocol. . . . 107
A.4 Outcomes for feasible interleaving combinations between two nodes involved in a collision when there is a single supplier node for baseline embedded-ring protocol. . . . 109
A.5 Outcomes for feasible interleaving combinations between two nodes involved in a collision when there is no supplier node for baseline embedded-ring protocol. 110
A.6 Rules derived from the embedded-ring network properties and the *UncoRq* protocol properties. . . . 112

A.7 Message interleavings at nodes involved in a collision and their feasibility for *UncoRq* protocol. 113
A.8 Feasibility of interleaving combinations between two nodes involved in a collision for *UncoRq* protocol. 114
A.9 Outcomes for feasible interleaving combinations between two nodes involved in a collision when there is a single supplier node for *UncoRq* protocol. . . . 116
A.10 Outcomes for feasible interleaving combinations between two nodes involved in a collision when there is no supplier node for *UncoRq* protocol. 118

List of Figures

2.1 Example of same-address serialization problem. 6

3.1 Miss serviced on an embedded-ring multiprocessor with *Eager Forwarding*. . 17
3.2 Machine architecture modeled and matrix of compatible cache states in the coherence protocol used. 19
3.3 Example of natural serialization on a collision in an embedded-ring. 25
3.4 Example of forced serialization on a collision in an embedded-ring. 26

4.1 Actions of a snoop request in three different algorithms. 40
4.2 Design space of Flexible Snooping algorithms according to the snoop request latency and the number of snoop operations per request. 47
4.3 Implementation of the supplier predictors. 48

5.1 Latency components for read miss services with cache-to-cache transfers. . . 55
5.2 Miss serviced on an embedded-ring multiprocessor with *UncoRq*. 56
5.3 Example of *UncoRq* failure to support invariant I_1. 58
5.4 LTT: Additional structure to support *UncoRq*. 60
5.5 Example of LTT supporting conditions C_2 and C_3 for correct operation of *UncoRq*. 62

6.1 Average number of snoop operations per read transaction for different *Flexible Snooping* algorithms. 75
6.2 Total number of read snoop requests and responses in the ring for different *Flexible Snooping* algorithms. 76
6.3 Execution time of the applications for different *Flexible Snooping* algorithms. 77
6.4 Energy consumed by on coherence transactions for different *Flexible Snooping* algorithms. 79
6.5 Data consumption latency on misses serviced with cache-to-cache transfers for *fmm* when executed on a system using *Eager* and *UncoRq*. 82
6.6 Execution time of *UncoRq* schemes. 83
6.7 Average number of snoop operations per read transaction for *UncoRq*. 84
6.8 Total number of read snoop requests and responses in the ring for *UncoRq*. . 85
6.9 Energy consumed on coherence transactions for different *UncoRq* schemes. . 85
6.10 Execution time of best embedded-ring snooping schemes on a Multi-CMP configuration. 86

6.11 Energy consumed on coherence transactions for best embedded-ring snooping schemes on a Multi-CMP configuration. 86
6.12 Execution time of best snooping schemes on a Single-CMP configuration. . . 87
6.13 Energy consumed on coherence transactions for best snooping schemes on a Single-CMP configuration. 88
6.14 Execution time of *HierBus* and *Superset Agg* on a Multi-CMP configuration. 89
6.15 Execution time of *HierBus* and *UncoRq* on a Single-CMP configuration. . . . 90
6.16 Execution time of *Superset Agg* on Multi-CMP configurations with varying number of nodes. 91
6.17 Execution time of *UncoRq+Pref* on Single-CMP configurations with varying number of nodes. 91
6.18 Execution time of *UncoRq+Pref* on a Single-CMP configuration with varying cache size. 92
6.19 Supplier predictor accuracy for the different implementations of the Subset, Superset, and Exact algorithms. 93
6.20 Execution time normalized to *Subset 2k* for a Multi-CMP system using the *Subset* predictor with varying predictor size. 94
6.21 Coherence energy for a Multi-CMP system using the *Subset* predictor with varying predictor size. 95
6.22 Execution time normalized to *Superset y2k* for a Multi-CMP system using the *Superset Con* algorithm with varying predictor configuration. 96
6.23 Coherence energy for a Multi-CMP system using the *Superset* predictor with varying predictor size. 96
6.24 Execution time normalized to *Exact 2k* for a Multi-CMP system using the *Exact* predictor with varying predictor size. 97
6.25 Coherence energy for a Multi-CMP system using the *Exact* predictor with varying predictor size. 97
6.26 Outcome breakdown of prefetching prediction on read misses. 98
6.27 Execution time normalized to *UncoRq+Pref* for a Single-CMP system using the prefetching predictor with varying predictor size. 99
6.28 Coherence energy for a Single-CMP system using the prefetching predictor with varying predictor size. 99

A.1 Assumed relative position of nodes for baseline embedded-ring protocol exhaustive enumeration. 108
A.2 Assumed relative position of nodes for *UncoRq* protocol exhaustive enumeration. 115

List of Abbreviations

CMP	Chip Multiprocessor
RT	Round-trip Time
SMP	Simultaneous Multiprocessor
UncoRq	Unconstrained Snoop Request Delivery

List of Symbols

R_X	Request issued by node X
$R_X(Y)$	Request of type Y issued by node X
r_X	Response corresponding to request issued by node X
r_X+	Positive response corresponding to request issued by node X
r_X-	Negative response corresponding to request issued by node X
$r_X+(Y, N)$	Positive combined response corresponding to request of type Y issued by node X, with N outcomes combined
r_X-(Y, N)	Negative combined response corresponding to request of type Y issued by node X, with N outcomes combined
$snoop(X, Y)$	Snoop operation of type Y requested by node X
$ls(X, Y)$	Snoop operation outcome for operation of type Y requested by node X

Chapter 1

Introduction and Motivation

Design complexity and limited power budget are causing the number of cores on the same chip to grow very rapidly. The wide availability of Chip Multiprocessors (CMPs) is enabling the design of inexpensive shared-memory machines of medium size (32-128 cores). However, as in traditional, less-integrated designs, supporting hardware cache coherence in these machines requires a major engineering effort.

Two traditional approaches to cache coherence are snooping and directory-based schemes. Snooping schemes that rely on one or more broadcast buses cannot scale beyond a small number of cores without significant increases in cost. On the other hand, directory schemes, while scalable, have the disadvantage of adding one level of indirection to coherence transactions, increasing latency. Moreover, directories can be expensive and complex to design.

A cost-effective approach for these machines is to support cache coherence on a point-to-point network rather than on a shared bus. This approach is sometimes referred to as *network-based* snooping cache coherence. While this approach is not as scalable as directory schemes, it is inexpensive and may represent the best design approach for medium machine sizes. This approach is used by IBM Power 5 systems [33].

Interestingly, while the shared bus in bus-based snooping schemes ensures that coherence messages are delivered in the same order to all the nodes, this is not the case in network-based snooping schemes — messages from two different concurrent transactions on the same address can be received by different nodes in different orders. This lack of ordering makes the design and verification of efficient network-based snooping schemes challenging.

In this work, we propose to *logically embed* a ring in the physical network topology

the machine uses. Snoop messages use the logical ring, while other messages can use any link in the network. The resulting design is simple and low cost. Specifically, it places no constraints on the network topology or timing. It needs no expensive hardware support such as a broadcast bus or a directory module. In addition, the ring's serialization properties enable the use of a simple cache coherence protocol. Finally, while it is not highly scalable, it is certainly appropriate for medium-range machines.

One advantage of the logical embedded-ring approach, as opposed to a physical ring approach, is that, unlike physical rings, embedded rings do not require nodes to use the ring on data transfers. Instead, nodes transfer data using shorter paths through the underlying physical network. This reduces cache-to-cache transfer latency and improves performance. An embedded-ring also improves on flexibility for system reconfiguration and partitioning.

Perhaps the main drawback of the embedded-ring approach is that snoop requests may suffer long latencies or induce 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. Alternatively, a scheme where each node immediately forwards the request and then performs the snoop will be shown to induce many snoop messages and snoop operations. This is energy inefficient. Unfortunately, as technology advances, these shortcomings become more acute: long latencies are less tolerable to multi-GHz processors, and marginally-useful energy-consuming operations are unappealing in energy-conscious systems.

Ideally, we would like to forward the snoop request as quickly as possible to the node that will provide a copy of the requested memory location while consuming as little energy as possible. To this end, we propose *Flexible Snooping* algorithms, a family of adaptive forwarding and filtering snooping algorithms [37]. 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.

We examine the design space of these algorithms and, based on the analysis, describe

four general approaches for these algorithms. They represent different trade-offs in number of snoop operations and messages, snoop response time, energy consumption, and implementation difficulty.

All the algorithms mentioned above, however, require that snoop requests and responses traverse the entire ring to service a transaction. Snoop request delivery to all nodes is sequential, which constrains the overlap of snoop operations and incurs long cache miss latencies.

To solve this problem, we also propose a novel technique we call *UncoRq* [36] that removes this restriction. It allows snoop requests to be delivered to multiple nodes *in parallel*, using *any path* in the network — as long as snoop responses, which are often off the critical path, use the logical ring. This greatly reduces miss latency, while still preserving protocol simplicity.

Overall, the contribution of this thesis is threefold:

1. It proposes embedded-ring multiprocessors and presents invariants used to ensure proper serialization of coherence transactions on the same address.

2. It introduces novel techniques called *Flexible Snooping* and *UncoRq* that substantially decrease coherence transaction latency and/or energy consumption.

3. It analyzes the performance and/or snoop energy consumption of the proposed techniques and shows they achieve the above goals.

Chapter 2

Traditional Cache Coherence Approaches

In shared memory multiprocessors, memory is exposed as a global shared memory with single addressing space that can be read or modified by any of the processors in the system. To reduce bandwidth requirements and improve performance, many of these systems use caches. Caches allow processors to replicate and store data locally, avoiding the need to contact main memory on every memory access.

However, replicating data causes a problem: there is no longer a single copy of the data accessible to all processors. There must be a mechanism to keep all copies of data coherent. This is called the cache coherence problem.

A number of mechanisms have been proposed to address the cache coherence problem. They can be generally divided into two classes of mechanisms: update-based or invalidate-based cache coherence. However, update-based protocols have been neglected by designers due to implementation difficulties. For the purposes of this thesis, we consider only invalidate-based cache coherence.

A cache coherent system is loosely defined by two conditions in [2]:

1. A write transaction is eventually made visible to all nodes.

2. Write transactions to the same location appear to be observed in the same order by all nodes.

For many memory consistency models, the second condition implicitly means that from the time a first write or invalidate transaction on a memory location completes until the moment the next write or invalidate transaction on the same memory location completes,

all nodes in the system either do not cache a copy of the memory location contents, or cache a copy of the memory location identical to what the node that issued the first write or invalidate transaction has written to the memory location. All write or invalidate transactions complete in a total order observed by all nodes, generating a sequence of values $v_0, v_1, ..., v_n$. This is called serialization of writes to the same location with respect to writes to that same memory location. A node cannot read a value v_i if any node has already completed a write or invalidate transaction modifying the memory location to value v_j, for $i < j$. For an invalidate-based mechanism, while a write or invalidate transaction that ultimately completes and modifies the memory location to a new value is in flux, nodes that have already received a write or invalidate request corresponding to that transaction should not be able to read another value that precedes the new value in the sequence of values. This is called serialization of writes to the same location with respect to reads to that same memory location.

Section 2.1 illustrates why a system that does not support the conditions above does not support cache coherence. Sections 2.2 and 2.3 show various approaches that address the cache coherence problem.

2.1 Collisions and Proper Serialization of Coherence Transactions

To provide cache coherence, a system needs to properly serialize write or invalidate transactions with respect to other write or invalidate transactions and write or invalidate transactions with respect to read transactions on a memory location basis. If the system cannot support this, different nodes may cache different values for the same memory location, making the system incoherent. For example, consider Figure 2.1.

Initially, nodes B and S cache the same copy of a memory location in Shared state (S) and node A does not cache it (I). B starts an invalidate transaction and broadcasts an

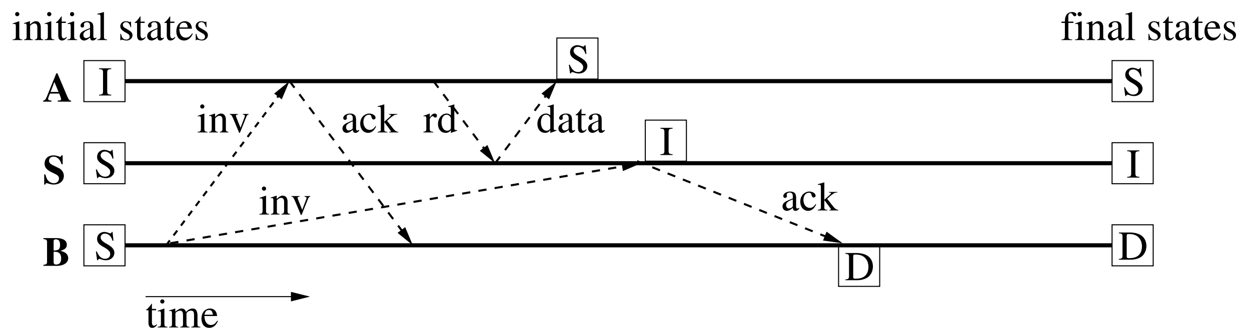


Figure 2.1: Example of same-address serialization problem.

invalidate request to the memory location, which reaches A first. A, which is not caching the memory location, acknowledges the invalidate to B. Later, A starts a read transaction and broadcasts a read request to the same memory location. A's read request reaches S before B's invalidate request does. S supplies the requested data (copy before B's modification) to A. A receives the response from S and caches the copy in Shared state (S). Later, B's invalidate request reaches S. S invalidates its copy and acknowledges the invalidate to B. B, having received acknowledgments from A and S, transitions to Dirty state (D) and modifies its copy. However, now the system is incoherent: A caches the old copy of the memory location in Shared state, while B caches the new copy of the memory location in Dirty state. This problem is caused by the inability of the system to serialize A's read transaction and B's invalidate transaction properly.

We define a *collision* as the simultaneous or overlapping processing of more than one coherence transactions involving the same memory location. Although collisions may be rare, a system needs to detect and handle them properly to maintain cache coherence. Without loss of generality, we will assume collisions of only two transactions throughout the text, but the solutions we propose work with any number of colliding transactions.

2.2 Shared Broadcast Bus

In this approach, transactions obtain access to a shared broadcast bus, which all processors snoop [16, 11]. Only one transaction can use the bus each cycle. If two transactions on any memory location attempt to simultaneously use the shared bus, only one succeeds, and the other is forced to wait, serializing all transactions, including those on the same memory location.

A drawback of buses is their limited scalability. When designers try to increase their frequency or number of connected nodes, they face challenging physical effects such as signal propagation delays, signal reflection and attenuation. Designers also face layout issues when the number of nodes connected to the same physical bus increases. In addition, scaling bus arbiters is a demanding task. Latencies in arbiters tend to grow superlinearly with the number of nodes. For example, Kumar et al. [22] estimate that, for 65nm process technology, no more than four nodes can be arbitrated in a cycle.

Modern designs use advanced implementations, such as split-transaction buses, hierarchical buses and partitioned buses, described next. For example, Sun's Starfire [35] uses four hierarchical, partitioned buses for snoop messages; data transfers are performed on another network.

Split-transaction buses allow more transaction parallelism because nodes issuing transactions do not hold the bus for their entire duration. After obtaining access to the bus and issuing a transaction, a node relinquishes the bus to other nodes. While the first transaction is being processed, nodes can issue other transactions that are processed in a pipelined fashion. After some time, the bus is used to communicate the results of the first transaction back to its issuing node. Note that separate buses can be used to broadcast transaction requests, to collect the transaction responses, and to transmit data.

Hierarchical buses are used to address scalability issues. Designing arbiters and routing wires for a board or chip become increasingly challenging tasks when the number of nodes connected to the same bus increases. Designers may use hierarchical buses to alleviate these problems. Only a few nodes are connected to a bus, which in turn is connected to other buses successively in a hierarchical structure, up to a root bus. This makes the design of arbiters and routing of wires more tractable, but also increases the number of arbiters a request has to go through before it can be broadcast, which could cost on latency and complexity.

Partitioned buses are used to increase the effective bandwidth of the system. The address space is partitioned among the available buses, and thus nodes can issue as many simultaneous transactions as available buses in the system, if they map to different memory locations.

Despite all these optimizations, it is difficult to imagine buses as the best solution for a high-frequency multiprocessor system with a significant number of processors (e.g. 32-128 cores) because of high design complexity, high miss latencies, low bandwidth or a combination of them.

2.3 Point-to-Point Networks

The alternative to shared broadcast buses is to use point-to-point networks. In this type of network, each link connects only two nodes, removing the need for a complex central arbiter. The system can be connected in a variety of topologies (e.g. ring, torus, mesh, hypercube). Cache coherence schemes that work on this type of networks are sometimes called *network-based* schemes.

Interestingly, while the shared bus in bus-based snooping schemes ensures that coherence messages are delivered in the same order to all the nodes, this is not the case in network-based schemes — messages from two different concurrent transactions on the same memory

location can be received by different nodes in different orders. This lack of ordering makes the design and verification of efficient network-based cache coherence schemes challenging.

Next, we present various network-based schemes for cache coherence. We subdivide the solution space into unified and distributed serialization cache coherence, discussed in Sections 2.3.1 and 2.3.2, respectively.

2.3.1 Unified Serialization Cache Coherence

The most straightforward solution to enforce ordering in a point-to-point network is to elect, for each memory location, a central point that all coherence transaction requests have to reach before they are sent to other nodes in the network. This central point is responsible for ordering all transactions on the same memory location. This is the approach used in directory-based protocols and in AMD's cache coherent HyperTransport, discussed in the next few sections. Although these solutions require a central point of serialization for each memory location, they allow different memory locations to be partitioned throughout multiple serialization points.

The main drawback of these protocols is that they require a 3-hop indirection, sometimes a 4-hop indirection, that incurs latency to cache-to-cache transfers.

Directory-Based Cache Coherence

In directory-based protocols, all transactions on a memory location are directed to the home node of that memory location [8, 38, 4, 23, 3]. The home node serializes the transactions — for example, by bouncing or buffering the transaction that arrives second until the one that arrived first completes.

Because the home node receives and forwards all requests corresponding to transactions on a memory location, it can easily keep track of all nodes that have a copy of that memory location in a directory, using this information to forward requests on this memory location only to nodes that need to take notice of the request, i.e., that currently cache a copy of the

memory location. This avoids unnecessary messages and saves bandwidth.

While directory protocols such as that in Silicon Graphics's Altix [32] are scalable due to their bandwidth saving capabilities, they add non-negligible overhead to a mid-range machine — directories introduce a time-consuming indirection in all transactions. All requests have to pass through the home node before being delivered to other nodes in the system, resulting in a 3-hop communication when another node supplies a copy of the requested memory location (requesting node to home node, home node to supplier node and supplier node back to requesting node), and that negatively impacts miss latency. In addition, to simplify the design, some protocols require a 4-hope communication (requesting node to home node, home node to supplier node, supplier node to home node and home node back to requesting node), which makes miss latency even longer. Moreover, since directory-based protocols keep track of memory location sharers to avoid unnecessary broadcasts, the directory itself is a complicated component, with significant state and logic.

Cache Coherent HyperTransport

The cache coherent HyperTransport [10, 20] uses the same serialization mechanism directory-based protocols use. Each memory location is assigned a home node, where all coherence transactions on this memory location are serialized. Unlike directory-based protocols, however, cache coherent HyperTransport does not keep track of sharers of any memory location. As a consequence, this mechanism broadcasts memory transaction requests after they are serialized at the home node.

The trade-off between cache coherent HyperTransport and directory-based protocols is one of storage and complexity against bandwidth savings. While cache coherent HyperTransport is simpler, it broadcasts requests, consequently using more bandwidth than directory-based protocols.

Main Drawback of Unified Serialization

As mentioned in the previous sections, the 3-hop or 4-hop communication required by unified serialization cache coherent protocols adds to the latency of cache transfers between nodes. As the size of caches grow, this type of cache transfer tends to increase for applications with high degrees of sharing, which may render protocols based on unified serialization less efficient.

2.3.2 Distributed Serialization Cache Coherence

The alternative to unified serialization is distributed serialization. In this approach, a distributed algorithm is responsible for serializing simultaneous transactions on the same memory location to support cache coherence. The distributed algorithm may or may not rely on partial ordering guarantees provided by the network to enforce proper serialization of transactions.

Token Coherence

In Token Coherence [25], each memory line, whether in cache or in memory, is associated with N tokens, where N is the number of nodes participating in the coherence protocol. A node cannot read a memory location until its cache obtains at least one of the memory location's tokens. A node cannot write to a memory location unless its cache obtains all of the memory location's tokens. This convention ensures that transactions on the same memory location are serialized, irrespective of the network used. Partial overlap results in failure of one or both transactions to obtain all necessary tokens. These transactions then retry.

While conceptually appealing, the scheme has some potentially difficult implementation issues. One of them is that retries may result in continuous collisions, potentially creating live-lock. A solution based on providing some queuing hardware to ensure that colliding

transactions make progress is presented in [26]. Another issue is that every memory location needs token storage in main memory, since some of the memory locations's tokens may be stored there. Unless special care is taken, such token memory may need to be accessed at write transactions. Finally, in multiprocessors with multiple CMPs, the scheme needs to be extended with additional storage and state to allow a local cache in the CMP to supply data to another local cache. Some of these issues are addressed in [26].

Physical Ring Cache Coherence

Unidirectional physical rings [41, 17, 5] have been proposed as a substitute for a bus in a multiprocessor system. They provide partial ordering of messages, requiring only a simple distributed arbitration algorithm to serialize concurrent transactions on the same address not naturally serialized by the ring network. To serialize these colliding transactions, the algorithm can pick a winner transaction, which is allowed to complete. All other colliding transactions are retried.

In such systems, there are two traditional mechanisms to handle transactions: *Lazy* and *Eager Forwarding*. With Lazy Forwarding, a transaction request stalls at a node while the node performs a snoop operation, and is combined with the snoop operation outcome before proceeding to the next node. With Eager Forwarding, a transaction request proceeds to the next node as soon as it arrives at a node. A separate response message follows collecting the snoop outcomes at each of the snooping nodes. The trade-off between Lazy and Eager Forwarding is one of snoop latency versus number of messages used to complete a transaction.

However, both mechanisms are restricted to a ring network topology, and require data to be sent to requesting nodes using the ring network itself, which can result in long latencies, depending on the number of nodes connected to the ring.

Embedded-ring Cache Coherence

This is the scheme we propose. The idea is to embed a logical unidirectional ring into any network topology. Only control messages (i.e. request and response messages) need to use the ring, while other messages (i.e. data messages) can use any links in the network.

Embedded-ring cache coherence can use the same distributed arbitration algorithm employed by physical unidirectional ring systems, as well as forwarding mechanisms. However, embedded-ring cache coherence provides better performance because it is able to communicate data using links other that the ring links, reducing the data latency from the supplier node to the requester node.

Chapter 3

Embedded-Ring Cache Coherence

3.1 Basics of Embedded-Ring Cache Coherence

We now define the terminology used throughout this thesis and describe how a cache miss is serviced in an embedded unidirectional ring multiprocessor using *Eager Forwarding*.

When a node suffers a miss, it initiates a *coherence transaction* and sends a *request* to the next node on the ring. A node issuing a request is called *requester node*. When a *snooping node* receives a request originated at a requester node, it first forwards the request to the next node on the ring and then initiates a *local snoop operation*, which in turn generates a *local snoop outcome* that indicates whether the snooping node can supply the requested data (*positive outcome* in case it can, *negative outcome* otherwise).

A *combined response* is a message that indicates the snoop outcomes at one or more nodes. Specifically, when a node receives a combined response, the response contains the combined outcomes of snoop operations performed by nodes positioned between the requester node and the receiving node on the ring. For each cache miss, a single combined response traverses the ring, visiting all nodes. At every snooping node, after the node receives the combined response from the previous node on the ring and completes its own local snoop operation, the node combines the contents of the received combined response with its own local snoop outcome and forwards the new combined response. To combine snoop outcomes, the node follows a simple rule: if any of them is positive, the result is positive; otherwise, the result is negative. Consequently, a response is called a *positive response* if it contains a positive outcome. A response is called *negative response* when it contains only negative outcomes.

There are two conditions for a node to generate and forward a newly combined response: (1) the request must have been received, and the snoop operation initiated by the request must have completed and generated a local snoop outcome; and (2) the node must have received a combined response from the previous node on the ring. If (1) is satisfied, but (2) is not, the node records the local snoop outcome and waits for (2) to be satisfied. If (2) is satisfied, but (1) is not, the node records the received combined response and waits for (1) to be satisfied[1].

When a local snoop operation determines that the snooping node can supply the requested data, the snooping node generates a positive snoop outcome and immediately sends the data to the requester (through the *shortest* path, not necessarily through the ring).

When the requester node receives the data, it can use the data immediately, retire the load instruction (at this point it is considered globally performed), and proceed with the execution of dependent instructions. However, the requester node only changes the line state to a stable state when it receives the corresponding positive combined response. Until the state is stable, the node does not provide data to other nodes.

When a requester node receives a positive response, a *cache-to-cache transfer* has taken place. As soon as the requester node receives both data and the positive response, the miss service is concluded. When the requester node receives a negative combined response, however, no node can supply the requested data. The requester node then sends a data request to memory and, when the memory responds with data, the miss service concludes. In this case, a *memory-to-cache transfer* has taken place.

Figure 3.1 illustrates how a read miss is serviced with a cache-to-cache transfer in an embedded-ring multiprocessor with *Eager Forwarding*. Figure 3.1(a) shows the ring and three nodes, A, S and B, as well as the direction request (solid line) and combined response

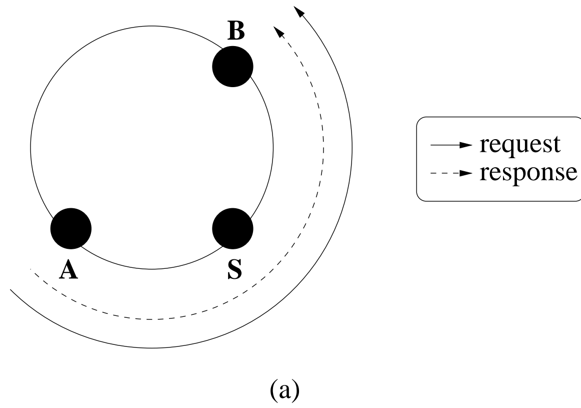
[1]For read transactions, when the local snoop outcome of (1) is positive, there is no need to wait for the combined response of (2) in order to send the positive response. However, more bookkeeping is needed because, eventually, the negative response from previous nodes arrives and has to be ignored. For write and invalidate transactions, it is necessary to wait for (2) in order to collect acknowledgments from all nodes on the ring in the same response.

(dashed line) traverse the ring. Figure 3.1(b) shows a timeline of events. The notation used in Figure 3.1(b) and throughout this thesis is as follows: requests are denoted by $R_X(Y)$, snoop operations are denoted by $snoop(X,Y)$ and local snoop outcomes are denoted by $ls(X,Y)$, where X is the requester node ID and Y is the type of request. A combined response is denoted by r_X+(Y,N), if the outcome is positive, or r_X-(Y,N), if the outcome is negative. N represents the number of collected snoop outcomes so far. Circled numbers are events we refer to in the text, usually, but not necessarily, in chronological order.

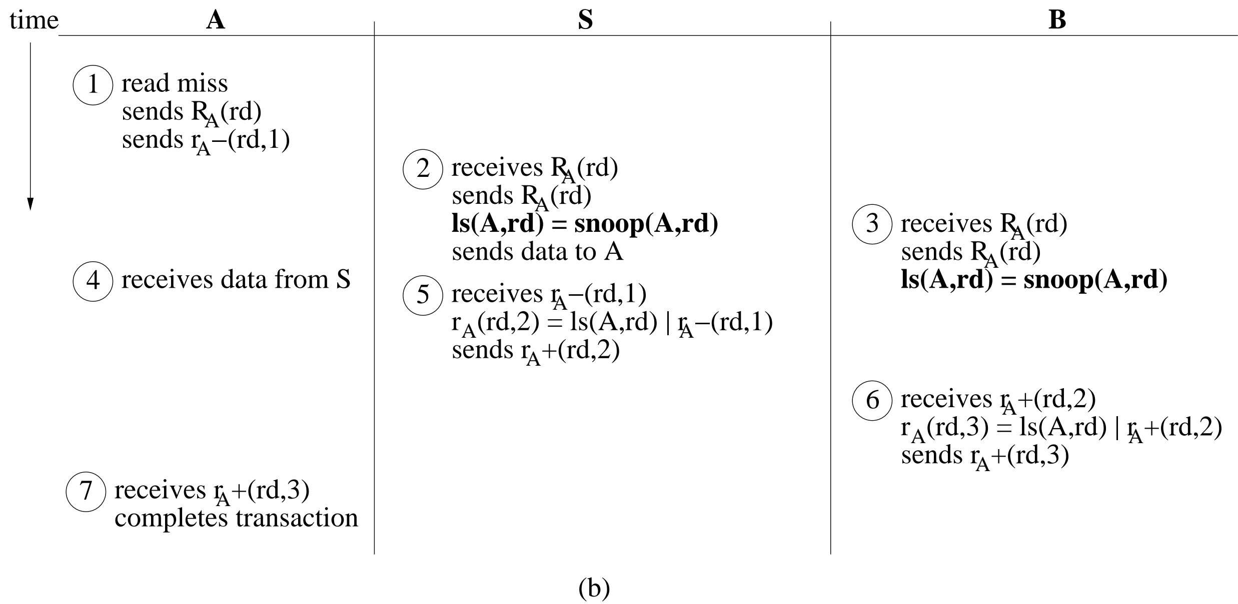
When A suffers a read miss (1), it places a read request and a negative response on the ring. S receives A's read request (2), sends it to B and initiates a snoop operation. B receives A's read request from S (3) and also initiates a snoop operation. When S completes its local snoop operation, the outcome indicates that S can supply data to A, which S does immediately. A receives the requested data from S (4) and is free to use the data before the miss service concludes. When S receives A's combined response (5), S combines it with its local snoop outcome, which is a positive outcome, and forwards the new (positive) combined response to B. When this combined response reaches B (6), B combines it with its local snoop outcome and forwards the new combined response to A. When the combined response reaches A (7), A completes the transaction.

As noted above, the embedded-ring protocol is designed such that it allows the requester, on read misses, to consume data as soon as it is received from the supplier, even if the requester has not yet received the combined response. This is beneficial because data does not need to be sent using the unidirectional ring, while responses do. Chances are that data will arrive to the requester before the combined response does. Observe that this fast data transfer is only possible in an embedded-ring system. In a physical ring system, data would have to traverse, on average, half of the ring before reaching the requester node.

Although read transactions are considered globally performed as soon as the requester receives the data, write and invalidate transactions are considered globally performed only after the requester receives the combined response.



(a)



(b)

Figure 3.1: Miss serviced on an embedded-ring multiprocessor with *Eager Forwarding*.

3.2 System and High-Level Protocol Overview

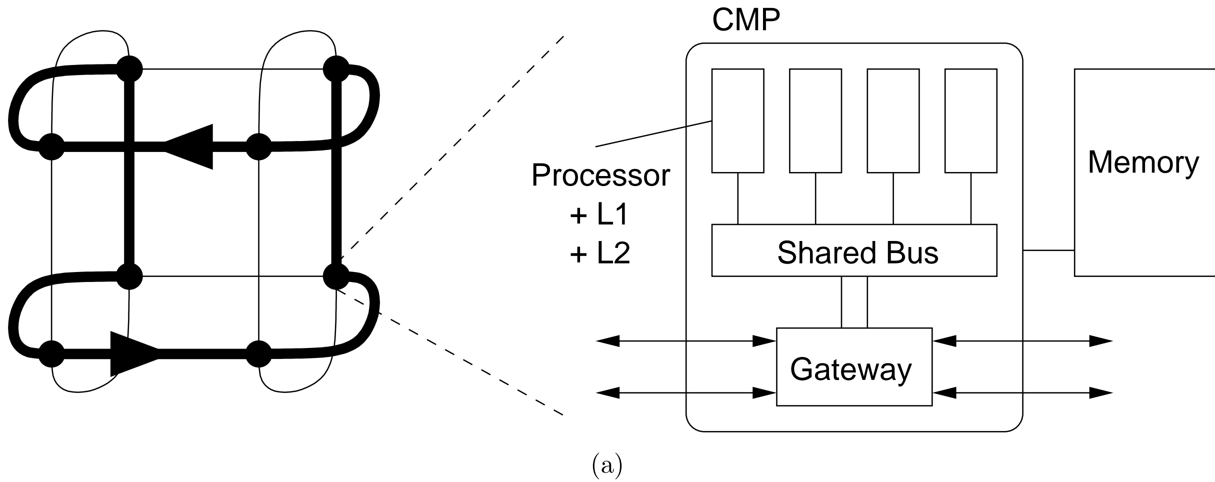
We briefly present two example systems to which we can apply the embedded-ring mechanism and outline the high level snooping protocol we consider.

The first system, a multi-CMP multiprocessor, is shown in Figure 3.2(a). It is composed of CMP nodes connected by a point-to-point network. Each CMP has several cores, each with a private L2 cache. The L2 caches within a CMP are connected by a shared broadcast bus, which in turn is attached to a portion of main memory. The CMPs can be interconnected in any physical topology (a 2D torus in this example), on which we embed one or more unidirectional rings for control messages. If more than one unidirectional ring is embedded, control messages may be mapped to different rings according to their memory address. This helps balancing the load on the underlying physical network. Data-transfer messages do not use the logical ring.

The second system, a single-CMP multiprocessor, differs from the first system in that each node is just a processor core together with its private L1 and L2 caches, and is not necessarily attached to a portion of main memory in a one-to-one mapping. Instead, the entire CMP is connected to one or a few main memory modules.

We chose to consider systems with private L2 caches because they provide lower hit latencies, require lower associativities and are simpler to design than shared caches. In addition, write-through L1 caches are important for reliability. Private L2 caches only handle write-through accesses from local L1 caches, reducing network traffic and, consequently, energy consumption.

The protocol we use is similar to the one used in IBM POWER4 systems [39]. Both protocols attempt to maximize the number of cache-to-cache transfers by allowing clean and dirty sharing. Cache-to-cache transfers are more desirable than memory-to-cache transfers with current technology, unlike previous technologies. The abundance of transistors on chip allows much more on-chip cache space, making cache accesses much faster than memory ac-



(a)

State	Compatible States
I	I, S, S_L, S_G, E, D, T
S	I, S, S_L, S_G, T
S_L	I, S, S_L^*, S_G^*, T^*
S_G	I, S, S_L^*
E	I
D	I
T	I, S, S_L^*

(b)

Figure 3.2: Machine architecture modeled (a) and matrix of compatible cache states in the coherence protocol used (b). In the network, the darker line shows the embedded ring. In the protocol table, "*" means that a line can be in this state only if it is in a *different* CMP.

cesses. In addition, providing data from caches reduces the off-chip bandwidth requirements, as well as energy consumption.

We use a MESI coherence protocol [11] enhanced with additional states. In addition to the typical Invalid (I), Shared (S), Exclusive (E), and Modified (or Dirty (D)) states of a MESI scheme, we add the global/local supplier status to the Shared state (S_G and S_L) and the Tagged (T) state.

To understand the global supplier status, consider a set of caches with a Shared line. If a cache outside the set reads the line, at most one of the caches can supply the line — the one with the global supplier status (S_G). In our protocol, the cache that first brings a copy of a memory location gets the global supplier status for that memory memory location until

it supplies a copy of the memory location and the supplier status to another cache (at which point it transitions to a Shared state).

If the system has multiple CMPs, performance would improve if reads were satisfied by a local cache (i.e., one in the same CMP), even if it did not have the global supplier status for that line. Consequently, we allow one cache per CMP to have the local supplier status (S_L). The cache that brings in the line from outside the CMP retains the local supplier status until it evicts the line or gets invalidated.

The T state is used to support the sharing of dirty data. In T state, the line is dirty, but coherent copies can also be found in other caches (in S or S_L state). On eviction, a line in T state is written back to memory.

Figure 3.2(b) shows which states are compatible with which other ones. It should be noted that, for any transaction, at most one cache (or none) can supply the requested data. In the following, we give two examples of read transactions serviced in a Multi-CMP system.

Read Satisfied by Another Cache When a requester cache issues a read transaction, a read request is created and results in snoop operations in the requester CMP's caches. If a copy of the memory location is found in S_L, S_G, E, D, or T state, it is supplied to the requesting node. Otherwise, the read request is forwarded to the ring. When the read request reaches another node, it enters the CMP and results in snoop operations in all of the CMP's caches. If no cache has a copy of the requested memory location in state S_G, E, D, or T, they generate a negative response indicating that the CMP cannot supply the data to the requester node. If instead a cache has a copy of the requested memory location in one of the above states, it generates a positive snoop outcome, sends a copy of the requested data to the requester node using any links in the network (not necessarily the ring) and transitions into S state. In parallel, a response message traverses the ring collecting snoop outcomes until it reaches the requester node. At this point, the combined response will be positive indicating that a cache was able to supply the requested data. The requester cache

transitions into a stable state with the global supplier status when it has received both the requested data and the combined response. However, the embedded-ring protocol allows the requester node to use the supplied data copy as soon as it reaches the requester node, even before the combined response arrives.

Read Satisfied by Memory If the combined response returns to the requester as negative indicating that node node was able to supply the requested data, the requester node sends a read message to the memory at the home node. When memory responds with the requested data, the requester cache transitions into E state. Both this message and the memory reply use the regular routing algorithm on any path in the network. To minimize the latency of this DRAM access, we may choose (with certain heuristics) to initiate a memory prefetch when the home node of the requested memory location is snooped. This would reduce the latency of a subsequent memory access.

3.3 Enforcing Cache Coherence on Embedded-Ring Multiprocessors

3.3.1 Cache Coherence Requirement

In order to enforce cache coherence, a system needs to guarantee that all nodes observe value changes to each memory location in the same sequence, i.e., that colliding transactions are properly ordered. However, doing that in a unordered network is challenging because there is no central coherence transaction arbitration.

To achieve this goal, we employ a single global supplier, invalidate-based cache coherence protocol, which guarantees that there is at most one node responsible for supplying copies of a memory location to other nodes. Within a node, there can be a local supplier cache, responsible for supplying copies of a memory location to other reader caches in the same

node[2]. However, only after obtaining global supplier status over this memory location and invalidating this memory location in all other caches, can a cache modify it. Global supplier status is transferred to the requester of the last successful transaction on a memory location. Since only the global supplier status is important for transaction serialization, we focus on this type of status and call it simply supplier status.

We determine which colliding transaction is successful (i.e., how to order colliding transactions) by using the sequence in which the supplier status is transferred. The supplier node provides data and supplier status to the requester of the first request to reach it, and that transaction, the *winner transaction*, is allowed to complete. We rely on the order of responses on the ring to notify nodes in the system about which colliding transaction has succeeded. *The order in which the initial supplier processes colliding requests is maintained in the order responses travel the ring after they leave the initial supplier.* Based on that, we introduce invariant I_1, which is sufficient to ensure serialization of transactions when there is a supplier node in an embedded-ring multiprocessor. The *loser transaction* may be retried or not. We describe the cases in which either happens in the next few sections, as well as what happens in case there is no supplier node to handle the conflict between colliding transactions.

Invariant I_1: The order in which two colliding transactions are ordered is the order in which their corresponding requests R_I and R_J arrive at the initial supplier node S. To communicate this sequence to the first colliding node after S in ring order (e.g. J), responses r_I and r_J must travel the ring in the same order as requests R_I and R_J arrived and were processed by S, such that the first of the two nodes involved in the collision can enforce the serialization of the two transactions by either (i) completing the winner transaction and then servicing the loser transaction or (ii) forcing the loser transaction to retry.

[2]To ensure correct operation, we assume all caches within a node are invalidated atomically, such that a node with local supplier status cannot provide an old copy of a memory location after another cache on the same node is invalidated.

More specifically, a positive response should not be overtaken by negative responses following it on the ring or through the nodes subsequent to the supplier on the ring until this positive response is removed from the ring by its requester.

3.3.2 Supporting the Invariant

To understand how an embedded-ring protocol supports Invariant I_1, consider the following. First, both requests and responses traverse the ring in the same direction. Second, we limit transaction overlap, which means that while handling a transaction originating from another node, a node cannot issue another request on the same memory location. Third, we enforce same-address FIFO processing of requests at the nodes and same-address FIFO transfer at the links. With that, the only possible reordering of messages on the same address is the request of one transaction passing responses of other colliding transactions.

Considering all that, responses traverse the ring in the same order as their corresponding requests do, i.e., if a node receives two requests in a particular order (e.g. R_I, then R_J), it will receive and send the corresponding responses in that same order (r_I, then r_J). This is sufficient to enforce invariant I_1 because if a supplier S receives requests R_I and R_J in this order, it will process them and send out responses r_I and r_J in this order, which will be preserved from then on.

3.3.3 Protocol Operation

We now present an embedded-ring protocol that uses invariant I_1 to enforce proper serialization of coherence transactions. In the next few sections, we introduce *natural* and *forced serialization*, and then present the overall protocol operation.

Collision with Natural Serialization

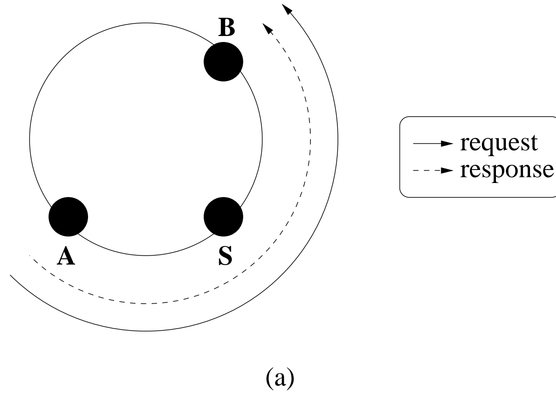
Natural serialization occurs when there is no interference between the nodes involved in a collision. Only after the first node completes its transaction will it receive messages corresponding to the other colliding transaction and the other node involved in the collision will not send any message corresponding to its transaction before processing and sending the response corresponding to the first transaction. Typically, when natural serialization occurs, the protocol does not require any transaction to retry. The winner node completes the transaction and then services the loser transaction. Figure 3.3 shows an example of natural serialization, based on the scenario in Figure 2.1.

As in Figure 2.1, B starts an invalidate transaction (1) and sends an invalidate request R_B, followed by a negative response r_B-, to A. A receives R_B (2), forwards it to S and performs a local snoop operation, which generates a negative outcome. A then combines the negative outcome with r_B- and forwards it to S. After that, A suffers a read miss to the same line (3) and sends a read request R_A, followed by a negative response r_A-, to S. The same-address FIFO forwarding policy guarantees that S receives R_B (4) before S receives R_A (5). Even though there is a collision, the situation in Figure 2.1 does not occur: the partial ordering imposed by the unidirectional ring prevents the read transaction from interfering with the invalidate transaction. This is what we call *natural serialization.*

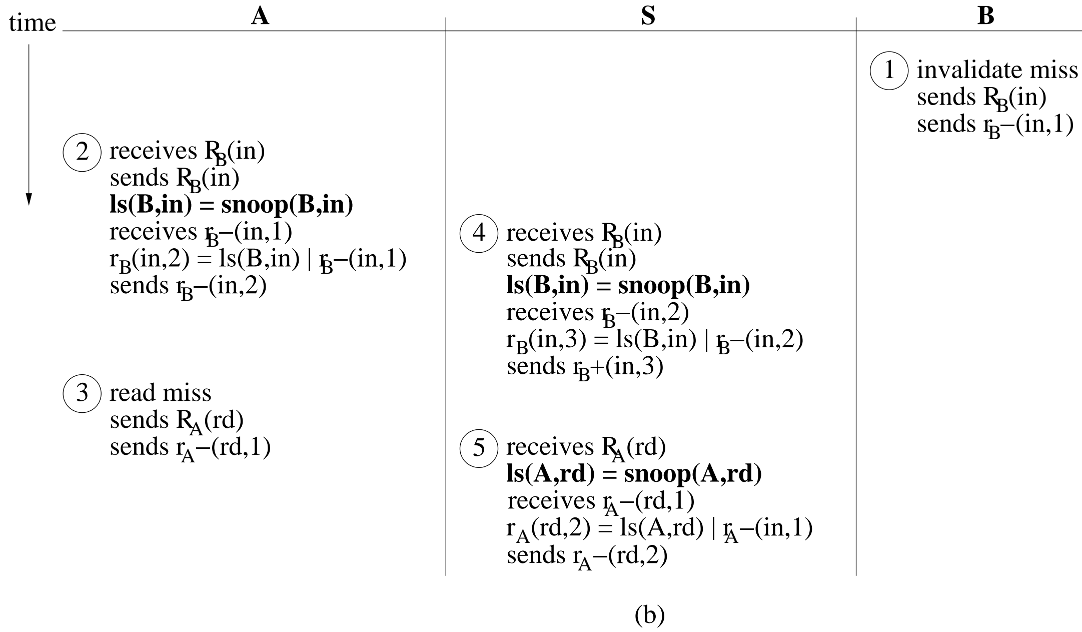
Collision with Forced Serialization

Opposed to natural serialization, forced serialization occurs when a colliding node receives a message from the other colliding transaction before it receives its own response. Forced serialization only allows one transaction to complete, and forces other colliding transactions to retry. Figure 3.4 shows an example of forced serialization on an embedded-ring, with a situation slightly different from the one presented in Figure 3.3.

Instead of suffering a read miss after receiving B's invalidate request, as in Figure 3.3, A suffers a read miss before receiving B's invalidate request. When A suffers the read miss

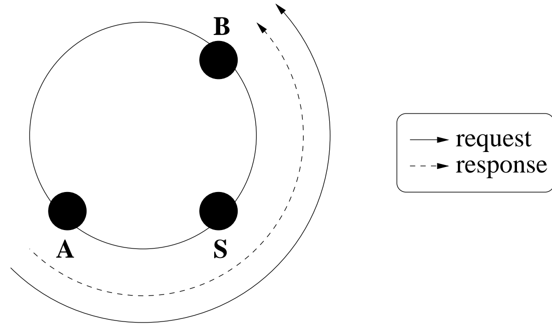


(a)

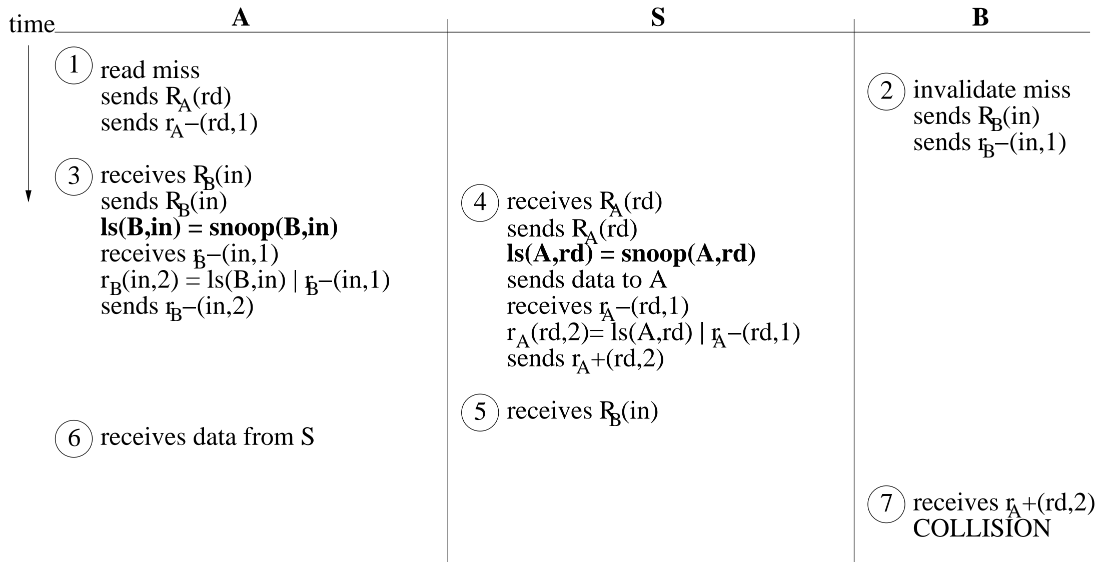


(b)

Figure 3.3: Example of natural serialization on a collision in an embedded-ring.



(a)



(b)

Figure 3.4: Example of forced serialization on a collision in an embedded-ring.

(1), A sends a read request R_A to S, followed by a negative response r_A-. B initiates an invalidate transaction (2) and sends an invalidate request R_B to A, followed by a negative response r_B-. A then receives R_B (3), forwards it to S, performs a snoop operation (with no effect, since it is in transient state), combines its snoop outcome with r_B-, and forwards r_B- to S. In this case, S receives R_A and r_A- (4) before receiving R_B (5). When S receives R_A (4), S forwards it to B, performs a snoop operation which results in a positive snoop outcome, supplies the requested data to A, combines its positive outcome with r_A- into r_A+, and forwards it to B. Only later does S receive R_B (5). By the time the data supplied by S reaches A (6), the invalidate snoop operation at A has already completed, and both R_B and r_B- have already been forwarded. Meanwhile, B receives r_A+ from S (7). If no special action is taken at this point, r_A+ will eventually reach A and cause it to transition its line to Shared state, while r_B- will also eventually reach B and cause it to transition its line to Dirty state. These two states are incompatible, so coherence would be violated. Fortunately, when B receives r_A+ from S (7), B can verify that it has an outstanding invalidate transaction on the same line and detect the collision. The protocol then forces one of the transactions to retry. Since data can already be en route from S to A when the collision is detected at B, the invalidate transaction from B is retried. After detecting and resolving the conflict, B forwards r_A+ back to A normally, which is able to complete the transaction (not shown). When r_B- reaches B, B retries the invalidate transaction (also not shown). This is what we call forced serialization.

The unidirectional ring properties have allowed the detection of the collision above by guaranteeing that at least one node involved in the collision, node B, receives responses in the same sequence as the supplier, node S, performs snoop operations.

Overall Operation

So far, we have focused on the case in which there is a supplier node for the requested data. However, a supplier may not always be present for a particular line. On another dimension,

serialization may be natural or forced. Thus we identify four cases that require different protocol actions, shown in Table 3.1. Without loss of generality, we assume the relative position of nodes on the ring is A (first colliding node), then S (supplier, if there is one), then B (second colliding node).

Case 1: Supplier Present, Natural Serialization In this case, r_A traverses the entire ring before A receives R_B. When A receives r_A+, if A has already received the requested data (as will happen most of the time), A completes its transaction and is ready to service B's transaction. However, if A receives r_A+ and then R_B before receiving the requested data, it cannot service B's transaction. The solution is to force B to retry its transaction, which is very similar to case 3.

Case 2: Supplier Present, Forced Serialization In this case, A and/or B observe each other's requests before observing their own responses. Due to the position of nodes on the ring, R_A reaches the supplier first, and A is allowed to conclude its transaction. B determines it should retry its transaction when it receives r_A+. B then retries its transaction when it receives r_B-.

Case 3: Supplier Not Present, Natural Serialization This case is similar to the first case: r_A traverses the entire ring before A receives any messages from B. However, A receives r_A-, indicating that no other node is able to supply the requested data. A then starts a memory access, but receives R_B before it receives the data from memory. A immediately identifies that B should retry its transaction. Later, when A receives r_B-, A communicates to B that it should retry its transaction using r_B-retry.

Case 4: Supplier Not Present, Forced Serialization In this case, A and B may observe each other's requests before observing their own responses, but cannot decide which transaction should be retried only based on the messages they receive; an additional algo-

Supplier Node	Serialization	Process
Present	Natural	A receives r_A+ (and data if relevant) A becomes the supplier A receives R_B A sends supplier status (and data if relevant) to B
Present	Forced	A receives R_B (no effect – A is in transient state) A receives r_B- A records B's ID and whether R_B is a write/invalidate S receives R_A (A's transaction is the winner) S receives R_B (S is no longer the supplier) B receives R_A (no effect – B is in transient state) B receives r_A+ B invalidates data if needed and marks its own transaction as loser A receives r_A+ A completes its transaction B receives r_B- B retries its transaction
Not Present	Natural	A receives r_A- A starts memory access A receives R_B A marks B's transaction as loser A receives r_B- A marks r_B- to force retry – r_B-retry A receives data from memory A becomes the new supplier B receives r_B-retry B retries its transaction
Not Present	Forced	A receives R_B, B receives R_A (no effect – A and B are in transient state) A receives r_B-, B receives r_A- A records B's ID and whether R_B is a write/invalidate B records A's ID and whether R_A is a write/invalidate A and B separately run winner selection algorithm if winner, node sends r_{other}-retry if loser, node invalidates data if needed and retries its transaction when it receives r_{own} A receives r_A-, B receives r_B-

Table 3.1: Four different collision cases that need to be handled by embedded-ring cache coherence protocol.

rithm is needed. A few possible algorithms are: (1) using ID numbers to determine priority (not fair, but never ties); (2) using random numbers attached to transactions (fairer, but could tie); (3) if one of them is a read transaction but the other one is not, the read transaction always loses. This allows a write or invalidate transaction to complete. When the read transaction retries, it reads a more updated value; (4) if one of them is an invalidate transaction but the other one is not, the invalidate transaction always wins. This avoids unnecessary accesses to memory, since the node issuing the invalidate transaction already caches a copy of the memory location (more optimized, but could also tie). We use a combination of these four algorithms, starting with 4 and then falling back to 3, 2 and 1, if any of them have a tie. If a node wins, it tags the other node's response with a retry signal

(r_{other}-$retry$) when it receives it. Otherwise, it retries its transaction when it receives the response corresponding to its own transaction (r_{own}).

Sometimes, forced serialization is necessary even if one of the nodes does not detect a collision. In these cases, at least one of the nodes is guaranteed to detect the collision and take appropriate action. Because of these situations, nodes detecting collisions cannot always rely on the other nodes involved in the collision to also have detected the collision, so every node detecting a collision (i.e., receiving a request corresponding to a transaction on the same address they have a transaction outstanding) immediately uses the distributed arbitration algorithm to determine a winner and notify the other node involved in the collision in case the other node is the loser. Since the distributed arbitration algorithm used on every node is the same, nodes involved in a collision never disagree about which is the winner. Because of the relative position of the supplier node, a node involved in a collision that has already used the distributed arbitration algorithm to determine a winner may receive a positive response corresponding to the other node, in which case the distributed arbitration algorithm output is ignored. For the same reason, a node could also receive a positive response with the retry signal, in which case the node ignores the retry signal. More details about these cases can be found in Appendix A.

3.4 Correctness of Embedded-Ring Protocol

In this thesis, we are only concerned with the correctness of the low level embedded-ring protocol portion of the system. We are not concerned with ordering of memory accesses locally at the nodes. We focus on events external to the coherent caches, i.e., the coherence transactions that result from cache misses and their relative order. Each node can have at most one outstanding coherence transaction on a particular memory location at a time.

Our goal is to show that the embedded-ring protocol in fact provides a total order for any set of transactions on the same address (these may include read, write and invalidate

transactions), which is used to support cache coherence. By total order we mean that coherence transactions on the same memory location are fully serialized, i.e., a read transaction returns the value written by the last write or invalidate transaction ordered before it in this total order. Next, we show at a high level that the embedded-ring protocol described in this chapter indeed provides this property. In Appendix A we provide an exhaustive enumeration of message interleavings and show that the protocols we propose are capable of detecting all collisions and recover from them while maintaining proper serialization of coherence transactions.

As briefly described in Section 3.3.1, the base mechanism to enforce a total order is the supplier status. At any time, at most one node has the supplier status for a given memory location. We call this the single supplier invariant.

Without the supplier status, a node can neither provide a copy of the corresponding data to other nodes nor modify its own copy. Therefore, if we show that the embedded-ring protocol maintains the single supplier invariant, we show that the protocol properly serializes transactions on the same memory location.

3.4.1 Embedded-Ring Protocol Maintains Single Supplier Invariant

At system initialization time, no node has the supplier status for any memory location, so the invariant holds initially. Supplier status is acquired in the following two situations: (a) when no node has the supplier status, in which case a requesting node accesses memory to get a copy of the memory location and to acquire the supplier status[3] or (b) when another node has the supplier status, in which case this other node will transfer the supplier status and a copy of the memory location, if needed, to the requesting node. As long as the protocol

[3]Note that memory does not keep any state to indicate supplier status. Any node that is able to complete a coherence transaction that indicates the requested data should be retrieved from memory can acquire the supplier status when it receives the requested data from memory. This is because the protocol only allows one node to complete its coherence transaction at a time, even when there is a collision.

guarantees that only one requesting node receives the supplier status at a time, the invariant holds.

When a requesting node issues a coherence transaction and it does not collide with any other coherence transaction on the same memory location, enforcing the single supplier invariant is trivial. The transaction is processed normally by each node. If any node has the supplier status, the node transfers it to the requesting node; otherwise, after all nodes have processed the transaction, the requesting node receives a negative response and accesses main memory to retrieve the requested data and acquire the supplier status. Since there are no colliding transactions, the requesting node is guaranteed to uniquely acquire the supplier status. This preserves the single supplier invariant.

For colliding transactions, we show that only one of the nodes involved in a collision gets the supplier status. Collisions can happen in either of the following cases: (a) no node has the supplier status for the memory location (no supplier); (b) one node has the supplier status for the memory location (single supplier). Next, we show that in both cases only one of the nodes involved in the collision gets the supplier status.

No Supplier

When coherence transactions on the same memory location collide and no node has the supplier status for a memory location, the unidirectional ring protocol guarantees that any node is able to detect that its coherence transaction has collided with another coherence transaction before the node concludes its own transaction [4]. This is because the protocol requires the combined response corresponding to any coherence transaction to circulate the ring before its requester node can acquire supplier status, if the supplier status is not sent by another node.

With natural serialization as presented in Section 3.3.3, by the time the second colliding

[4] The node may prematurely access memory, but it cannot acquire supplier status or conclude the transaction before it receives a combined response.

transaction request arrives at the first node involved in the collision, that node has already received the combined response for its transaction, and a memory access is already in flight. At this point, the first node detects the collision and marks the corresponding response to cause the second transaction to retry. No further arbitration is necessary and the first node is the only node with the supplier status, which preserves the single supplier invariant.

With forced serialization as described in Section 3.3.3, by the time a node involved in the collision receives the other colliding node's request, its own response may not have arrived back. If this happens, when the node receives the request corresponding to the other colliding transaction, it detects the collision. Then, it individually uses the distributed arbitration algorithm to decide which transaction should be ordered first in the total order in case there is no supplier. The algorithm presented in Section 3.3.3 is guaranteed to always chose a single winner node to receive the supplier status. This is because algorithms 2, 3 and 4 ultimately fall back to algorithm 1 if their outcome is a tie. Algorithm 1 never ties and always chooses a single winner because it uses node ID numbers to determine node priorities, and each node has its unique ID number.

To avoid starvation when there is no supplier node, a designer can implement a strictly fair algorithm that, in addition to never tying, also determines the winner such that the distribution of winner transactions is uniform across all nodes. This can be accomplished, for example, by periodically changing the relative priorities of nodes. However, care must be taken when updating priorities because of the distributed nature of the arbitration algorithm and the fact that only one colliding transaction should be declared the winner. The key is to maintain the invariant that only one transaction wins on any collision.

Perhaps a simpler approach is to count on the anti-starvation policy used for the case in which there is a supplier node, as will be described in Section 3.5. The reason is that, after a collision when there is no supplier node, one of the transactions is the winner, accesses memory and becomes the new supplier. As long as the system guarantees that this memory location is not evicted or downgraded from the new supplier node before the supplier status is

transferred to the starving node, a supplier node is present and therefore the policy described in Section 3.5 can be used. In order to guarantee that the supplier status is not evicted from a cache, the starving node needs to notify all other nodes in the system. The starving node can accomplish this by marking every response on the same address it processes, notifying the node that the memory address should not be evicted from its cache, but saved or forwarded to the starving node instead.

Single Supplier

When coherence transactions on the same memory location collide and a node has supplier status for that memory location, the total order of these transactions is determined by which transaction's request reaches the supplier node first. We examine the case for natural and forced serialization and show that both preserve the single supplier invariant.

With natural serialization, as shown in Section 3.3.3, by the time the second colliding transaction request arrives at the first node involved in the collision, that node has already received the combined response for its transaction and therefore reached the supplier node before the request corresponding to the second colliding transaction. If this first node receives the second node's request after it receives the supplier status and the data, it can immediately supply the data and transfer the supplier status to the second node, as if the two transactions had not collided. Otherwise, the first node detects the collision and marks the corresponding response to cause the second node to retry its transaction. The first node keeps the supplier status. In either case, the single supplier invariant is maintained because no two nodes hold the supplier status simultaneously.

With forced serialization, as described in Section 3.3.3, no response may have arrived back at its corresponding requester by the time the nodes involved in the collision receive each other's colliding transaction's request. For such cases, the node whose request arrives first at the initial supplier gets the supplier status. By the time the request of the second transaction is processed by the initial supplier node, this node is no longer the supplier (it has

sent its supplier status to the first node), so the second node does not get the supplier status. In addition, in order to preserve the single supplier invariant, the second node involved in the collision needs to retry its transaction instead of accessing memory and acquiring the supplier status.

This is accomplished with the aid of the unidirectional ring partial ordering properties. Each node detects a collision when it receives a message corresponding to the other node's transaction. The second node detects it should retry its transaction when it receives the positive response corresponding to the first node's transaction or when it receives a combined response corresponding to its own transaction marked for retry by the first node. The first node detects it should mark the second node's combined response for retry when it receives its own positive combined response after it has received the second node's request but before it has received the second node's combined response.

If the relative ring order among the nodes is the first node involved in the collision, then the supplier node, then the second node involved in the collision, by the time the positive response corresponding to the first node's transaction gets to the second node, it is already positive, indicating that the first node will receive the supplier status. The second node detects that it should retry at this point, and the single supplier invariant is preserved.

If the relative ring order is, instead, the second node involved in the collision, then the supplier node, then the first node involved in the collision, the only situation in which this will not result in a case of natural serialization is if the request of the second colliding node passes the response of the first colliding node, causing the first node to receive the second node's request before it receives its own response. In this case, when the first node receives the second node's request, it detects the collision; when it receives its own positive response, it detects its transaction is the winner transaction and that the second transaction should be retried; when it receives the other node's response, it marks this response to cause the second node to retry. Again, the single supplier invariant is maintained.

3.5 Forward Progress

Forward progress can be understood in two different manners: system forward progress and node forward progress (which we call starvation freedom). It is very easy to see that an embedded-ring protocol as we describe in Section 3.3 guarantees system forward progress. On any collision, at least one of the requests is guaranteed to be successful, and therefore, even in face of collisions, the system keeps making forward progress. In rare pathological cases, however, the system could enter a retry pattern that ends up causing a particular node to always retry, resulting in node starvation. Because these cases are rare, many systems do not provide node starvation freedom guarantees. For completeness, we provide a description of how node starvation freedom can be guaranteed by an embedded-ring protocol. The key is to prevent nodes other than the starving node to reach the supplier node before the starving node's request does.

Starvation-free Embedded-ring Protocol

In order to guarantee that a starving node can eventually complete its coherence transaction, this node can intercept conflicting requests that pass by, either delaying them or forcing them to retry. When requests are delivered using a unidirectional ring, many of the conflicting requests may be intercepted by the starving node before reaching the supplier, but some might not be. These are requests issued by nodes between the starving node and the supplier node in ring order. These requests reach the supplier node before the starving node's request does, causing the supplier to process them before processing the request issued by the starving node, which has to retry. A careful look shows that nodes issuing requests not intercepted by the starving node are closer in the ring to the starving node than the initial supplier node itself. Therefore, the new supplier node will be closer to the starving node than before. Eventually, the supplier node will be the starving node's neighbor on the ring, and the starving node can finally complete its transaction because it will be able to intercept

all other conflicting requests. If there are many starving nodes, they are serviced in reverse ring order starting at the supplier node.

3.6 Consistency Implications

Embedded-ring protocols optimize cache-to-cache transfers by allowing load operations to consume the requested data as soon as it becomes available, even before a read transaction is complete. This is possible because the protocol guarantees that a read transaction will not be retried if data has already been provided to the requesting node. However, in order to complete store operations, it is necessary to wait for the response to traverse the entire ring in order to guarantee that all copies of the memory location being modified have been invalidated. As the ring scales, the store operation latency grows with the ring size. This is not a significant problem with relaxed consistency models like the one we assume because stores retire right after they are issued, and wait on a write buffer until they are complete. As long as the write buffer is large enough, and the workload does not have a pathological fence behavior, store completion latency is not in the application's critical path. However, this could affect performance in stricter consistency models because, with some, stores cannot retire before they are complete, which means the ring latency is in the critical path of the application. Techniques such as the one proposed by Gniady et al. [15], Ceze et al. [9] and Wenisch et al. [43] can help mitigate this inconvenience.

3.7 Potential Optimizations

3.7.1 Read Transactions Do Not Move Supplier Status

Although we explain the protocol operation assuming that the supplier status is transferred on every successful transaction, the supplier could keep its supplier status when it services read transactions, instead of sending it to the requester.

This allows multiple simultaneous read transactions on the same memory location, increasing read transaction parallelism, which can be beneficial in case of contended memory locations. This optimization has one unintuitive result. When a read and a write transaction collide and the read transaction is the winner, the node that issues the write transaction will both obtain the data and supplier status from the supplier, and have to retry its transaction (because the reader still needs to be invalidated).

3.7.2 Servicing Other Nodes Before Combined Response Arrives

An embedded-ring protocol could allow nodes to service other transactions after data and supplier status have been received, but before the corresponding combined response is received. This is beneficial because, by doing that, a node could avoid forcing other transactions to retry, servicing them instead.

While a node has an outstanding read transaction, it may supply data and supplier status to another node requesting a copy of the same memory location with no implications on the consistency model because it is not modifying the data.

However, for a node that has an outstanding write or invalidate transaction on a memory location, this same optimization could break stricter consistency models. This is because a node would be able to observe the new value of a memory location while other nodes (the ones that still have not completed their local snoop operations and invalidated their own copies of the data) could still observe the old value of that same memory location. Relaxed memory models can still use this optimization, as long as special care is taken with fences. For a fence operation to complete, the fence operation needs to reach the head of the reorder buffer (i.e., all previous load instructions must have retired) and every outstanding write and invalidate transaction has to complete (i.e., a node needs to wait for responses corresponding to each of the outstanding transactions to arrive back without being retried before the fence operation completes).

Chapter 4

Flexible Snooping: Adaptive Forwarding and Filtering of Snoop Requests

As already mentioned in Chapter 1, the main drawback of the embedded-ring approach is that snoop requests may suffer long latencies or induce many snoop messages and operations, which may consume excessive energy. This is the case for Lazy and Eager Forwarding, respectively. In this chapter, we propose *Flexible Snooping*, a set of Adaptive Forwarding and Filtering algorithms that reduce miss service latencies while using energy sparingly.

Section 4.1 describes Lazy and Eager Forwarding and introduces Adaptive Forwarding and Filtering algorithms, as well as available primitives for adaptation. Section 4.2 presents the design space and implementation of the algorithms. Finally, Section 4.3 discusses related work.

4.1 Toward Flexible Snooping

4.1.1 Lazy and Eager Forwarding

One of the basic forwarding algorithms traditionally used in ring multiprocessors is *Lazy Forwarding* or *Lazy*. The actions of a snoop request in this algorithm are shown in Figure 4.1(a). In the figure, a requester node sends a request that causes a snoop operation at every node until it reaches the supplier node. After that, the message proceeds without causing any other snoop operations until it reaches the requester.

Lazy has two limitations: the long latency of snoop requests and the substantial number of snoop operations performed. The first limitation slows down program execution because

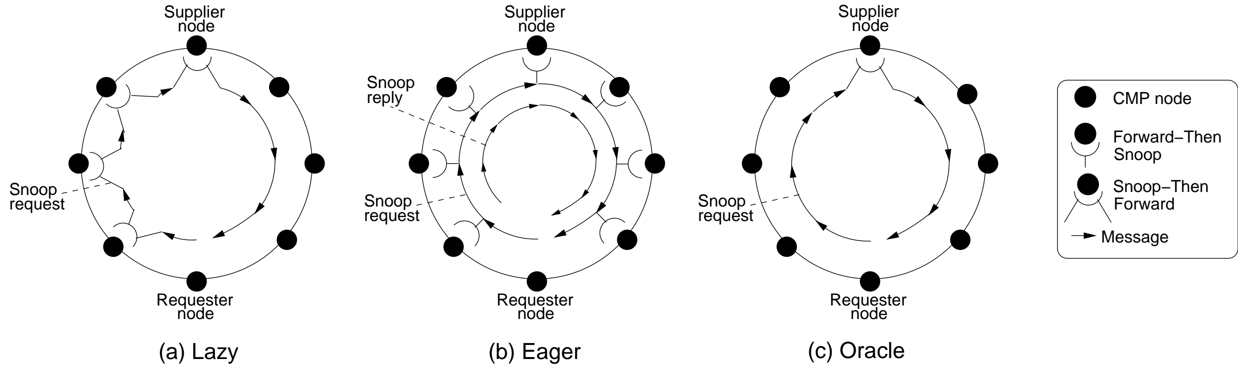


Figure 4.1: Actions of a snoop request in three different algorithms.

copies of memory locations take long to be obtained; the second one results in high energy consumption and may also hinder performance by inducing contention in the nodes.

Table 4.1 shows the characteristics of Lazy. If we assume a perfectly-uniform distribution of the accesses and that one of the nodes can supply the data, the supplier is found half-way through the ring. Consequently, the number of snoop operations is (N-1)/2, where N is the number of nodes in the ring.

Algorithm	Snoop Request Latency (Unloaded Machine)	Avg. # of Snoop Operations per Snoop Request	Avg. # of Messages per Snoop Request
Lazy Forwarding	High	(N-1)/2	1
Eager Forwarding	Low	N-1	≈ 2
Oracle	Low	1	1
Adaptive Forwarding & Filtering	Between Oracle and Lazy	Between Oracle and Eager	Between Oracle and Eager

Table 4.1: Comparing different snooping algorithms. The table assumes a perfectly-uniform distribution of the accesses and that one of the nodes can supply the data. *N* is the number of nodes in the machine.

An alternative is to forward the snoop request from node i to node $i+1$ before starting the snoop operation on i. We call this algorithm *Eager Forwarding* or *Eager*. It is used by Barroso and Dubois for a slotted ring [5]. For the non-slotted, embedded ring that we consider, we slightly change the implementation to the one Section 3.1 describes.

When a snoop request arrives at a node, it is immediately forwarded to the next one, while the current node initiates a snoop operation. When the snoop operation completes, the

outcome is combined with a snoop response message coming from the preceding nodes in the ring, and is then forwarded to the next node (Figure 4.1(b)). Some temporary buffering may be necessary for the incoming combined response or for the outcome of the local snoop. All along the ring, we have two messages: a snoop request that moves ahead quickly, initiating a snoop at every node, and a snoop response that collects all the snoop operation outcomes.

Table 4.1 compares Eager to Lazy. Eager reduces snoop request latency. Since the snoop operations proceed in parallel with request forwarding, the supplier node is found sooner, and the data is returned to the requester sooner. However, the disadvantages of Eager are that it causes many snoop operations and messages. Indeed, Eager snoops all the nodes in the ring (N-1). Moreover, what was one message in Lazy, now becomes two — except for the first ring segment (Figure 4.1(b)). These two effects increase the energy to service a snoop request.

A third design point is the *Oracle* algorithm of Figure 4.1(c). In this case, we know which node is the supplier and only perform a snoop operation there. As shown in Table 4.1, Oracle's latency is low and there is a single snoop operation and message.

4.1.2 Adaptive Forwarding and Filtering

We would like to develop snooping algorithms that have the low request latency of Eager, the few messages per request of Lazy, and the few snoop operations per request of Oracle. Toward this end, we propose *Adaptive Forwarding and Filtering* algorithms (*Adaptive*). These algorithms use two techniques. First, when a node receives a snoop request, depending on the likelihood that it can provide a copy of the requested memory location, it will perform the snoop operation first and then the request-forwarding operation or vice-versa (Adaptive Forwarding). Second, if the node can prove that it will not be able to provide a copy of the requested memory location, it will skip the snoop operation (Adaptive Filtering).

Adaptive forwarding is original. Adaptive filtering has been proposed in schemes such as JETTY [30] and Destination-Set Prediction [24], but our proposal is the first to integrate it

Primitive	Action				
	Snoop Request or Combined R/R Arrives at Node	Can Supply Line		Node Cannot Supply Line	
		Snoop Operation Completes at Node (if applicable)	Snoop Response Arrives at Node (if applicable)	Snoop Operation Completes at Node (if applicable)	Snoop Response Arrives at Node (if applicable)
Forward Then Snoop	Forward snoop request, then snoop	Send snoop response	Discard snoop response	If had received combined R/R then send snoop response else wait for snoop response	Forward snoop response
Snoop Then Forward	Snoop	Send combined R/R	Discard snoop response	If had received combined R/R then send new combined R/R else wait for snoop response	Forward it as combined R/R
Forward	Forward	N/A	Forward	N/A	Forward

Table 4.2: Actions taken by each of the Flexible Snooping primitive operations *Forward Then Snoop*, *Snoop Then Forward*, and *Forward*. In the table, R/R stands for Request/Response. Recall that at most one node has supplier status for a memory location.

with adaptive forwarding in a taxonomy of adaptive algorithms.

Adaptive hopes to attain the behavior of Oracle. In practice, for each of the metrics listed in Table 4.1, *Adaptive* will exhibit a behavior that is somewhere between Oracle and the worst of Lazy and Eager for that metric.

Adaptive works by adding a hardware *Supplier Predictor* at each node that predicts if the node has a copy of the requested memory location in any of the supplier states (S_G, E, D, and T). When a snoop request arrives at a node, this predictor is checked and, based on the prediction, the hardware performs one of three possible primitive operations: *Forward Then Snoop*, *Snoop Then Forward*, or *Forward*.

Table 4.2 describes the actions taken by these primitives. At a high level, *Forward Then Snoop* divides a snoop message into a Snoop Request sent before initiating the snoop operation and a Snoop Response sent when the current node and all its predecessors in

the ring have completed the snoop. On the other hand, *Snoop Then Forward* combines the incoming request and response messages into a single message issued when the current snoop completes. We call this message *Combined Request/Response (R/R)*. Consequently, in a ring where different nodes choose a different primitive, a snoop message can potentially be divided into two and recombined multiple times. In all cases, as soon as the requested memory location is found in a node that has the supplier status, a copy of the memory location is sent to the requester through regular network paths.

Consider *Forward Then Snoop* in Table 4.2. As soon as the node receives a snoop request or a combined R/R, it forwards the snoop request and initiates a snoop operation. When the node completes the snoop operation, if it can supply a copy of the memory location, the node sends the requested data to the requester node through regular network paths; regardless of the snoop outcome, if the node had received a combined R/R message, it immediately combines the snoop outcome to the response and sends it on the ring. The node waits for a response message if it had received only a request. Overall, the node will always send two messages, a snoop request and a snoop response.

On the other hand, with *Snoop Then Forward*, when a node receives a snoop request or a combined R/R, it starts a snoop operation without forwarding the request. When the node completes the snoop operation, if it can supply a copy of the memory location, the node sends the requested data to the requester node through regular network paths, just like before; again, regardless of the snoop outcome, if the node had received a combined R/R message, it immediately combines the snoop outcome to the R/R message and sends it on the ring. The node waits for a response message if it had received only a request, augments the response as usual and forwards both as a combined R/R. Overall, the node will always send a single message, namely a combined R/R.

The *Forward* primitive simply forwards the two messages (snoop request and response) or one message (combined R/R) that constitutes the received message.

It is possible to design different *Adaptive* algorithms by simply choosing between these

three primitives at different times or conditions. In the following, we examine the design space of these algorithms.

4.2 Algorithms for Flexible Snooping

4.2.1 Design Space

To understand the design space for these algorithms, consider the types of supplier predictors that exist (Table 4.3). One type is predictors that keep a *strict subset* of the lines that are in supplier states in the node. These predictors have no false positives, but may have false negatives. They can be implemented with a cache tag structure. We call the algorithm that uses these predictors *Subset*.

A second type of predictors are those that keep a *strict superset* of the supplier lines. Such predictors have no false negatives, but may have false positives. They can be implemented with a form of hashing, such as a Bloom filter [6]. We call the algorithm that uses these predictors *Superset*. Predictors of this type have been used in JETTY [30] and RegionScout [29] to save energy in a broadcast-based multiprocessor.

The third type of predictors are those that keep the *exact set* of supplier lines. They have neither false positives nor false negatives. They can be implemented with an exhaustive table. We call the algorithm that uses these predictors *Exact*. Note that there is a fourth type of predictors, namely those that suffer both false positives and false negatives. These predictors are uninteresting because they are less precise than all the other types, while those that suffer either false positives or false negatives are already reasonably inexpensive to implement.

Table 4.4 compares these algorithms in terms of latency of snoop requests, number of snoop operations, snoop traffic, and implementation difficulty. In the following, we examine them in detail (Section 4.2.2) and then present implementations (Section 4.2.3).

4.2.2 Description of the Algorithms

<table>
<tr><th colspan="2">Algorithm</th><th>False Pos?</th><th>False Neg?</th><th>Action If Predict Positive</th><th>Action If Predict Negative</th></tr>
<tr><td colspan="2">Subset</td><td>N</td><td>Y</td><td>Snoop Then Forward</td><td>Forward Then Snoop</td></tr>
<tr><td rowspan="2">Superset</td><td>Con</td><td rowspan="2">Y</td><td rowspan="2">N</td><td>Snoop Then Forward</td><td>Forward</td></tr>
<tr><td>Agg</td><td>Forward Then Snoop</td><td>Forward</td></tr>
<tr><td colspan="2">Exact</td><td>N</td><td>N</td><td>Snoop Then Forward</td><td>Forward</td></tr>
</table>

Table 4.3: Proposed Flexible Snooping algorithm actions.

We first consider the *Subset* algorithm (Table 4.3). On a positive prediction, since the supplier is guaranteed to be in the node, the algorithm uses *Snoop Then Forward*. On a negative prediction, since there is still chance that the supplier is in the node, the algorithm selects *Forward Then Snoop*. The latency of the snoop requests is low because requests are not slowed down by any snoop operation as they travel from the requester to the supplier nodes. The number of snoops is equal or higher than in Lazy because at least all the nodes up to the supplier are snooped. In addition, if the supplier node is falsely predicted negative, more snoops occur on subsequent nodes. Consequently, Table 4.4 shows that the number of snoops is $Lazy \times (1 - FN) + Eager \times FN$. The number of messages per snoop request is between 1 and 2 because negative predictions produce 2 messages but the positive prediction combines them.

Figure 4.2(a) shows, in a shaded pattern, the design space of Flexible Snooping algorithms in a graph of snoop request latency versus the number of snoop operations per request. The figure also shows the placement of the baseline algorithms. Figure 4.2(b) repeats the figure and adds the data points corresponding to the algorithms proposed. Based on the previous

Algorithm		Characteristics			
		Snoop Request Latency (Unloaded Machine)	Avg # Snoop Operations per Snoop Request	Avg # Msgs per Snoop Request	Implementation Difficulty
Subset		Low	Medium $(Lazy \times (1 - FN) + Eager \times FN)$	1-2	Low
Superset	*Con*	Medium	Low $(1 + (Lazy - 1) \times FP)$	1	Medium
	Agg	Low	Low $(1 + (Eager - 1) \times FP)$	1-2	Medium
Exact		Low	1	1	Medium

Table 4.4: Proposed Flexible Snooping algorithms characterization. This characterization assumes that one of the nodes supplies the data and, therefore, the request does not go to memory. In the table, FN and FP stand for the ratio of false negatives and false positives over all predictions, respectively.

discussion, we place *Subset* in the Y axis and above Lazy.

Consider the *Superset* algorithm in Table 4.3. On a negative prediction, since there are no false negatives, it uses *Forward*. On a positive prediction, since there is still a chance that the supplier is not in the node, we have two choices. A conservative approach (*Superset Con*) assumes that the node has the supplier and performs *Snoop Then Forward*. An aggressive approach (*Superset Agg*) performs *Forward Then Snoop*. The latency of the snoop requests is medium in *Superset Con* because false positives introduce snoop operations in the critical path of getting to the supplier node; the latency is low in *Superset Agg* because requests are not slowed down by any snoop operation as they travel to the supplier node. The number of snoops is low in both algorithms: 1 plus a number proportional to the number of false positives. Such term is lower in *Superset Con* than in *Superset Agg*: *Superset Con* only checks the supplier predictor in the nodes between the requester and the supplier, while *Superset Agg* checks the supplier predictor in all the nodes. The number of messages per snoop request is 1 in *Superset Con* and 1-2 in *Superset Agg*. Based on this analysis, we place

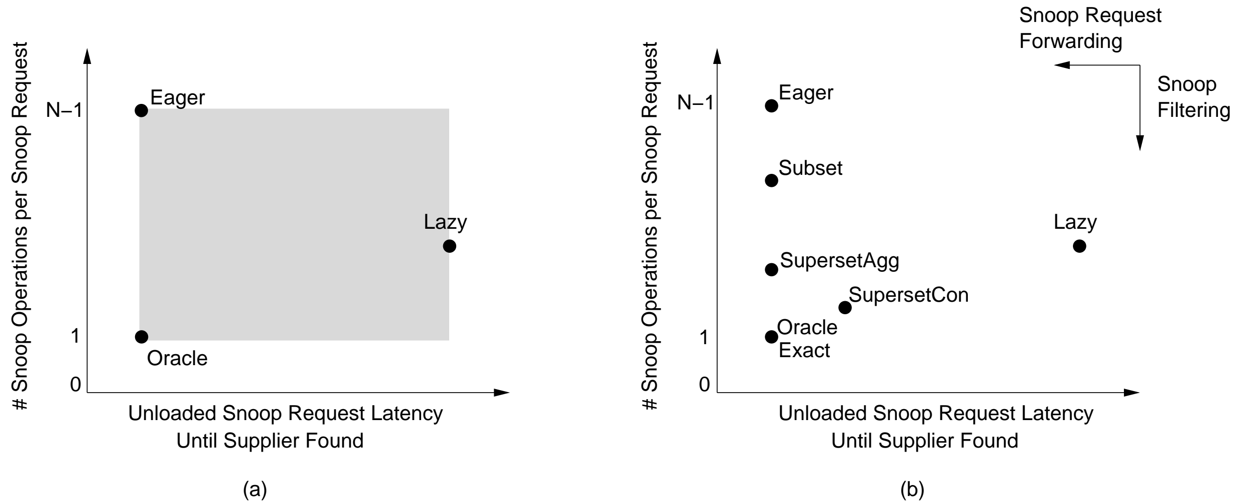


Figure 4.2: Design space of Flexible Snooping algorithms according to the snoop request latency and the number of snoop operations per request. Chart (a) shows the baseline algorithms, with the shade covering the area of Flexible Snooping algorithms. Chart (b) places the proposed algorithms. The charts assume that one of the nodes provides the line.

these algorithms in the design space of Figure 4.2(b).

The *Exact* algorithm uses *Snoop Then Forward* on a positive prediction and *Forward* in a negative one (Table 4.3). Since it has perfect prediction, the snoop request latency is low and the number of snoops and messages is 1. Figure 4.2(b) places it in the origin with the Oracle algorithm.

4.2.3 Implementation of the Algorithms

The proposed snooping algorithms enhance each node with a hardware-based supplier predictor. When a snoop request arrives at the node, the predictor is checked. The predictor predicts whether the node contains the requested memory location in any of the supplier states (S_G, E, D, or T). Based on the predictor's outcome and the algorithm used, an action from Table 4.3 is taken. Next, we describe possible implementations of the supplier predictors we study.

Subset Algorithm

The predictor for *Subset* can be implemented with a set-associative cache that contains the addresses of memory locations known to be in supplier states in the node (Figure 4.3(a)). When the copy of a memory location is brought to the node with supplier status, its address is inserted into the predictor. If possible, the address overwrites an invalid entry in the predictor. If all the lines in the corresponding set are valid, the LRU one is overwritten. This opens up the possibility of false negatives.

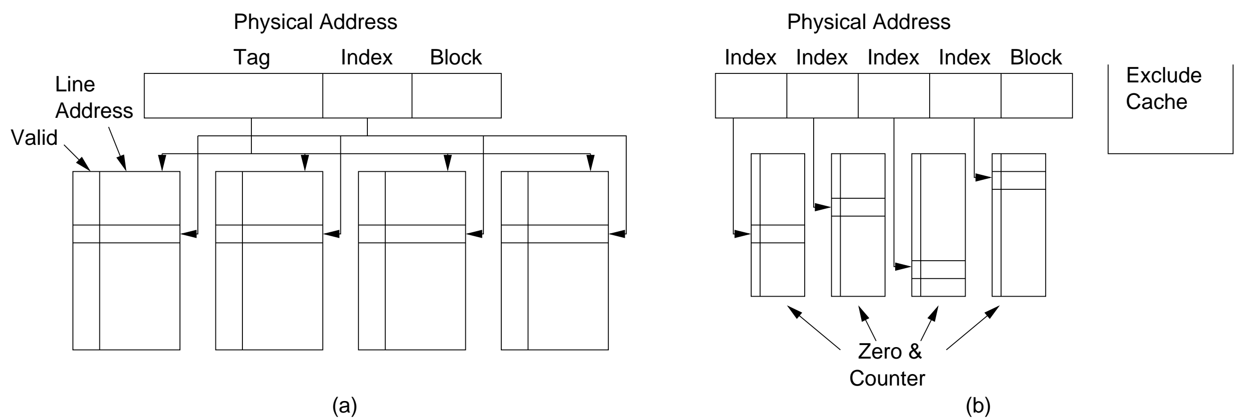


Figure 4.3: Implementation of the supplier predictors.

A line in a supplier state in the node can only lose its state if it is evicted, downgraded or invalidated. Note that, at any time, only one copy of a given line can be in a supplier state. Consequently, when any of these lines is evicted, downgraded or invalidated, the hardware removes the address from the supplier predictor if it is there. This operation eliminates the possibility of false positives.

Superset Algorithm

The predictor for the two *Superset* algorithms of Table 4.3 can be implemented with a Bloom filter [6] (Figure 4.3(b)). The Bloom filter is implemented by logically breaking down the line address into P fields. The bits in each field index into a separate table. Each entry in each

table has a count of the number of lines in a supplier state in the node whose address has this particular bit combination. Every time a line in supplier state is brought into the node, the corresponding P counters are incremented; when one such line is evicted, downgraded or invalidated, the counters are decremented. When a snoop request is received, the address requested is hashed and the corresponding entries inspected. If at least one of the counters is zero, the address is guaranteed not to be in the node in a supplier state. However, due to aliasing, this scheme can incur false positives.

To reduce the number of false positives, we can follow the JETTY design [30] and augment the Bloom filter with an Exclude cache. The latter is a set-associative cache storing the addresses of memory locations that are known not to be in supplier states in the node. Every time a false positive is detected, the corresponding address is inserted in the Exclude cache. Every time a line in supplier state is brought into the node, the Exclude cache is checked and potentially one of its entries is invalidated. With this support, when a snoop request is received, its address is checked against both the Bloom filter and the Exclude cache. If one of the counters in the filter is zero or the address is found in the Exclude cache, the prediction is declared negative.

Exact Algorithm

The predictor for *Exact* can be implemented by enhancing the *Subset* design. We eliminate false negatives as follows: every time that a valid entry in the supplier predictor is overwritten due to a conflict, the hardware downgrades the supplier state of the corresponding line in the node to a non-supplier state. Specifically, if the line is in S_G or E, it is silently downgraded to S_L; if the line is in D or T state, the line is written back to memory and kept cached in S_L state.

This support eliminates false negatives but can hurt the performance of the application. Specifically, a subsequent snoop request for the downgraded line from any node will have to be serviced from memory. Moreover, if the downgraded cache attempts to write one of the

lines downgraded from E, D, or T, it now needs one more network transaction to upgrade the line to its previous state again before the write can complete.

Overall, the performance impact of this implementation depends on two factors: the size of the predictor table relative to the number of *supplier lines* in a node, which affects the amount of downgrading, and the program access patterns, which determine whether or not the positive effects of downgrading dominate the negative ones.

Difficulty of Implementation

We claim in Table 4.3 that *Superset* and *Exact* are more difficult to implement than *Subset*. The reason is that these algorithms have no false negatives, whereas *Subset* has no false positives. To see why this matters, assume that a hardware race induces an unnoticed false negative in *Superset* and *Exact*, and an unnoticed false positive in *Subset*. In the first case, a request skips the snoop operation at the node that has the line in supplier state; therefore, execution is incorrect. In the second case, the request unnecessarily snoops a node that does not have the line; therefore, execution is slower but still correct.

Consequently, implementations of *Superset* and *Exact* have to guarantee that no hardware race ever allows a false negative to occur. Such races can occur at two time windows. The first window is between the time the node receives a line in supplier state and the time the *Superset* or *Exact* predictor tables are updated to include the line. Note that the line may be received from local memory (Figure 3.2(a)) or through other network links that do not go through the gateway. The second race window is between the time the *Exact* predictor table removes an entry due to a conflict and the time the corresponding line in the node is downgraded. Careful design of the logic involved will ensure that these races are eliminated.

4.3 Related Work

Our work is related to several schemes that improve performance or energy consumption in coherence protocols. In Destination-Set Prediction [24], requester caches predict which other caches need to observe a certain memory request. Unlike our proposal, the prediction is performed at the source node, rather than at the destination node. Moreover, destination-set prediction targets a multicast network environment. It leverages specific sharing patterns like pairwise sharing to send multicasts to only a few nodes in the system.

JETTY [30] is a filtering proposal targeted at snooping bus-based SMP systems. A data presence predictor is placed between the shared bus and each L2 cache, and filters part of the snoops to the L2 tag arrays. The goal of the mechanism is exclusively to save energy. While we used one of the structures proposed by JETTY, our work is more general: we leverage snoop forwarding in addition to snoop filtering; we use a variety of structures; we use these techniques to improve both performance and energy; and finally we use a supplier predictor.

Power Efficient Cache Coherence [31] proposes to perform snoops serially on an SMP with a shared hierarchical bus. Leveraging the bus hierarchy, close-by caches are snooped in sequence, potentially reducing the number of snoops necessary to service a read miss. Our work is different in the following ways: our work focuses on a ring, on which we detail a race-free coherence protocol; we present a family of adaptive protocols; finally we focus on both high performance and low energy.

Ekman et al. [14] evaluate JETTY and serial snooping in the context of a cache-coherent CMP (private L1 caches and shared L2 cache) and conclude these schemes are not appropriate for this kind of environment.

Owner prediction has been used to speed-up cache-to-cache interventions in a CC-NUMA architecture [1]. The idea is to shortcut the directory lookup latency in a 3-hop service by predicting which cache in the system would be able to supply the requested data and sending

the request directly to it, only using the home directory to validate the prediction.

Barroso and Dubois [5] propose the use of a slotted ring as a substitute for a bus in an SMP system. As indicated in Section 2.3.2, their work is different in that they look at a ring network topology (while we use a logically-embedded ring) and that they use slotting (while we do not have these timing constraints). They use the Eager algorithm, which we use as a baseline here. Another system that uses a slotted ring topology is Hector [41]. Hector uses a hierarchy of rings.

Moshovos [29] and Cantin et al. [7] propose coarse-grain mechanisms to filter snoops on memory regions private to the requesting node. They differ from our work in that they are source-filtering mechanisms. In addition, these mechanisms work at a coarser granularity and target only a certain category of misses (cold misses or misses to private data). These techniques may be combined with our techniques to further improve performance and energy savings.

Chapter 5

Unconstrained Snoop Request Delivery

As mentioned in Chapter 1, one fundamental shortcoming of using a unidirectional ring to forward all snoop requests and responses is that forcing snoop requests to traverse the ring sequentially constrains the overlap of snoop operations in a coherence transaction and results in long miss latencies.

In order to reduce the request delivery latency, we propose using *any* network links for request delivery, and using the embedded unidirectional ring only to collect snoop responses, which are often off the critical path. This avoids the sequential request delivery imposed by the unidirectional ring, allowing further overlap of snoop operations.

However, one concern is whether it is still possible to serialize coherence transactions properly when requests are not delivered in a sequential fashion through the ring. It turns out that we can achieve proper transaction serialization by delivering only snoop responses through the ring, and that most of the mechanisms we need for doing that are already in place in an embedded-ring multiprocessor as presented in Section 3.1. We call the mechanism we propose Unconstrained Snoop Request Delivery, or *UncoRq*.

UncoRq eliminates a fundamental performance limitation of previous embedded-ring multiprocessor proposals by allowing unconstrained request delivery, with simple modifications.

5.1 Benefits of Unconstrained Snoop Request Delivery

Figure 5.1(a) shows the difference between snoop request delivery using the *Eager Forwarding* mechanism, which we simply call *Eager*, and *UncoRq*. *Eager* uses the logical unidirectional ring to deliver snoop requests, while *UncoRq* delivers snoop requests using any path available in the underlying physical network (preferably the fastest).

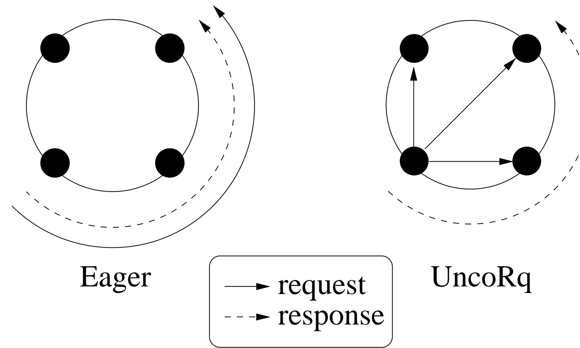
UncoRq targets read misses serviced with cache-to-cache transfers. For this type of miss, we find it useful to make a distinction between the data reception latency and the response reception latency.

Figure 5.1(b) presents the latency components for a read miss that is serviced with a cache-to-cache transfer. The top diagram shows the latency for data arrival. It includes the latency for request propagation to the supplier, the latency for the snoop at the supplier and the latency for data propagation back to the requester. Because *UncoRq*, unlike *Eager*, does not deliver requests using the ring, it allows requests to reach the supplier node sooner ((1) in Figure 5.1(b)), achieving shorter data reception latency. As soon as the processor receives the requested data, dependent instructions can be executed and retired.

The bottom diagram in Figure 5.1(b) shows the response reception latency, which is simply the time it takes for the response to traverse the ring and collect the local snoop outcome at every node. Note that for read misses that are serviced with cache-to-cache transfers, this latency is not important, even if it is longer than the data reception latency. For read misses serviced by memory, the response propagation latency matters because, many times, only after receiving the combined response and verifying that it is negative will the node send a data request to memory.

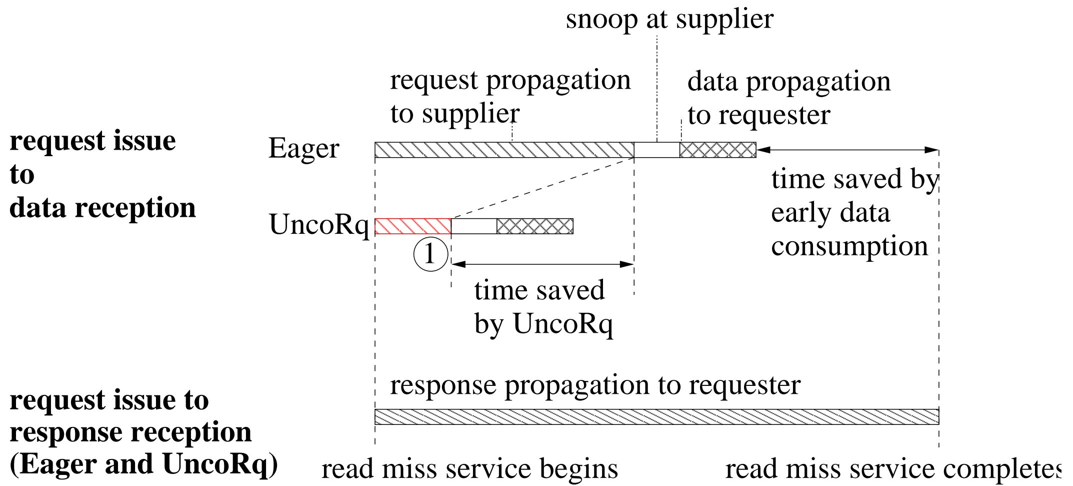
In summary, *UncoRq* benefits performance by reducing the data reception latency of misses serviced with cache-to-cache transfers. This directly reduces processor stalls.

Request and response distribution:



(a)

Latencies in cache–to–cache transfer:

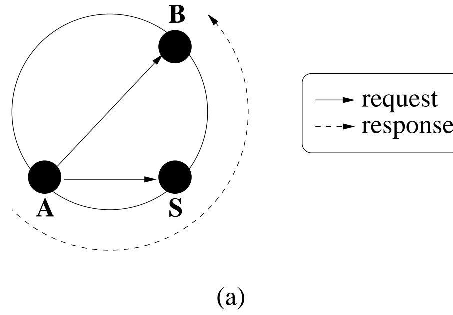


(b)

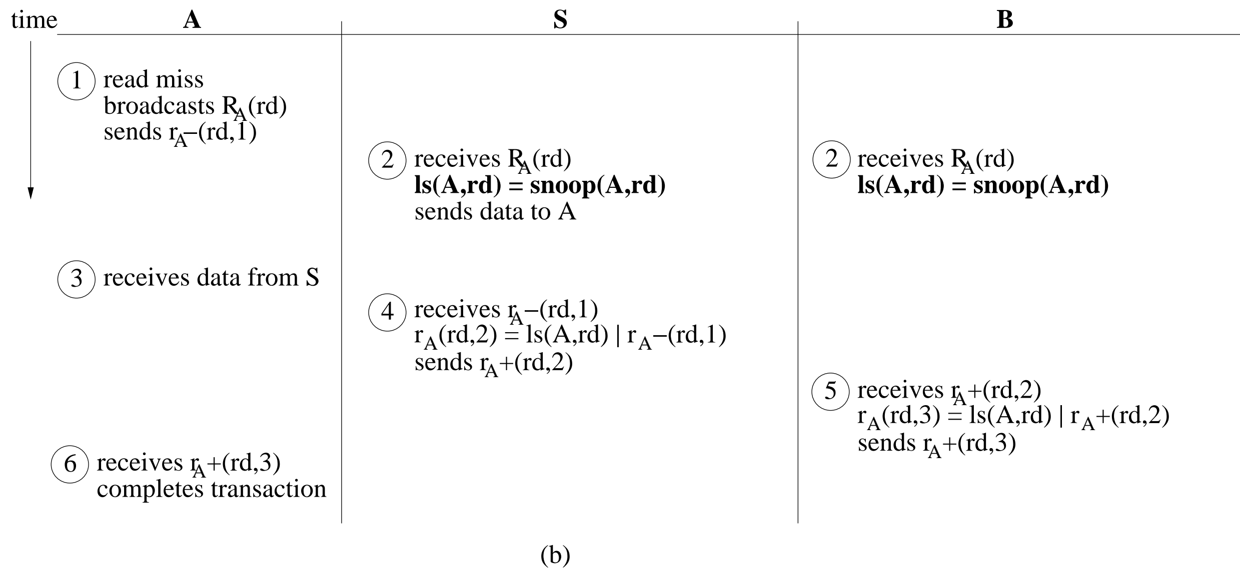
Figure 5.1: Latency components for read miss services with cache-to-cache transfers.

5.2 *UncoRq* in Action

Figure 5.2 shows how *UncoRq* services a read miss when there are no collisions. Figure 5.2(a) shows that requests are directly delivered to all nodes, and the direction the response traverses the ring. Figure 5.2(b) shows a timeline of events at each node.



(a)



(b)

Figure 5.2: Miss serviced on an embedded-ring multiprocessor with *UncoRq*.

When A suffers a read miss (1), it broadcasts a read request R_A directly to S and B, and then forwards a negative read response r_A- to S using the ring. S and B receive R_A (2), and

both initiate snoop operations simultaneously. After S completes its snoop operation and determines it can supply the requested data to A, it sends the data to A. A receives data from S (3), and is free to use the data before the coherence transaction concludes. When S receives r_A- (4), S combines its positive outcome with r_A- into r_A+ and sends it to B. Meanwhile, B has already completed its snoop operation. When B receives r_A+ from A (5), it combines its local snoop outcome with r_A+ and forwards it back to A. When A receives r_A+ (6), it completes the coherence transaction.

With *UncoRq*, a response may follow a different path and arrive at a node before the corresponding request. When this happens, the node buffers the response with no further action. Only when the node receives the request will it perform a snoop operation, combine the local snoop outcome with the response and forward a new combined response.

5.3 Supporting Invariant I_1 with *UncoRq*

Because *UncoRq* does not require requests to travel on the ring, it makes it more difficult to guarantee that messages do not get reordered in the network. If no countermeasure is taken, this could break invariant I_1. To understand why, consider the example in Figure 5.3, in which only one node C is depicted. First, C receives r_B+, even before it receives R_B. Then, C receives R_A and performs a snoop operation. After the snoop operation is complete, C receives r_A-, combines it with the local snoop outcome, and forwards it. By doing this, node C has reordered r_B+ and r_A-, which clearly breaks invariant I_1.

Invariant I_1 determined that no positive response should be overtaken by negative responses following it on the ring or through the nodes subsequent to the supplier on the ring until this positive response is removed from the ring by its requester. To maintain invariant I_1 with *UncoRq*, we present two conditions, C_2 and C_3.

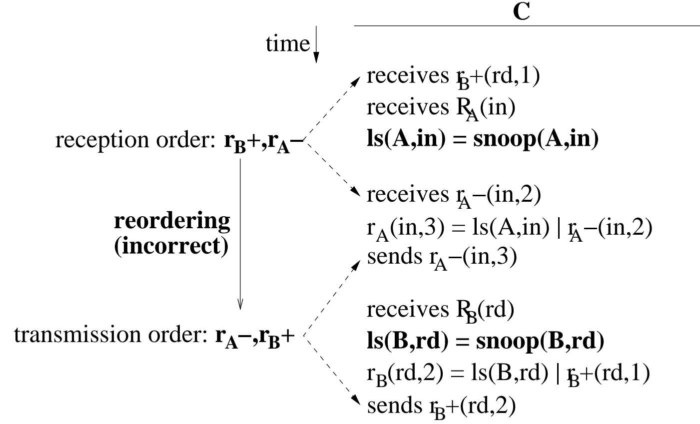


Figure 5.3: Example of *UncoRq* failure to support invariant I_1.

Condition C_2: If a supplier node processes R_I (consequently generating a positive snoop outcome), it cannot forward any response r_J that arrives at the node later until it receives response r_I- and forwards response r_I+.

Condition C_3: If a node receives a positive response r_I+, it cannot forward any response r_J- received after r_I+ until it receives R_I (if not already received) and forwards response r_I+.

These two conditions only differ in that C_2 concerns the case in which there is an outstanding positive snoop outcome in a node, and C_3 concerns the case in which there is an outstanding positive response in a node. In both cases, however, once the node starts buffering a positive snoop outcome or response, it cannot forward any subsequent responses on the same line until it forwards the corresponding positive response. If these two conditions are true, I_1 holds as well, covering the case in which there is a supplier node. Cases in which there is no supplier are covered in the same way as with the baseline embedded-ring protocol. C_2 and C_3 add a third overall condition to the response forwarding conditions presented in Section 3.1: *(3) the node has not recorded any positive snoop outcome or positive combined response involving the same memory line that is still waiting for conditions (1) and (2) to be satisfied.* With such a simple supplementary condition, *UncoRq* adds little complexity to

existing embedded-ring multiprocessors.

5.3.1 Overall Operation

The only new sub-case *UncoRq* introduces is an instance of the second case in Table 3.1, the case of forced serialization in which a supplier node is present. Table 5.1 shows the four cases, but focuses on the new sub-case.

Supplier Present, Forced Serialization In this case, A and B may observe each other's requests before observing their own responses. Since *UncoRq* no longer restricts requests to the ring, now R_B may reach S before R_A, even in the case B sends R_B after it receives R_A. In this new sub-case, since R_B reaches S before R_A does, B is allowed to conclude its transaction when it receives r_B+. B immediately identifies that A should retry its transaction, and later, when B receives r_A-, it communicates this to A using r_A-retry.

Supplier Node	Serialization	Process
Present	Natural	Same as original embedded-ring protocol.

Present	Forced	If S receives R_A first, same as original embedded-ring protocol.
		If S receives R_B first,
		    A receives R_B (no effect – A is in transient state)
		    A receives r_B-
		        A records B's ID and whether R_B is a write/invalidate
		    B receives r_B+
		        B marks A's transaction as loser
		    B receives r_A-
		        B marks r_A- to force retry – r_A-retry
		    A receives r_A-retry
		        A invalidates data if needed and retries its transaction

Not Present	Natural	Same as original embedded-ring protocol.
Not Present	Forced	Same as original embedded-ring protocol.

Table 5.1: Four different collision cases that need to be handled by *UncoRq*.

5.4 Implementation Issues

5.4.1 Enforcement of Invariants C_2 and C_3 for ***UncoRq***

To enforce the conditions described above, we use a table in each node to record information about in-flight transactions at the node. We call this table the *Local Transaction Table* (LTT). A transaction is in-flight at a node if a request or response corresponding to the transaction has been received by the node, but the node has not sent a combined response for the transaction.

The goal of the LTT is to act as a barrier for other responses when there is an in-flight transaction at the node for which it has generated a positive snoop outcome or received a positive combined response. In this case, the node cannot forward other colliding responses before the new positive combined response is generated and forwarded. With a single supplier protocol, there is no risk of deadlock because at most one node will be the source of positive snoop outcomes and positive snoop responses for a particular memory line at any time.

All simultaneous in-flight transactions at a node involving the same memory line are mapped into the same entry of the node's LTT. As Figure 5.4 shows, each entry consists of an address field, a Data Receiving Node (DRN) field, and two bit vectors, Local Vector (LV) and Remote Vector (RV), each with as many bits as nodes.

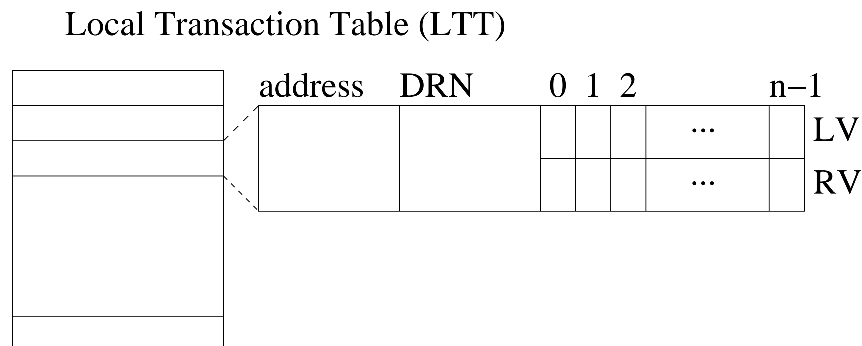


Figure 5.4: LTT: Additional structure to support *UncoRq*.

The address field stores the address of the line being requested by the transactions that

map to the entry. The DRN records the ID of an in-flight transaction requester node, if any, for which the node containing the LTT has generated a positive snoop outcome or received a positive combined response. The LV records the IDs of in-flight transaction requester nodes for which a local snoop response has been generated. The RV records the IDs of in-flight transaction requester nodes for which a combined response has been received.

When a node receives a request or response, it accesses the LTT. If an entry is not already allocated for the requested line, one is allocated. In this case, the line address is recorded in the entry, both the LV and the RV are cleared and the DRN is set to "None". If the node has received a request, the node initiates a snoop operation. If the node has received a response, it sets the bit in the RV corresponding to the requester node. In addition, if the received response is positive, the node records the ID of the corresponding requester node in the DRN. When a node completes a snoop operation, it accesses the appropriate entry in the LTT, sets the corresponding bit in the LV and, if the local snoop response is positive, the node records the requester node ID in the DRN.

According to the conditions presented in Sections 3.1 and 5.3, a node is ready to forward the combined response for a particular transaction when: (1) the corresponding bit in the LV is set, meaning that the node has generated a local snoop outcome for that transaction; (2) the corresponding bit in the RV is set, meaning that the node has received a combined response for that transaction; and (3) DRN is "None", meaning that the node has not recorded any positive snoop outcome, or DRN is equal to the ID of the node that issued the transaction, meaning that the node has recorded a positive outcome or response for that specific transaction.

When a response is forwarded, the corresponding bits in the LV and RV are cleared. If the ID of the requester node was that in the DRN (which is the case only for positive responses), the DRN is set to "None". When all bits in LV and RV are cleared, the entry is recycled.

To see how the mechanism works, consider Figure 5.5. When C receives r_B+, it allocates

an entry in the LTT, recording the line address, setting the RV bit corresponding to B, and setting the DRN to B. When C receives R_A, it initiates a snoop operation. When the snoop operation completes with a negative snoop outcome, C sets the LV bit corresponding to A. When C receives r_A-, it sets the RV bit corresponding to A and verifies that the LV bit is already set. At this point, C checks whether the ID in the DRN is A. Since it is not, C stalls r_A-. Eventually, C receives R_B, at which point it initiates another snoop operation. When this second snoop operation completes with a negative snoop outcome, C sets the LV bit corresponding to B, verifies that the RV bit is already set, and checks whether the ID in the DRN is B. Since it is, C sends r_B+ to the next node, and resets both the LV and RV bits corresponding to B, as well as the DRN. After that, C checks if there are any stalled responses. C finds r_A- and immediately forwards it as well, reseting the corresponding LV and RV bits. This guarantees that r_B+ and r_A- do not get reordered, preserving the proper serialization of transactions.

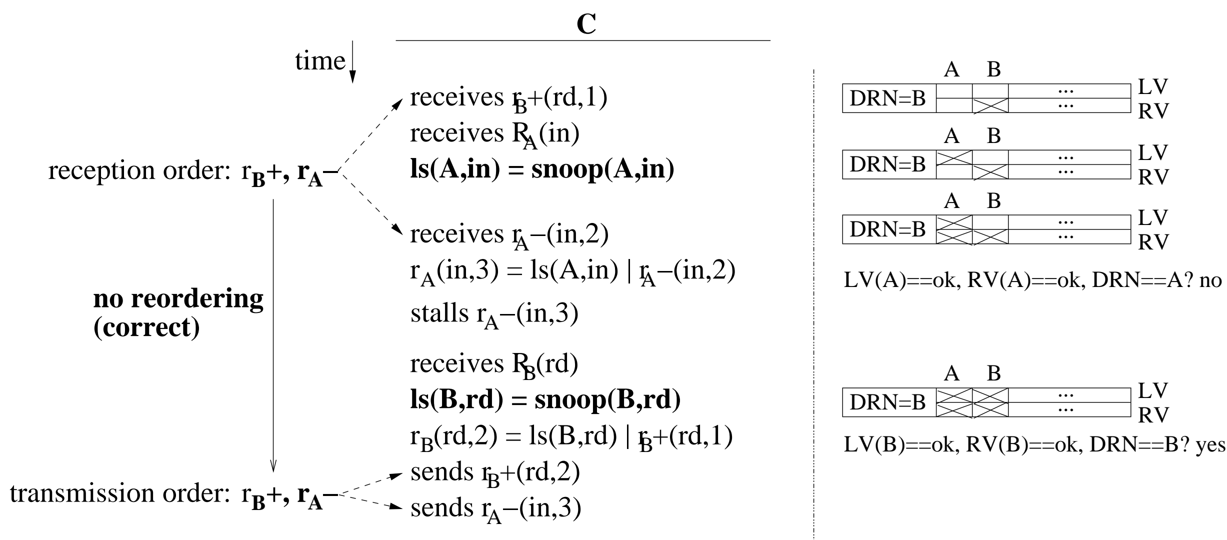


Figure 5.5: Example of LTT supporting conditions C_2 and C_3 for correct operation of *UncoRq*.

LTTs are obviously of limited size. In order to support all outstanding transactions in a multiprocessor, a designer may choose one of two strategies. Conceptually, the simplest

solution is to use associative structures as large as the maximum number of outstanding transactions (number of nodes multiplied by maximum number of outstanding transactions per node). These structures also need to support associativity as high as the maximum number of outstanding transactions to a particular set. Building such structures is possible up to a certain size, but as systems scale, they become bottlenecks. An alternative to that is to design LTTs for a smaller number of outstanding transactions, and use negative acknowledgment to handle rare LTT overflows.

5.4.2 Additional Optimization for *UncoRq*

UncoRq benefits read misses serviced with cache-to-cache transfers. However, cache-to-cache transfers are a fraction of all transfers, and optimizing read misses serviced with memory-to-cache transfers is desirable for applications with high global miss rates (i.e., misses for which no cache can supply data).

A very effective optimization for memory-to-cache transfers is memory prefetching. However, if memory is accessed on every cache miss, it may be too wasteful, especially in a CMP. If another cache is able to supply the data, it is beneficial to avoid the access to memory both because it saves the energy of a memory access (which is larger than the energy to snoop the CMP) and because it saves the bandwidth used by a memory access (which is a critical resource on a CMP). Next, we present a prefetching prediction mechanism to select which lines to prefetch, which also saves control messages on occasion.

The prefetching prediction mechanism works as follows: when a node suffers a read miss, it accesses its node predictor to determine whether the line may be supplied by another node. If the prediction is that it can, no prefetch is issued. Otherwise, in addition to broadcasting the request, the node also sends the request to the memory controller. The memory controller independently predicts whether a node can supply the line. If it predicts that no node can supply the line, it prefetches the line from memory.

The node predictor is simply a table in which the node records addresses of transactions

it has observed recently, together with the transaction requester. If the address is found in the table, the prediction is that the recorded node can supply the line, and a request is sent only to that node. If the prediction is correct, the supplier node sends the requested data and a positive response. If the prediction is not correct, the node incorrectly predicted as supplier sends a negative response, in which case the requester now has to broadcast the request. To implement this structure, we use a cache tag array with LRU replacement and a few additional bits. A node inserts an entry in this table on three occasions: when it observes a cache-to-cache transfer from one node to another, and when it receives an ownership request from other nodes, and when it incorrectly predicts a supplier.

The memory controller predictor is also simple. It consists of a table that records one page per entry to leverage spatial locality. Each entry records the page address and one bit per line in the page to indicate whether a cache in the system is likely to supply that line. We use a cache structure with LRU replacement to implement this structure. If the memory controller finds the page address in the table and the bit corresponding to the line is clear, a cache is not likely to supply the corresponding line, so the memory controller issues a prefetch. Otherwise, the memory controller takes no further action. When a page entry is allocated, all line bits are cleared. The memory controller trains the predictor in three situations: when a line is brought from memory, it sets the bit corresponding to the line; when a line is written-back, it clears the corresponding bit; when a prefetched line is not used, it sets the corresponding bit. The memory controller only allocates a new page entry on the first type of training event, if necessary.

5.4.3 Forward Progress

UncoRq guarantees system forward progress in the same way a baseline embedded-ring protocol does: on any collision, at least one of the requests is guaranteed to be successful, and therefore, even in face of collisions, the system keeps making forward progress. Starvation freedom is enforced using a different method, although the spirit is the same: to prevent

nodes other that the starving node to snoop the supplier node before the starving node does.

Starvation-free *UncoRq*

With *UncoRq*, since requests are not distributed using the ring, a starving node cannot intercept conflicting requests. However, we can leverage the LTT mechanism to avoid node starvation. A starving node can add its ID to each conflicting response it intercepts in the ring. If a positive response arrives at its requester (new supplier) with the ID of a starving node, the new supplier node allocates an LTT entry and sets its DRN to the starving node's ID, even before a request is received and the corresponding snoop operation is performed. The result is that the starving node is guaranteed to complete its memory operation. In case several nodes simultaneously starve, a starving node writes its ID in a response (overwriting any previously recorded ID) when: (1) the response has been issued by another starving node, (2) the response has been issued by a node that is not starving and the response is negative, (3) the response has been issued by a node that is not starving, the response is positive, and no other starving node has recorded its ID in the response, and (4) the response has been issued by a node that is not starving, the response is positive, another starving node has recorded its ID in the response and the distance between the supplier node and the other starving node in ring order is shorter than the distance between the supplier node and the current starving node. This causes starving nodes to be serviced in reverse ID order [1].

Collisions for which there is no supplier can be handled using the same strategies described in Section 3.4.1, or even using a simpler strategy. When a node receives a negative response with a starving node ID that also indicates that the requester node (future supplier node) should access memory to retrieve a copy of the requested memory location, the requester node allocates an LTT entry and sets its DRN to the ID of the starving node. If, after receiving the supplier status and the data from memory, but before processing the request from the starving node, the new supplier node has to evict that memory location, it

[1] We assume IDs are assigned in sequence along the direction of the ring.

stores the data and supplier status in a special buffer that is freed once the starving node's request is processed by that node.

5.5 Related Work

Barroso and Dubois propose the use of a slotted ring for implementing snooping cache coherence [5]. However, they propose using the ring as the physical network topology, as opposed to a logical embedded unidirectional ring as in our case. In addition, they use time slots, while we impose no timing constraints. They use an *Eager* forwarding algorithm, which we apply to an embedded ring and compare against.

In [27], Marty and Hill propose Ring-Order, a physical ring protocol that avoids transaction retries by intercepting response-and-data messages on the ring and completing transactions in ring position order. Although their proposal is suitable for physical rings, we feel it may not be profitable on an embedded-ring system. The reason is that Ring-Order requires data to follow the embedded ring, which would add long latencies to the data reception path, especially in a mid-size system like the one we target. However, if desired, *UncoRq* and Ring-Order could be combined to reduce the request delivery latency while avoiding retries.

In [13], Eisley et al. propose a directory-based coherence scheme that involves the network in the coherence process. Their goal is the same as ours: to reduce the data reception path, consequently reducing stall time. However, our understanding is that they use a request forwarding scheme similar to *Lazy*, which may add extra latency to the request path. Other concerns are considerable additional storage at every node and routing complexity.

Martin et al. [24] propose using Destination-Set Prediction to exploit sharing behavior of applications. Each proposed predictor targets a different latency/bandwidth trade-off and predicts the set of nodes that need to receive a given request. The node-side predictor we use with *UncoRq* is similar to a Destination-Set Predictor, in that its goal is to predict which

node can supply the requested data. However, there are a few differences. Our predictor records a single ID, which is the ID of the last observed supplier for a particular line. In addition, the goal of our predictor is to save energy, not to improve performance. Other techniques to improve snooping protocols, such as JETTY [30] and coarse grain coherence tracking [29, 7], can also be applied in our framework.

Kırman et al. [21] study the use of optical interconnects in a ring topology to implement a snooping shared bus cache-coherent CMP. They propose using a physical optical ring to interconnect on-chip clusters of cores. However, the ring is used to implement the highest level of a hierarchical opto-electrical shared bus, which is different from the embedded-ring mechanism we consider.

Chapter 6

Evaluation of Embedded-Ring Cache Coherence

We start this chapter by describing the experimental setup in Section 6.1. We then present our evaluation in Section 6.2, which studies all embedded-ring cache coherence techniques we propose in this thesis in two different system configurations, identifies the best ones, compares them to a high performance bus-based implementation of cache coherence, and provides a sensitivity analysis.

6.1 Experimental Setup

This section introduces the two different system configurations we simulate for our evaluation, the predictor parameters we use, as well as other details of our simulation infrastructure and assumptions.

6.1.1 Simulation Infrastructure

We evaluate embedded-ring cache coherence using an extended version of SESC, a detailed cycle-accurate simulator of out-of-order processors and memory subsystems. The memory system model we developed implements an enhanced MESI protocol, which includes support for dirty sharing and cache-to-cache transfers of clean data, as described in Section 3.2. We assume a release consistency memory model.

We run 10 of the SPLASH-2 applications [44] (all except radiosity and volrend), SPECjbb 2000 and SPECweb 2005 [34]. We skip initialization and run SPLASH-2 applications to completion, with parameters from [44], and 64 processors using SESC only. SPEC applications

are first run in Simics [40] to generate 8-processor traces. We skip initialization and then run each application for over 750 million instructions. Then these traces are replicated to total 64 processors and their addresses adjusted to emulate multiple instances of the application. Note that there is no sharing whatsoever between different instances of the trace, which gives no advantage to the techniques we propose. The traces are fed into SESC for timing simulation. SPECjbb uses an 8 warehouse configuration, SPECweb uses the e-commerce workload. Since the same trace is fed to the different configurations we compare, it is not necessary to run multiple instances of each experiment.

6.1.2 Simulated Architectures

The first baseline system architecture we simulate (*Multi-CMP* configuration) is a multiprocessor with 8 CMPs as nodes, where each CMP has 4 processors and their associated private caches (Figure 3.2(a)). The interconnect inside each CMP is a high-bandwidth shared bus. The second baseline system architecture (*Single-CMP* configuration) is a single-CMP multiprocessor with 64 nodes, each composed of a core and its associated private caches.

In both configurations, the nodes are interconnected with a 2-dimensional torus, on which we embed two unidirectional rings for snoop messages, which are assigned to rings based on their address. The cache coherence protocol used is described in Section 3.2. We provide the same aggregate network bandwidth to all designs. Tables 6.1, 6.2 and 6.3 show the architectural parameters used in our simulations for processors and each of the system configurations, respectively.

For the Multi-CMP configuration, we assume 65nm technology. We estimate that a message in the embedded ring network needs 48 cycles at 4 GHz to access the CMP bus and snoop the caches. This includes 34 cycles for on-chip network transmission (transmission from gateway to arbiter, from arbiter to L2 caches, and from L2 caches to gateway), 7 cycles for on-chip network arbitration, and 7 cycles for L2 cache snooping plus buffering. All on-chip L2 caches are snooped in parallel. These numbers are consistent with those in [22].

Processor and Private Caches	
Frequency: 4.0 GHz Branch penalty: 17 cyc (min) RAS: 32 entries BTB: 2K entries, 2-way assoc. Branch predictor: bimodal size: 16K entries gshare-11 size: 16K entries Int/FP registers: 176/130	Fetch/issue/comm width: 6/4/4 I-window/ROB size: 80/176 LdSt/Int/FP units: 2/3/2 Ld/St queue entries: 64/64 D-L1 size/assoc/line: 32KB/4-way/64B D-L1 RT: 2 cyc L2 size/assoc/line: 512KB/8-way/64B L2 RT: 10 cyc

Table 6.1: Architectural parameters used for simulated processors. In the table, RT means minimum Round-Trip time from the processor. Cycle counts are in processor cycles.

Multi-CMP Configuration		
CMP	Global Network	Memory
Number of CMPs: 16 Processors per CMP: 4 Private DL1 and IL1 caches Private L2 cache Intra-CMP network: Topology: shared bus Bandwidth: 64 GB/s RT to another L2: 48 cyc	Topology: 2D torus CMP to CMP latency: 39 cyc Embedded ring network: Link bandwidth: 8GB/s CMP bus access & L2 snoop time: 48 cyc Data network: Link bandwidth: 32GB/s	RT to local memory: 214 cyc Main memory: Frequency: 667MHz Width: 128bit DRAM bandwidth: 10.7GB/s

Table 6.2: Architectural parameters used for Multi-CMP configuration. Cycle counts are in processor cycles.

For the Single-CMP configuration, we base our evaluation on a 64-core CMP, where each core has private data and instruction L1 caches and a private L2 cache. We roughly estimated this design to fit in a large chip in 32nm technology, which is in line with other studies [21]. Some other parameters listed in Table 6.3 are based on data from the 2005 ITRS Roadmap [19] and cache parameters are calculated using CACTI 4 [12].

For both configurations, we estimate the energy consumed on coherence activity. To estimate this energy we employ several tools. We use models from CACTI [12] to estimate the energy consumed accessing cache structures (when snooping or downgrading lines) and

Single-CMP Configuration		
CMP	CMP Network	Memory
Number of CMPs: 1 Processors per CMP: 64 Private DL1 and IL1 caches Private L2 cache	Topology: 2D torus Node to Node latency: 8 cyc Embedded ring network: Link bandwidth: 16GB/s L2 snoop time: 7 cyc Data network: Link bandwidth: 64GB/s	RT to local memory: 214 cyc Main memory: Frequency: 667MHz Width: 128bit DRAM bandwidth: 10.7GB/s

Table 6.3: Architectural parameters used for Single-CMP configuration. Cycle counts are in processor cycles.

predictors. We use Orion [42] to estimate the energy consumed to access the on-chip network. We use the HyperTransport I/O Link Specification [18] to estimate the energy consumed by the transmission of messages in the ring interconnect. Finally, we use Micron's System-Power Calculator [28] to estimate the energy consumed in main memory accesses.

As an example of the numbers obtained, transferring one snoop request message on a single off-chip link is estimated to consume 3.17 nJ. In contrast, the energy of a 4-node CMP snoop is estimated to be only 0.69 nJ. Finally, reading a line from main memory is estimated to consume 24 nJ.

6.1.3 Predictors Used

Tables 6.4 and 6.5 show the parameters used in our simulations for *Flexible Snooping* and prefetching predictors, respectively.

For our experiments, we use three or four different predictors for each of our *Flexible Snooping* algorithms. The predictors used are shown in Table 6.4. The default values used for most of the evaluation are shown in bold. We chose these values such that the predictors would not result in significant area overhead.

The four predictors for *Subset* are 8-way set associative structures with either 512, 2K,

Flexible Snooping Predictors	
Subset Predictor	*Exact* Predictor
Size: 512, **2K**, 8K or 32K entries Entry size: 20, 18, 16 or 14 bits Total size: 1.3, 4.8, 17KB or 60KB Access time: 2, 2, 3 or 4 cyc Assoc: 8-way Ports: 1	Size: 512, **2K**, 8K or 32K entries Entry size: 20, 18, 16 or 14 bits Total size: 1.3, 4.8, 17KB or 60KB Access time: 2, 2, 3 or 4 cyc Assoc: 8-way Ports: 1
Superset Con or *Superset Agg* Predictor	
Bloom filter: *n* filter: Fields: 9,9,6 bits Total size: 2.3KB *y* filter: Fields: **10,4,7** bits Total size: 2.5KB Size of filter entry: 16 bits Access time: 2 cyc	Exclude cache: Size: 512 or **2K** entries Entry size: 20 or 18 bits Total size: 1.3 or 4.8KB Access time: 2 cyc Assoc: 8-way Ports: 1

Table 6.4: Architectural parameters used for *Flexible Snooping* predictors. Cycle counts are in processor cycles.

8K or 32K entries. We call them *Subset 512*, *Subset 2k*, *Subset 8k* and *Subset 32k*. The three predictors for *Superset Con* and *Superset Agg* are as follows: *y512* has the *y* Bloom filter of Table 6.4 and a 512-entry exclude cache; *y2k* has the same Bloom filter and a 2K-entry exclude cache; and *n2k* has the *n* Bloom filter of Table 6.4 and a 2K-entry exclude cache. We call the resulting predictors *Superset y512*, *Superset y2k*, and *Superset n2k*. Finally, the predictors for *Exact* are an 8-way predictor cache with 512, 2K, 8K and 32K entries. We call them *Exact 512*, *Exact 2k*, *Exact 8k* and *Exact 32k*.

Table 6.5 shows the parameters we assume for the prefetching predictors. Since we only evaluate the system with both node-side and memory-side predictors or none of these predictors, we name different configurations with parameters of both, using the following convention: *Pref X Y* is a prefetching predictor with *X* entries in the node-side predictor table and *Y* entries in the memory-side predictor table. We chose default sizes larger than

Prefetching Predictors	
Node-side Predictor	Memory-side Predictor
Size: **8K** or 16K entries Entry size: 16 or 15 bits Total size: 17KB or 32KB Access time: 1 or 2 cyc Assoc: 8-way Ports: 1	Size: **16K** or 32K entries Entry size: 64 bits Total size: 130KB or 260KB Access time: 2 cyc Assoc: 8-way Ports: 1

Table 6.5: Architectural parameters used for prefetching predictors. Cycle counts are in processor cycles.

the other predictors because these predictors have to record information about *all* nodes in the system, not only for a single node.

6.1.4 Handling Write Snoop Requests

In this study, we focus on *read* snoop requests. While our contribution also applies to write snoop requests, it is more relevant to reads due to their higher number and criticality.

Thus, write snoop requests do not use the *Flexible Snooping* predictors. The reason is that writes need to invalidate *all* the cached copies of a line. As a result, they would need a predictor of memory location *presence*, rather than one of memory location with supplier status.

In our simulations, we handle write snoop requests as follows. Recall from Table 4.4 that our *Flexible Snooping* algorithms fall into two classes: those that do not decouple read snoop messages into request and response (*Superset Con* and *Exact*, together with *Lazy*), and those that do (*Subset* and *Superset Agg*, together with *Eager*). Consequently, for the former, we do not decouple write snoop messages either. For the latter, we think it is fair to always decouple write snoops into request and response — it enables parallel invalidation of the nodes. Also, since we are not concerned with the implementation feasibility of *Oracle* and we use it to estimate the potential performance improvement of our techniques, we allow

write snoop messages to decouple for *Oracle* as well.

6.2 Evaluation

To assess the profitability of the techniques we propose, we first show results for the system configuration each of the techniques is most profitable on (Sections 6.2.1 and 6.2.2). We start by presenting performance and energy consumption improvements of *Flexible Snooping* techniques on a Multi-CMP configuration in Section 6.2.1. Section 6.2.2 shows performance and energy consumption improvements of *UncoRq* and *UncoRq* with prefetching on a Single-CMP configuration. In Section 6.2.3, we compare *Flexible Snooping* and *UncoRq* on both configurations. Section 6.2.4 compares the performance of the best technique on each configuration to the performance of a partitioned, hierarchical shared bus system. Finally, Section 6.2.5 analyses how sensitive these techniques are to variations in number of nodes, cache size and predictor size.

6.2.1 *Flexible Snooping* on Multi-CMP Configuration

In this section, we first compare *Flexible Snooping* algorithms to each other and to Lazy, Eager, and Oracle on a Multi-CMP configuration. For this comparison, we use the *Subset 2k*, *Superset y2k* and *Exact 2k* predictors. In all cases, the per-node predictors have 2K entries in their cache or exclude cache — although the Superset algorithms additionally use a Bloom filter. The predictor sizes are 4.8 Kbytes for Subset and Exact, and 7.3 Kbytes for the Superset algorithms.

In the following, we compare our *Flexible Snooping* algorithms to Lazy, Eager, and Oracle along four dimensions: number of snoop operations per read snoop request, number of read messages in the ring, total execution time, and energy consumption. We consider each dimension in turn.

All results in this section are presented in clusters of bars for each simulated workload.

In each cluster, bars represent Lazy (*Lazy*), Eager (*Eager*), Oracle (*Oracle*), Subset (*Subset*), Superset Conservative (*Superset Con*), Superset Aggressive (*Superset Agg*) and Exact (*Exact*), from left to right. We also add a cluster with the mean of all SPLASH-2 applications.

Number of Snoops per Read Request

The number of snoop operations per read snoop request for the different algorithms is shown in Figure 6.1. The SPLASH-2 cluster shows the arithmetic mean of all SPLASH-2 applications.

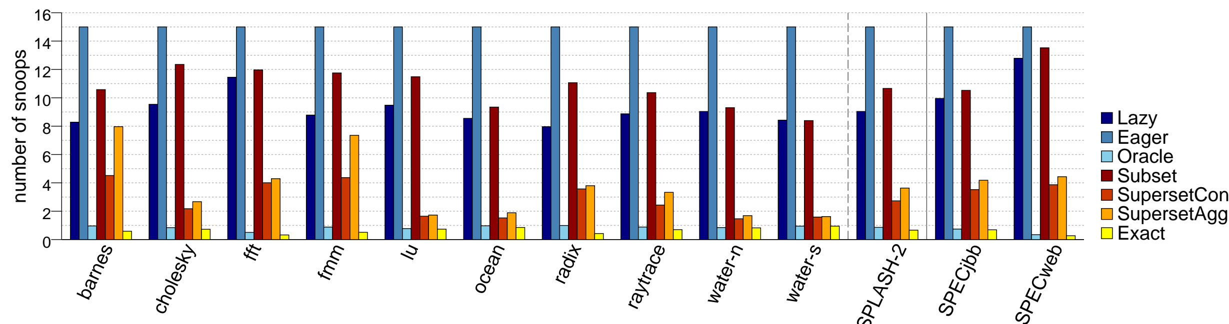


Figure 6.1: Average number of snoop operations per read transaction for different *Flexible Snooping* algorithms.

The figure shows that *Eager* snoops the most. As expected, it snoops all 15 CMPs in every request. As a result, it consumes more energy. If there is a supplier node, *Lazy* should snoop on average about half of the nodes, namely 7-8. In practice, since many requests do not find a supplier node and need to go to memory, *Lazy* snoops more. In particular, in SPECweb, threads do not share much data, and many requests go to memory. For this workload, *Lazy* incurs an average number of snoops close to 13.

The relative snoop frequency of the *Flexible Snooping* algorithms follows the graphical representation of Figure 4.2(b). For example, *Subset* snoops slightly more than *Lazy*. As indicated in Table 4.4, its additional snoops over *Lazy* depend on the number of false negatives. On the other hand, the Superset algorithms have many fewer snoops, typically 3-4.

As indicated in Figure 4.2(b), *Superset Con* snoops slightly less than *Superset Agg*.

Oracle has a very low value. Its number of snoops is less than one because when the line needs to be obtained from memory, *Oracle* does not snoop at all. This dilutes the one snoop *Oracle* performs when the miss can be serviced by another node. Finally, *Exact* is very close to *Oracle*. It has in fact fewer snoops than *Oracle* because, as indicated in Section 4.2.3, its supplier predictor induces some line downgrades. These downgrades result in fewer lines supplied by caches.

Number of Read Messages in the Ring

The total number of read snoop requests and responses in all segments of the ring for the different algorithms is shown in Figure 6.2. The results in this figure indirectly represent the relative bandwidth consumption of the different algorithms. In the figure, the bars for SPLASH-2 correspond to the geometric mean. Within an application, the bars are normalized to *Lazy*.

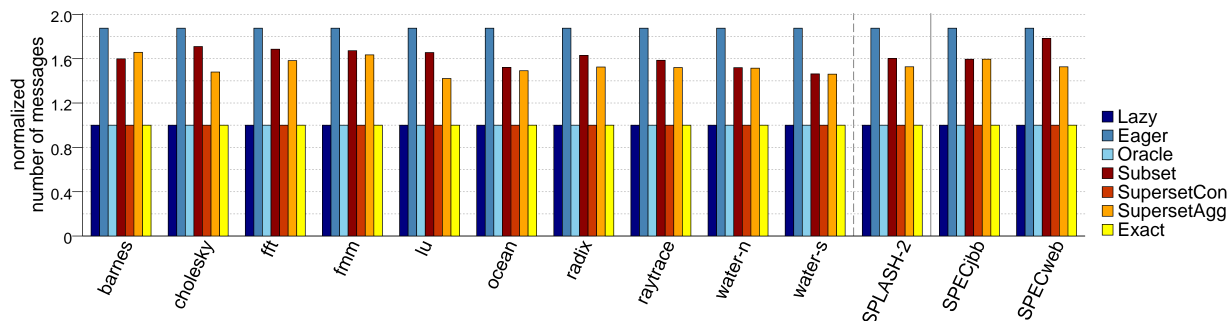


Figure 6.2: Total number of read snoop requests and responses in the ring for different *Flexible Snooping* algorithms. The bars are normalized to *Lazy*.

The figure shows that *Eager* has the most read messages in the ring. As indicated in Table 4.1, *Eager* generates nearly twice the number of messages of *Lazy*. The number is not exactly twice because request and response travel together in the first ring segment (Figure 4.1(b)) and the request does not traverse the last ring segment. Because of this

higher number of messages per miss, *Eager* consumes significant energy in the ring. *Lazy*, on the other hand, uses a single message on every segment and, therefore, is more frugal.

The relative number of messages in *Flexible Snooping* algorithms follows the discussion in Table 4.4. The number of messages in *Subset* and *Superset Agg* is between that of *Eager* and *Lazy*. The reason is that, while *Subset* often produces two messages per request, it merges them when it predicts that a line can be supplied by the local node. *Superset Agg* allows the request and the response to travel in the same message until it first predicts that the line may be supplied by the local node.

Finally, as indicated in Table 4.4, *Superset Con* and *Exact* have the same number of read messages as *Lazy* (and *Oracle*). This gives these schemes an energy advantage. The figure also shows that downgrades do not affect the number of read messages per miss in *Exact* — only the number of misses and the location from where they are supplied.

Total Execution Time

The total execution time of the applications for the different algorithms is shown in Figure 6.3. In the figure, the SPLASH-2 bars show the geometric mean of the applications. The bars are normalized to *Lazy*.

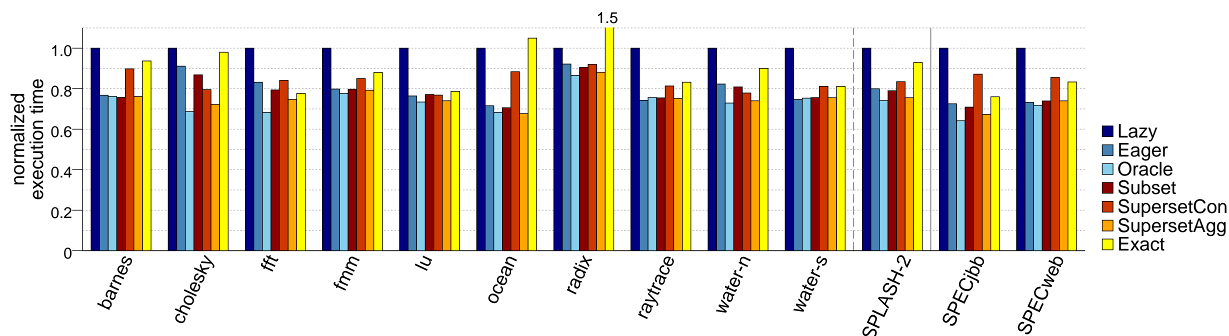


Figure 6.3: Execution time of the applications for different *Flexible Snooping* algorithms. The bars are normalized to *Lazy*.

To understand the results, consider Figure 4.2(b), which qualitatively shows the relative

snoop request latency in the different algorithms. In the table, *Lazy* has the longest latency, while *Superset Con* has somewhat longer latency than the other *Flexible Snooping* algorithms.

Figure 6.3 is consistent with these observations. The figure shows that, on average, *Lazy* is the slowest algorithm, and that most of the other algorithms track the performance of *Eager*. Of the *Flexible Snooping* schemes, *Exact* is slow when running SPLASH-2 and, to a lesser extent, SPECjbb and SPECweb. This is because it induces downgrades in these workloads, which result in more requests being satisfied by main memory than before.

Among the remaining three *Flexible Snooping* algorithms, *Superset Con* is the slightly slower one, as expected. When it runs the SPEC workloads, it suffers delays caused by false positives, which induce snoop operations in the critical path.

Superset Agg is the fastest algorithm. It is always very close to *Oracle*, which is a lower execution bound. On average, *Superset Agg* is faster than *Eager*. Compared to *Lazy*, it reduces the execution time of the SPLASH-2, SPECjbb, and SPECweb workloads by 24%, 33% and 26%, respectively.

Energy Consumption

Finally, we compute the coherence energy consumed by the different algorithms. We are interested in the energy consumed in servicing all read, write and invalidate transactions. Consequently, we add up the energy spent snooping nodes other than the requester, accessing and updating predictors, and transmitting request and response messages along the ring links. In addition, for *Exact*, we also add the energy spent downgrading lines in caches and, most importantly, the resulting additional cache line write backs to main memory and eventual re-reads from main memory. These accesses are counted because they are a direct result of *Exact*'s operation.

Figure 6.4 shows the resulting coherence energy consumption for the different algorithms. As usual, the SPLASH-2 bar is the geometric mean of the applications, and all bars are normalized to *Lazy*. The figure shows that *Eager* consumes about 85% more energy than

Lazy. This is because it needs more messages and more snoop operations than *Lazy*. Of the *Flexible Snooping* algorithms, *Subset* and *Superset Agg* are also less efficient than *Lazy*, as they induce more messages than *Lazy* and, in the case of *Subset*, more snoop operations as well. Still, *Superset Agg* consumes 15%, 10%, and 12% less energy than *Eager* for SPLASH-2, SPECjbb, and SPECweb, respectively.

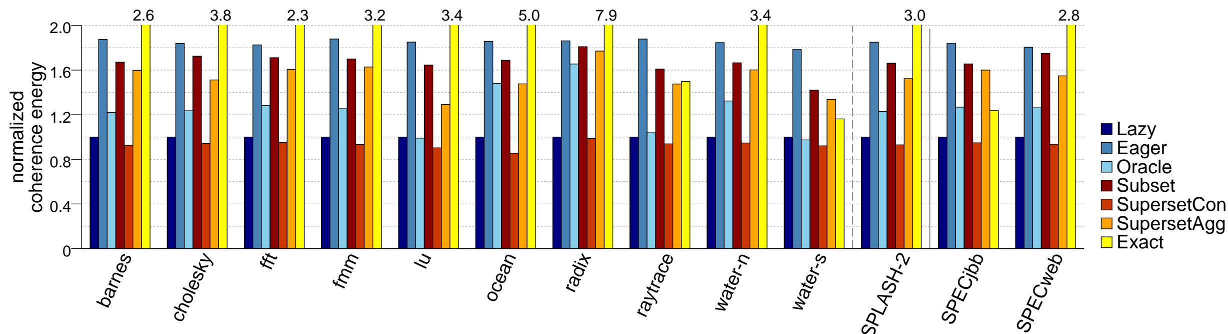


Figure 6.4: Energy consumed by on coherence transactions for different *Flexible Snooping* algorithms. The bars are normalized to *Lazy*.

Exact is not an attractive algorithm energy-wise. While it needs few snoop messages and snoop operations per read miss, it induces cache-line downgrading, which results in additional misses and traffic. As indicated in Section 4.2.3, some of these lines need to be written back to memory and later re-read from memory. As shown in Figure 6.4, this increases the energy consumption significantly, especially for applications with large amounts of sharing such as SPLASH-2.

Finally, *Superset Con* is the most energy-saving algorithm. Its energy consumption is the lowest, thanks to needing the same number of messages as *Lazy* and fewer snoop operations. Compared to *Lazy*, however, it adds the energy consumed in the predictors. In particular, the predictors used by the Superset algorithms consume substantial energy in both training and prediction. As a result, *Superset Con*'s energy is only slightly lower than *Lazy*'s. Compared to *Eager*, however, it consumes 48%, 47%, and 47% less energy for SPLASH-2, SPECjbb, and SPECweb, respectively.

Summary of Results

Based on this analysis, we argue that *Superset Agg* and *Superset Con* are the most cost-effective algorithms. *Superset Agg* is the choice algorithm for a high-performance system. It is the fastest algorithm — faster than *Eager* while consuming 10-15% less energy than *Eager*. For an energy-efficient environment, the choice algorithm is *Superset Con*. It is 15-20% faster than *Lazy* while consuming approximately the same energy.

Interestingly, both *Superset Con* and *Superset Agg* use the same supplier predictor. The only difference between the two is the action taken on a positive prediction (Table 4.3).

Therefore, we envision an adaptive system where the algorithm is chosen dynamically. Typically, *Superset Agg* would be the algorithm of choice. However, if the system needs to save energy, it would use *Superset Con*.

6.2.2 *UncoRq* on Single-CMP Configuration

This section shows the behavior of *UncoRq* in a Single-CMP system configuration. We show that *UncoRq* reduces request delivery time and improves cache-to-cache transfer latency, as expected. We then present the performance impact of *UncoRq* with and without the prefetching prediction optimization. The prefetch predictor is further characterized in Section 6.2.5.

In this section, we present our results contrasting four schemes: Lazy (*Lazy*), Eager (*Eager*), Unconstrained Snoop Request Delivery (*UncoRq*) and *UncoRq* with prefetching prediction (*UncoRq+Pref*). These are the bars that appear in each cluster, from left to right. Like in the previous section, an additional cluster shows the mean of all SPLASH-2 applications.

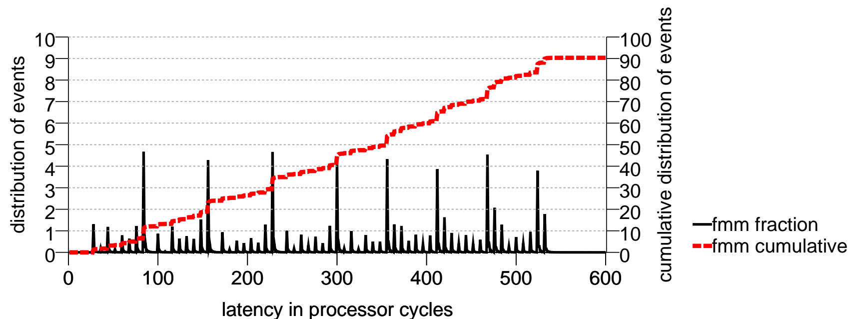
Data Consumption Latency

To verify that *UncoRq* reduces request delivery time and improves cache-to-cache transfer latency, we plot the data consumption latency distribution for *Eager* and *UncoRq*. We measure data consumption latency by recording, for each read miss, the elapsed time from the moment the miss is detected in the requester's L2 cache to the moment the L2 cache makes the data available to the requester's L1 cache. This corresponds to the request issue to data reception path from Figure 5.1.

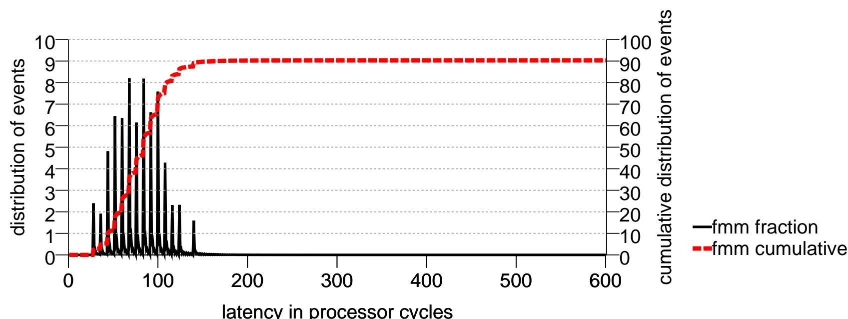
Figure 6.5 shows histograms generated for *fmm*, one of the SPLASH-2 applications. Other applications behave similarly. We limit the histograms to the part of the distribution corresponding to cache-to-cache transfers because those are the misses we are interested in. We also plot the cumulative distribution (dashed line). Figure 6.5(a) shows the data consumption latency distribution for *Eager*, while Figure 6.5(b) shows the data consumption latency distribution for *UncoRq*.

From Figure 6.5, we observe that *UncoRq* achieves its goal. First consider Figure 6.5(a): latencies are spread out, and the cumulative distribution grows gradually. Now consider Figure 6.5(b): latencies are more concentrated toward the left (shorter latencies), and the cumulative distribution grows faster. This shows *UncoRq* shortens data consumption latencies on cache-to-cache transfers.

Another interesting behavior Figure 6.5(a) shows is a number of spikes with different heights at relatively constant intervals. This behavior is a consequence of the node layout on the network and associated contention, and not exclusively to the application data distribution. For example, in *Eager*, requests are forwarded through the ring and data is sent back to the requester through the 2D torus. This combination leads to a configuration in which, for a fixed requester, particular distances to some nodes are more common than others. Assuming a uniform data distribution, the behavior of a few higher spikes and many shorter spikes is expected.



(a) SPLASH-2 *fmm* on *Eager*.



(b) SPLASH-2 *fmm* on *UncoRq*.

Figure 6.5: Data consumption latency on misses serviced with cache-to-cache transfers for *fmm* when executed on a system using (a) *Eager* and (b) *UncoRq*.

Table 6.6 provides further insight into application behavior. Column 2 and 3 list average data consumption latency for *all* read misses, on *Eager* and *UncoRq*, respectively; column 4 shows the average data consumption latency reduction of *UncoRq* compared to *Eager*; and column 5 shows the percentage of read misses serviced with cache-to-cache transfers.

For SPLASH-2 applications and SPECjbb, the average data consumption latency is lowered considerably by *UncoRq*. For SPECweb, however, the reduction is not as large, although still significant. The reason is that SPECweb has a lower fraction of misses that are serviced with cache-to-cache transfers, so it does not benefit as much from *UncoRq*.

benchmark	*Eager*	*UncoRq*	reduction	% c2c
barnes	317	111	65%	97%
cholesky	353	147	58%	90%
fft	559	524	6%	58%
fmm	344	144	58%	90%
lu	379	200	47%	82%
ocean	459	355	22%	99%
radix	320	100	69%	99%
raytrace	320	106	67%	96%
water-nsquared	387	190	51%	90%
water-spatial	312	93	70%	98%
SPLASH-2 avg	375	197	52%	90%
SPECjbb	413	247	40%	72%
SPECweb	598	522	13%	32%

Table 6.6: Average data consumption latency for *Eager* and *UncoRq*, percent data consumption latency reduction and percentage of cache-to-cache transfers.

Performance of ***UncoRq***

Figure 6.6 shows execution time normalized to *Lazy*. *UncoRq* is consistently faster than both *Lazy* and *Eager*. Compared to *Eager*, *UncoRq* reduces execution time by 20% for SPLASH-2 applications, 15% for SPECjbb and 6% for SPECweb. This reduction is correlated with the data consumption latency reduction from Table 6.6.

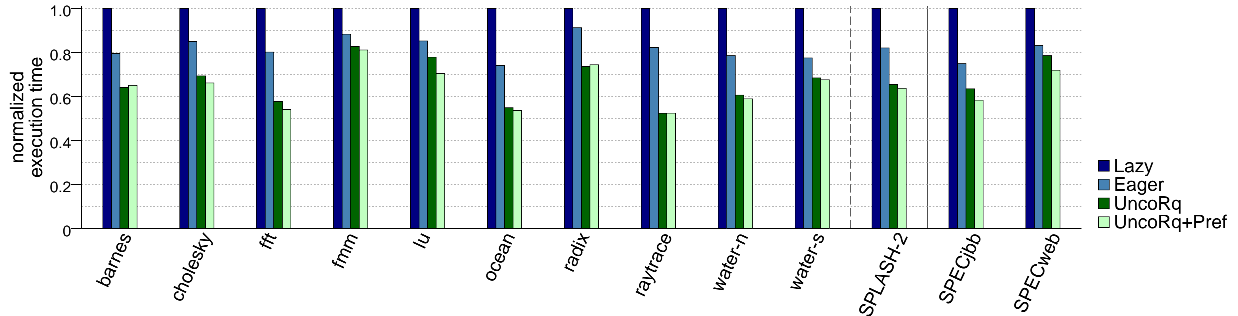


Figure 6.6: Execution time of *UncoRq* schemes. The bars are normalized to *Lazy*.

In addition, *UncoRq* with the prefetching prediction optimization reduces execution time even further. Compared to *Eager*, *UncoRq+Pref* provides a reduction in execution time of 22% for SPLASH-2 applications, 22% for SPECjbb and 14% for SPECweb. As expected, the

impact of the prefetching prediction optimization is more significant for applications with higher memory-to-cache transfer ratio, which is the case for commercial applications.

Figure 6.7 shows the average number of snoop operations necessary to service a read transaction. The SPLASH-2 cluster shows the arithmetic mean of all SPLASH-2 applications. We observe that, although *UncoRq* results in as many snoop operation as *Eager* when servicing a read miss, *UncoRq+Pref* generates less snoop operations. This is due to the node-side predictor, which many times predicts the supplier node correctly and sends the read request only to that node.

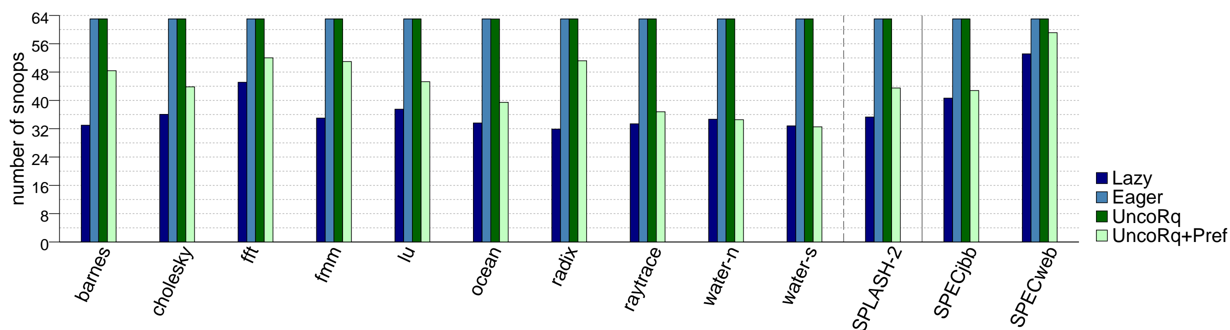


Figure 6.7: Average number of snoop operations per read transaction for *UncoRq*.

In Figure 6.8, we plot the number of snoop messages used to service a read miss in all network links. This indirectly shows what the bandwidth consumption is for all algorithms. The figure shows that, like with the number of snoop operations, there is a reduction on the number of messages used to service a read miss when the prefetching predictor is used. However, the reduction in the number of messages is smaller than the reduction in number of snoops because, although the request is sent directly to the predicted supplier, it has to pass through other intermediate nodes between requester and supplier. In addition, the response still has to traverse the entire ring, so savings apply only to requests (but to all snoop operations).

Figure 6.9 shows coherence energy consumption normalized to *Lazy*. *Eager*, *UncoRq* and *UncoRq+Pref* consume almost twice the energy *Lazy* does. This results directly from the

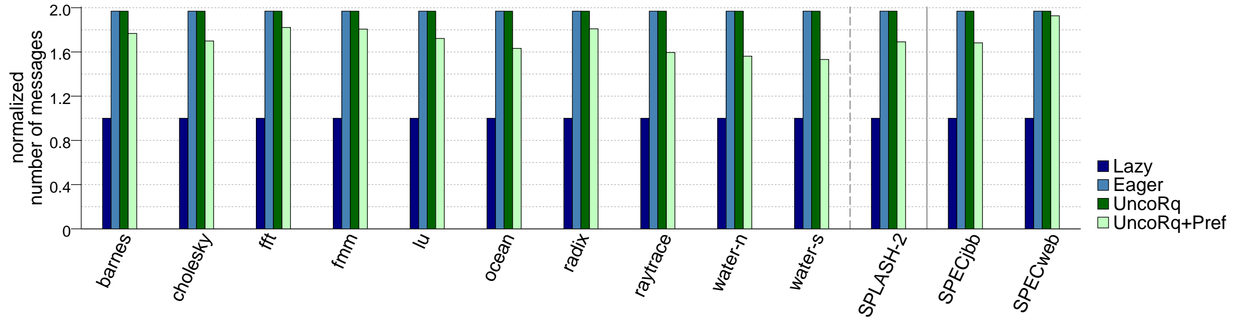


Figure 6.8: Total number of read snoop requests and responses in the ring for *UncoRq*. The bars are normalized to *Lazy*.

number of messages each of the protocols use to service a miss. For SPLASH-2 workloads, *UncoRq+Pref* uses less energy than *UncoRq* because of the lower number of snoops and messages *UncoRq+Pref* uses to service read requests. SPEC workloads do not show energy consumption reduction for *UncoRq+Pref* because they unnecessarily prefetch a larger portion of lines, as will be seen in Section 6.2.5.

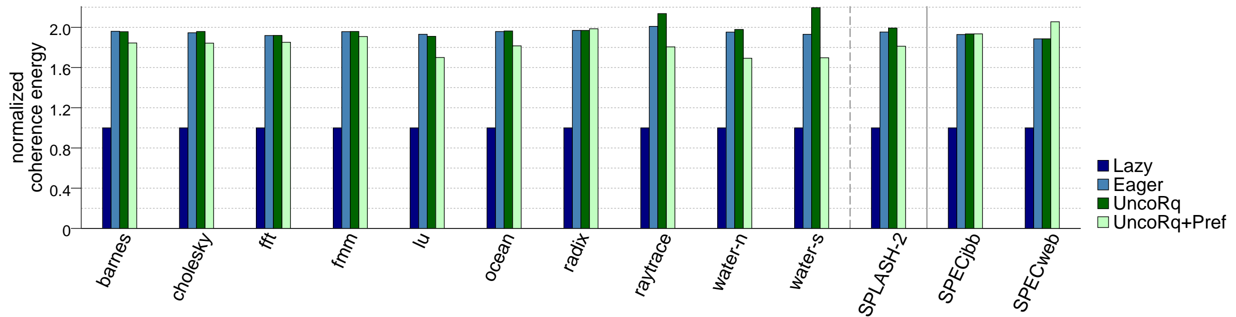


Figure 6.9: Energy consumed on coherence transactions for different *UncoRq* schemes. The bars are normalized to *Lazy*.

6.2.3 *Flexible Snooping* and *UncoRq* on Both Configurations

Multi-CMP Configuration

Figures 6.10 and 6.11 show performance and energy consumption, respectively, of various algorithms on a Multi-CMP configuration. The bars in each cluster correspond to *Lazy*,

Eager, *Superset Con*, *Superset Agg* and *UncoRq*, and we add a cluster with the geometric mean of all SPLASH-2 workloads.

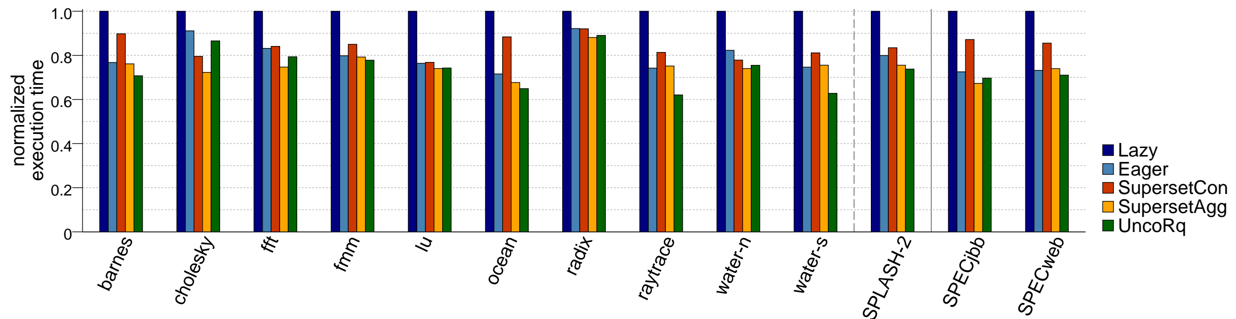


Figure 6.10: Execution time of best embedded-ring snooping schemes on a Multi-CMP configuration. The bars are normalized to *Lazy*.

Figure 6.10 shows that, for a Multi-CMP configuration, *Superset Agg* performs almost as well as *UncoRq*, on average for SPLASH-2 workloads. For SPECjbb, *Superset Agg* performs even better. This is due to the fact that the number of nodes in the ring is not very large, so the benefit of *UncoRq* is still not apparent. On the other hand, Figure 6.11 shows that *UncoRq* consumes as much coherence energy as *Eager*, while *Superset Agg* consumes less.

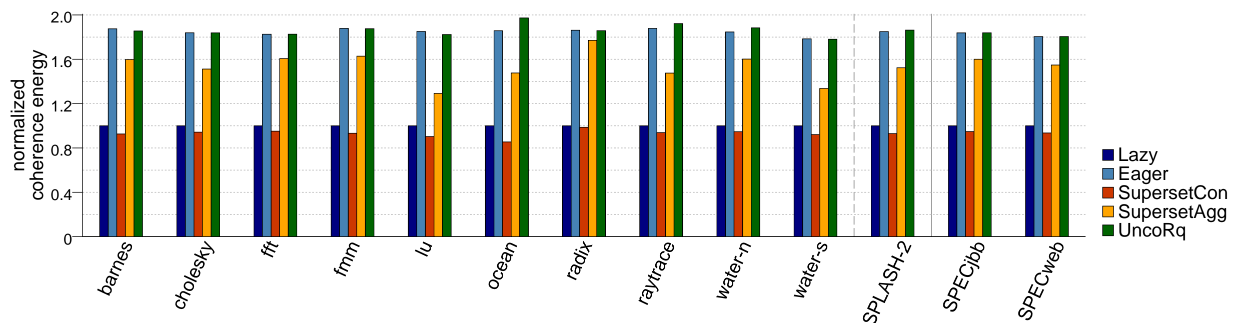


Figure 6.11: Energy consumed on coherence transactions for best embedded-ring snooping schemes on a Multi-CMP configuration. The bars are normalized to *Lazy*.

Since *Superset Agg* performs similarly to *UncoRq* but consumes less coherence energy, we choose *Superset Agg* as the most suitable algorithm for the Multi-CMP configuration we evaluate.

Single-CMP Configuration

Figures 6.12 and 6.13 show performance and energy consumption, respectively, of various algorithms on a Single-CMP configuration. Once again, the bars in each cluster correspond to *Lazy*, *Eager*, *Superset Con*, *Superset Agg* and *UncoRq*, and we add a cluster with the geometric mean of all SPLASH-2 workloads.

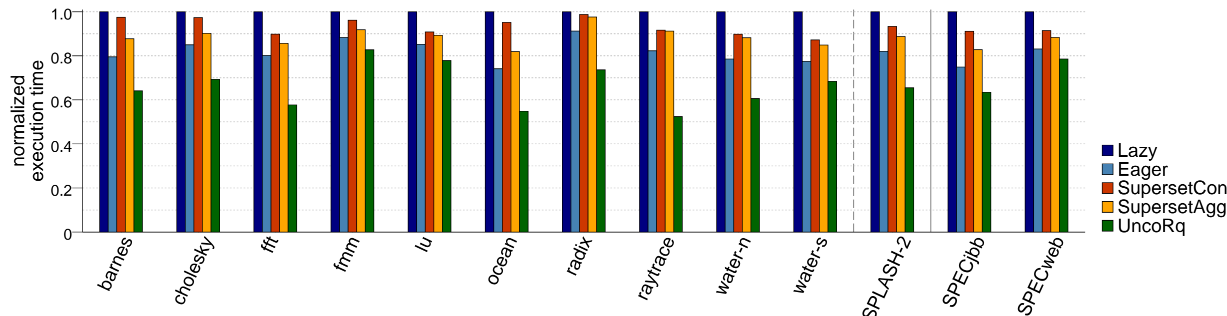


Figure 6.12: Execution time of best snooping schemes on a Single-CMP configuration. The bars are normalized to *Lazy*.

If we compare *Superset Con* and *Superset Agg* with *Eager* in Figure 6.12, we observe that both perform worse than *Eager*. This seems to contradict the results in the previous section, which show that both have performance similar to *Eager*, but it actually does not. The reason for this discrepancy is simple: in a Multi-CMP multiprocessor setting, the inter-node latency is much higher. This makes the relative latency the snoop predictors add to the request delivery, or the prediction overhead, less significant. In a Single-CMP multiprocessor setting, the inter-node latency is much lower and the prediction overhead more significant, rendering Superset algorithms less attractive in this setting.

Figure 6.13 shows that the relative energy consumption behavior of the algorithms in a Single-CMP configuration is very similar to the one in a Multi-CMP configuration.

In a Single-CMP configuration, there are more nodes connected to the ring. This increases the performance advantage of *UncoRq* when compared to algorithms that restrict requests to the ring. Even though *UncoRq* does not conserve energy, we choose *UncoRq* as our preferred

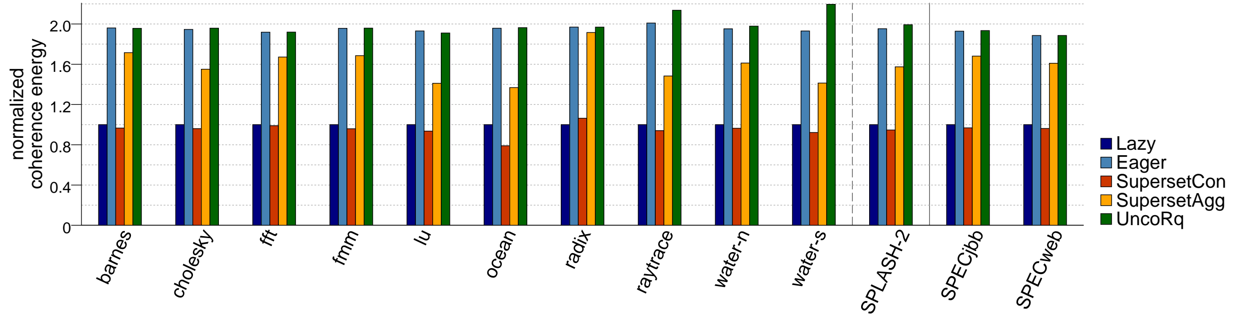


Figure 6.13: Energy consumed on coherence transactions for best snooping schemes on a Single-CMP configuration. The bars are normalized to *Lazy*.

algorithm for a Single-CMP configuration because its is significantly faster than the others.

6.2.4 Comparison to Hierarchical Buses

We now compare the best results obtained in Section 6.2.3 with a partitioned, hierarchical shared bus and show that embedded-ring cache coherence is a competitive approach.

We size the bus such that the number of wires it uses is equivalent to the total wire length (number of wires * length of each wire) in any of the embedded-ring approaches, which results in four address partitions. The network frequencies are the same, as well as the wire delays. However, the bus arbiters result in additional latencies not present in the embedded-ring approaches. The bus is laid out in a hierarchical pattern (a tree of degree four) with three levels of arbitration for 64 nodes. The above assumptions result in a design with four-bit flits that take two processor cycles each to propagate on the bus closest to the nodes (a message is 64 bits long). A coherence request has to reach the root arbiter before it is ordered and broadcast to all nodes. Responses are collected at the root arbiter and then sent back to the requester. All buses are pipelined with cycle time equivalent to the propagation time of one flit at the bus level closest to the nodes.

Figure 6.14 compares a partitioned, hierarchical shared bus (*HierBus* – first bar in each cluster) to an embedded-ring using *Superset Agg* (second bar in each cluster), in a Multi-CMP configuration. Bars are normalized to *HierBus*.

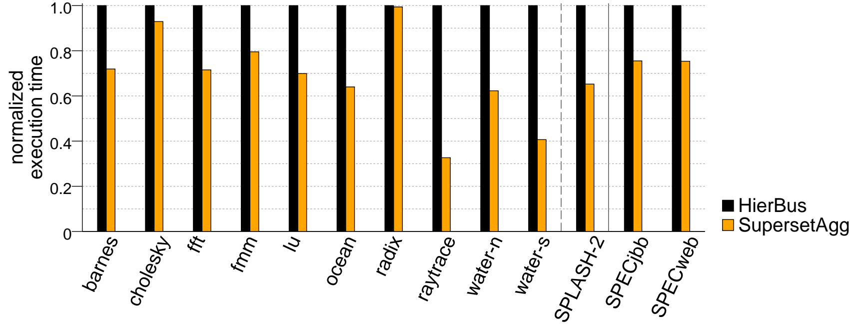


Figure 6.14: Execution time of *HierBus* and *Superset Agg* on a Multi-CMP configuration. The bars are normalized to *Lazy*.

We observe from Figure 6.14 that *Superset Agg* is always much faster than *HierBus*. For SPLASH-2, *Superset Agg* is 35% faster than *HierBus*. For SPECjbb and SPECweb workloads, *Superset Agg* is 25% faster. As explained above, the performance difference comes from the fact that an embedded-ring does not rely on central arbitration and therefore can achieve higher performance. In addition, the embedded-ring offers more request parallelism than the bus. While the bus can initiate only four requests in a particular cycle (because it only has four address partitions), a ring can initiate as many requests as nodes in the ring.

Figure 6.14 compares a partitioned, hierarchical shared bus (*HierBus* – first bar in each cluster) to an embedded-ring using *UncoRq+Pref* (second bar in each cluster), in a Single-CMP configuration. Bars are normalized to *HierBus*. In this case, *UncoRq+Pref* is 27%, 44% and 35% faster than *HierBus*, respectively, for SPLASH-2, SPECjbb and SPECweb.

These results show that, for both Multi-CMP and Single-CMP configurations, embedded-ring cache coherence is a competitive approach even when compared to a high-performance partitioned, hierarchical shared bus.

6.2.5 Sensitivity Analysis

Next, we present a sensitivity study and further characterize embedded-ring cache coherence protocols. We start by showing how the performance improvement of *Superset Agg* and

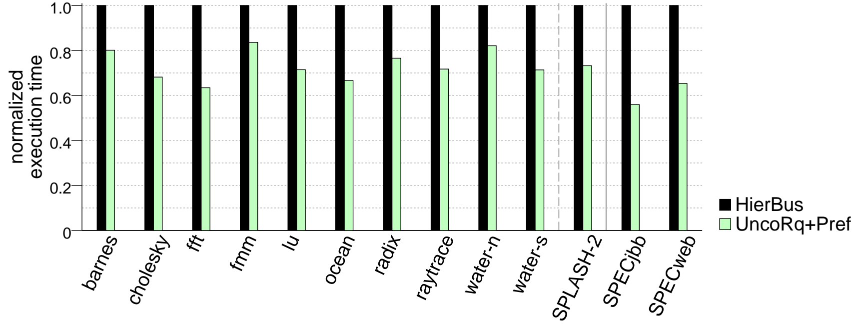


Figure 6.15: Execution time of *HierBus* and *UncoRq* on a Single-CMP configuration. The bars are normalized to *Lazy*.

UncoRq+Pref scales when the number of nodes on the ring increases. Then, we show how the performance improvement of *UncoRq+Pref* changes as the cache size varies. Finally, we provide a deeper characterization of algorithms and predictors we propose.

Sensitivity to Number of Nodes

To assess the sensitivity of embedded-ring cache coherence to the number of nodes in the system, we simulate all applications for both *Superset Agg* and *UncoRq+Pref* on their corresponding best configurations. Figures 6.16 and 6.17 show the execution time of *Superset Agg* and *UncoRq+Pref*, respectively, normalized to *Lazy*, on configurations with different number of processors (8, 16, 32, and 64). Note that an 8-processor configuration does not make much sense for the Multi-CMP configuration we use for *Superset Agg* because there would be only two nodes on the ring.

From Figure 6.16, we observe that the benefit of *Superset Agg* over *Lazy* increases with the ring size. Although the data can use any path from the supplier to the requester, the request still has to use the ring from the requester to the supplier. In a 2D torus topology, the data path on cache-to-cache transfers grows only with the square root of the number of nodes, but the request path grows with the number of nodes. Because of that, the fact that *Superset Agg*, unlike *Lazy*, does not require the request to wait for any snoop operation

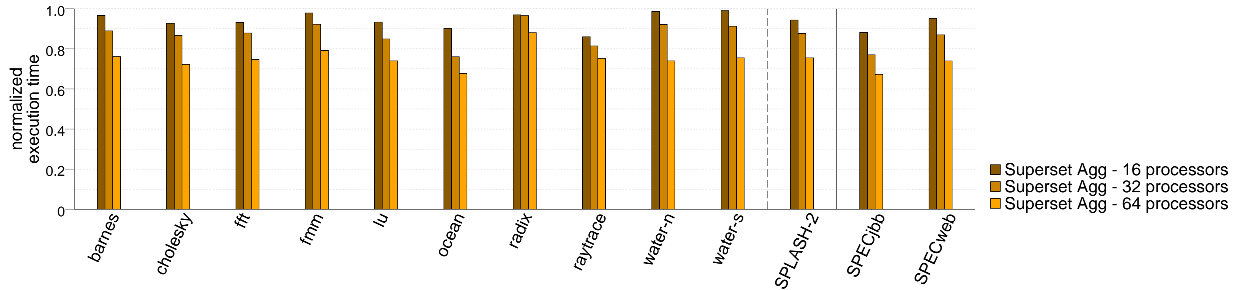


Figure 6.16: Execution time of *Superset Agg* on Multi-CMP configurations with varying number of nodes. The bars are normalized to *Lazy* on each configuration.

allows it to scale better with the number of nodes. Even though the cache-to-cache transfer ratio for SPEC workloads is lower, they still benefit from *Superset Agg* because its aggressive forwarding allows the global miss to be detected faster and thus accelerates memory-to-cache transfers as well.

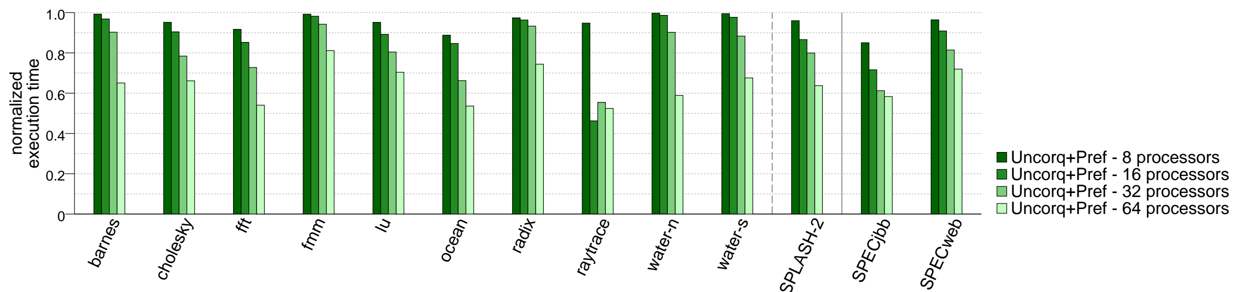


Figure 6.17: Execution time of *UncoRq* on Single-CMP configurations with varying number of nodes. The bars are normalized to *Lazy* on each configuration.

Figure 6.17 confirms that the benefit of *UncoRq+Pref* increases even more than *Superset Agg* with the number of nodes. This is because, with *UncoRq*, both the request delivery latency to the supplier and the data latency from supplier to requester grow only with the square root of the number of nodes. SPEC workloads, again, benefit from *UncoRq+Pref* because memory-to-cache transfers are accelerated, this time not only by aggressive forwarding, but also by prefetching.

Sensitivity to Cache Size

We also studied the impact of varying the cache size on the performance of *UncoRq+Pref* when compared to *Lazy*. Figure 6.18 shows the execution time of *UncoRq+Pref* normalized to *Lazy* using four different L2 cache sizes (128KB, 256KB, 512KB and 1MB), represented by the four bars in each cluster.

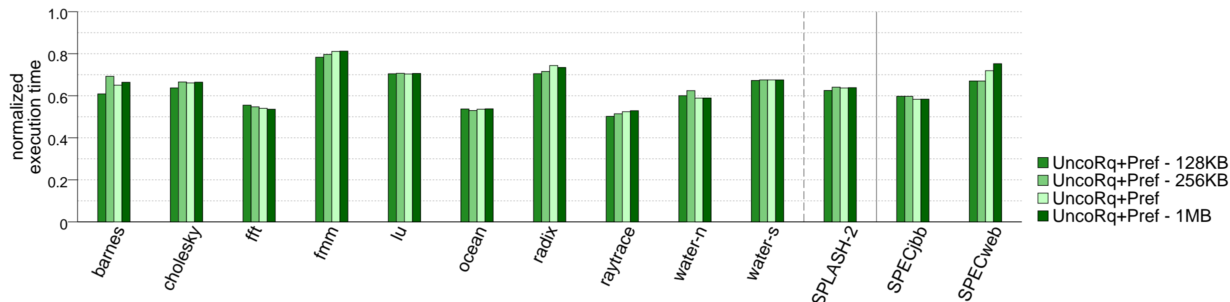


Figure 6.18: Execution time of *UncoRq+Pref 8k 16k* on a Single-CMP configuration with varying cache size. The bars are normalized to *Lazy* on each configuration.

Figure 6.18 exhibits an interesting behavior. No consistent trend can be found among benchmarks: for some, the performance benefit of *UncoRq+Pref* compared to *Lazy* decreases, for some it stays the same, and for some it increases. There are two opposing forces at play when we change the cache size. When the cache sizes increase, the miss rates decrease. This benefits *Lazy* more than it benefits *UncoRq+Pref* because *Lazy* takes longer to service misses. On the other hand, when caches increase, the number of cache-to-cache transfers may increase if the application has data sharing unexploited due to limitations in the cache size, which benefits *UncoRq+Pref* more then it benefits *Lazy*. Therefore, for applications where the first force dominates, like SPECweb, the benefit of *UncoRq+Pref* over *Lazy* decreases as we increase the cache size. For applications where the second force dominates, like fft, the benefit of *UncoRq+Pref* increases as we increase the cache size. For applications where both forces are equally strong, like lu, the benefit stays the same.

Flexible Snooping Supplier Predictors

In this section, we evaluate the impact of the supplier predictor size on the performance of *Flexible Snooping* algorithms. We evaluate the predictors described in Section 6.1, namely *Subset 512*, *Subset 2k*, *Subset 8k* and *Subset 32k* for Subset; *Superset y512*, *Superset y2k*, and *Superset n2k* for Superset; and *Exact 512*, *Exact 2k*, *Exact 8k* and *Exact 32k* for Exact.

To gain insight into how these predictors work, Figure 6.19 shows the fraction of true positive, true negative, false positive, and false negative predictions issued by read snoop requests using each of the above supplier predictors. The figure also includes a perfect predictor (*Oracle*) that is checked by the snoop request at every node, until the request finds the supplier node. The predictors for the two Superset algorithms behave very similarly and, therefore, we only show one of them. Each cluster of bars corresponds to the average of SPLASH-2 benchmarks, SPECjbb and SPECweb, from left to right.

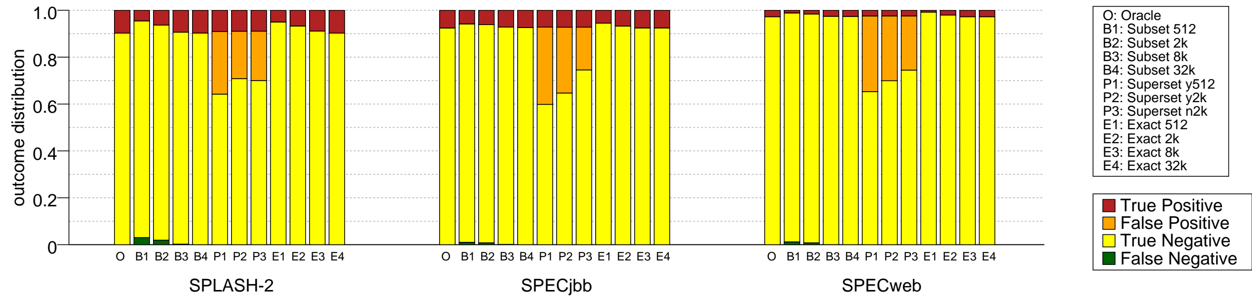


Figure 6.19: Supplier predictor accuracy for the different implementations of the Subset, Superset, and Exact algorithms.

Oracle represents a predictor with perfect information. Therefore, the predictor can tell exactly whether a node is able to supply the requested data or not. This is the behavior the other predictors should approximate. For SPLASH-2 and SPECjbb, the predictor makes about 9 negative predictions for every positive prediction. In SPECweb, however, the number of positive predictions is lower. This is because there is rarely a supplier node, and the request typically gets the line from memory.

The next four sets of bars show that the *Subset* predictors have few false negatives. As

we increase the size of the predictor from 512 entries to 32K entries, the number of false negatives decreases. For 32K entries, false negatives disappear, being entirely substituted by true positives.

On the other hand, the *Superset* predictors in the next three sets of bars have a significant fraction of false positives. For the best configuration for SPLASH-2 workloads (*Superset y2k*), false positives account for 20-28% of the predictions. The exclude cache helps reduce the false positives for all workloads, as can be observed when comparing bars P1 and P2: when the exclude cache size increases, many false positives are turned into true negatives.

Finally, the bars for the *Exact* predictors give an idea of the impact of downgrades. In Figure 6.19, the difference between these predictors and *Oracle* is due to limited space in the predictor tables, which results in downgrades. The more downgrades issued, the lower the fraction of true positives. We can see from the figure that, for an 32K entry predictor cache, the effect of the downgrades is inexistent because the *Exact* table is large enough to hold all lines cached by the node. However, as we decrease the predictor size, more downgrades result in a lower fraction of true positives.

Figure 6.20 shows how the performance of a system using the *Subset* predictor varies with the predictor size. Each cluster shows the execution times of an application as the predictor size increases. All bars in a cluster are normalized to the corresponding default predictor.

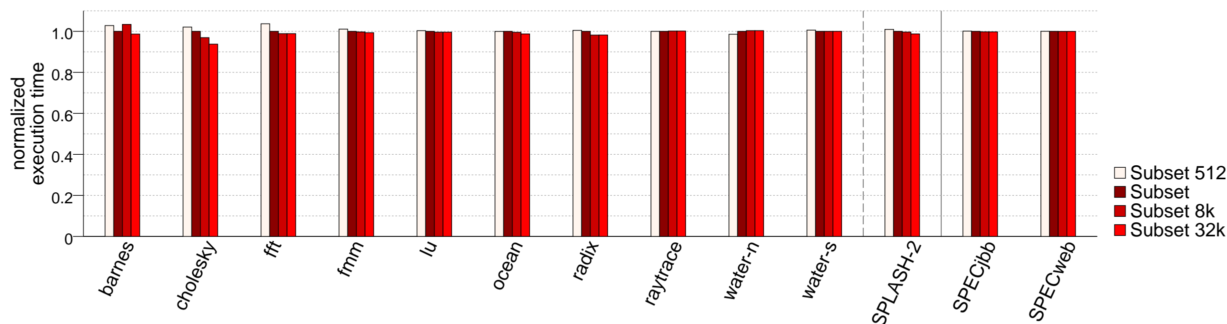


Figure 6.20: Execution time normalized to *Subset 2k* for a Multi-CMP system using the *Subset* predictor with varying predictor size.

For SPLASH-2, we can barely notice a reduction in execution time as the predictor size

increases. There is no change in execution time for SPEC workloads.

Although increasingly larger *Subset* predictors do not benefit performance, they allow the forwarding algorithm to detect a particular node is able to supply data to the requester increasingly more often. This allows the node predicted as the supplier to prevent the request to propagate further in the ring, avoiding unnecessary snoops and unnecessary messages, and saving energy.

This indeed saves energy, as shown in Figure 6.21. Each cluster in the figure shows the coherence energy consumed for increasing predictor sizes. For a few SPLASH-2 workloads, such as *lu*, the energy consumption is reduced by as much as 16% as the predictor grows from 512 entries to 32k entries. SPEC workloads are less sensitive to the predictor size because there are less cache-to-cache transfers for these applications and thus more read transactions for which a supplier node is never found.

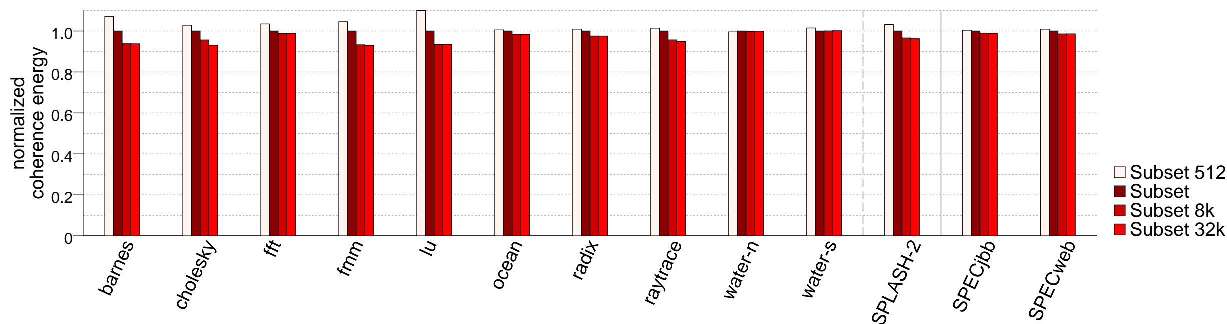


Figure 6.21: Coherence energy for a Multi-CMP system using the *Subset* predictor with varying predictor size. The bars are normalized to *Subset 2k*.

Figure 6.22 shows how the performance of a system using the *Superset Con* predictor varies with the predictor. Each cluster shows the execution times of an application for different predictor configurations.

We observe that the execution time does not vary consistently with different predictor configurations. For SPLASH-2 applications, the best performing predictor varies depending on the application and, although the average performance difference is practically insignificant, the *Superset y2k* predictor is slightly better. For SPEC workloads, *Superset n2k* is the

best predictor. Like execution time, energy consumption (shown in Figure 6.23) is mostly insensitive to the predictor configuration.

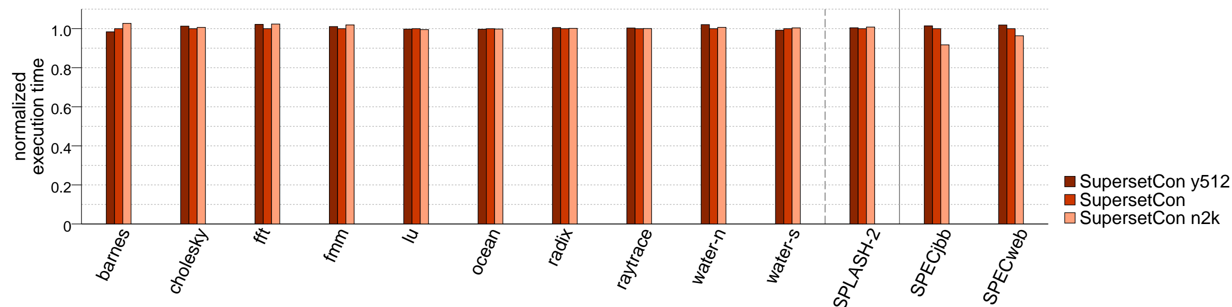


Figure 6.22: Execution time normalized to *Superset y2k* for a Multi-CMP system using the *Superset Con* algorithm with varying predictor configuration.

Figure 6.24 shows how the performance of a system using the *Exact* predictor varies with the predictor size. Each cluster shows the execution times of an application as the predictor size increases.

Unlike the other predictors, the execution times vary significantly with the size of the *Exact* predictor. This is due to the fact that small *Exact* predictors suffer many downgrades. As we increase the predictor size, the number of downgrades is reduced and, consequently, the amount of cache-to-cache transfers increases.

Energy consumption, presented in Figure 6.25, follows the same pattern because bigger predictors result in less accesses to memory, for two reasons: (1) there are more cache-to-

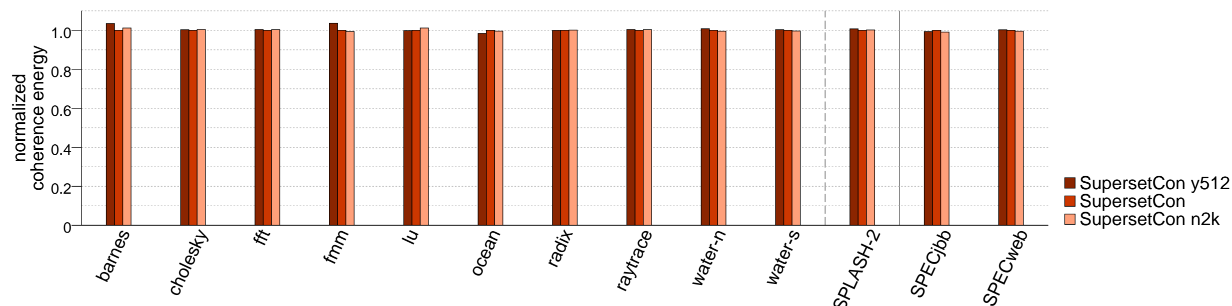


Figure 6.23: Coherence energy for a Multi-CMP system using the *Superset* predictor with varying predictor size. The bars are normalized to *Superset y2k*.

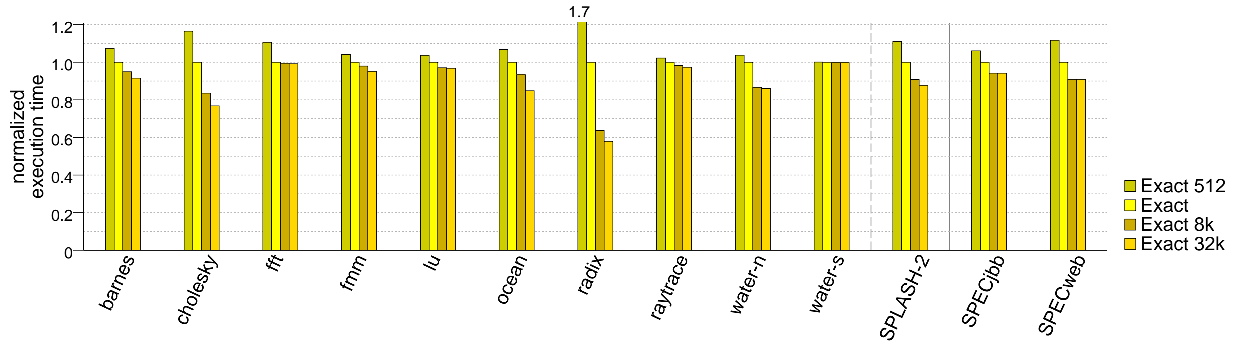


Figure 6.24: Execution time normalized to *Exact 2k* for a Multi-CMP system using the *Exact* predictor with varying predictor size.

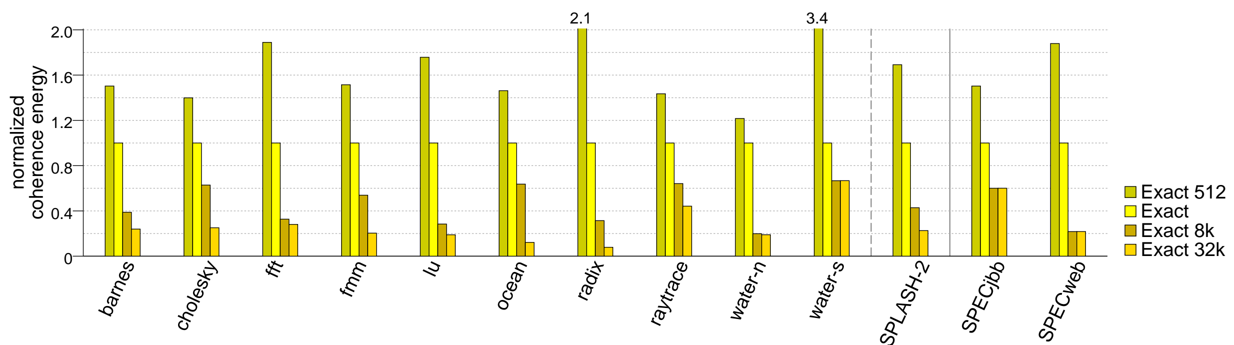


Figure 6.25: Coherence energy for a Multi-CMP system using the *Exact* predictor with varying predictor size. The bars are normalized to *Exact 2k*.

cache transfers (and consequently less memory accesses) and (2) caches have to write back dirty data less often due to the lower number of downgrades. Note that some applications observe no improvement when the predictor grows from 8k-entries to 32k-entries. This is because 8k-entries are usually sufficient to record all lines that can be supplied by a node. Predictors with 32k-entries are sufficient to record information about all lines in a CMP with 4 L2 caches of 512KB each and therefore there is no improvement beyond that point.

Prefetching Predictor

To provide further insight into the prefetching predictor behavior, we show its prediction outcome breakdown in Figure 6.26. From top to bottom, the categories correspond to: pre-

diction is "prefetch", but miss is serviced with a cache-to-cache transfer (*Pref,Cache* – wasted energy); prediction is "do not prefetch" and miss is indeed serviced with a cache-to-cache transfer (*NoPref,Cache*); prediction is "do not prefetch", but miss is serviced with a memory-to-cache transfer (*NoPref,Memory* – missed opportunity); and prediction is "prefetch" and miss is indeed serviced with a memory-to-cache transfer (*Pref,Memory*).

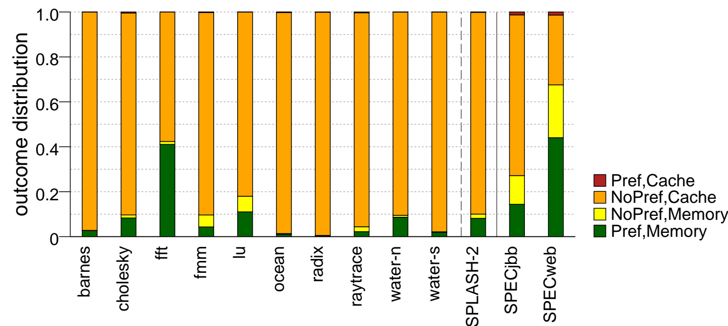


Figure 6.26: Outcome breakdown of prefetching prediction on read misses.

One important observation is that the *Pref,Cache* category is small. This means that the prefetching prediction optimization is not wasteful: the great majority of prefetches bring useful data from memory. From the two bottom categories, we observe that the prefetcher is able to prefetch a good percentage of all memory-to-cache transfers.

Table 6.7 characterizes average data consumption latency when the prefetching optimization is used. Column 2 shows average data consumption latency for *UncoRq+Pref* and column 3 shows the data consumption latency reduction of *UncoRq+Pref* compared to *UncoRq*. Again, the data consumption latency reduction is largely correlated with the reduction in execution time. One notable exception is *ocean*, which changes its sharing behavior and suffers from thread imbalance when the prefetching prediction optimization is used.

Figure 6.27 shows execution time normalized to *UncoRq+Pref*. The bars in each cluster correspond to *UncoRq+Pref*, *UncoRq+Pref 16k 16k*, *UncoRq+Pref 8k 32k* and *UncoRq+Pref 16k 32k*. The figure shows that execution time does not vary much as the size of the prefetcher is increased.

benchmark	*UncoRq+Pref*	reduction
barnes	103	7%
cholesky	125	14%
fft	418	20%
fmm	133	8%
lu	175	13%
ocean	235	34%
radix	98	2%
raytrace	102	4%
water-nsquared	147	22%
water-spatial	88	5%
SPLASH-2 avg	162	13%
SPECjbb	215	13%
SPECweb	427	18%

Table 6.7: Average data consumption latency for *UncoRq+Pref* and percent reduction of *UncoRq+Pref* compared to *UncoRq*.

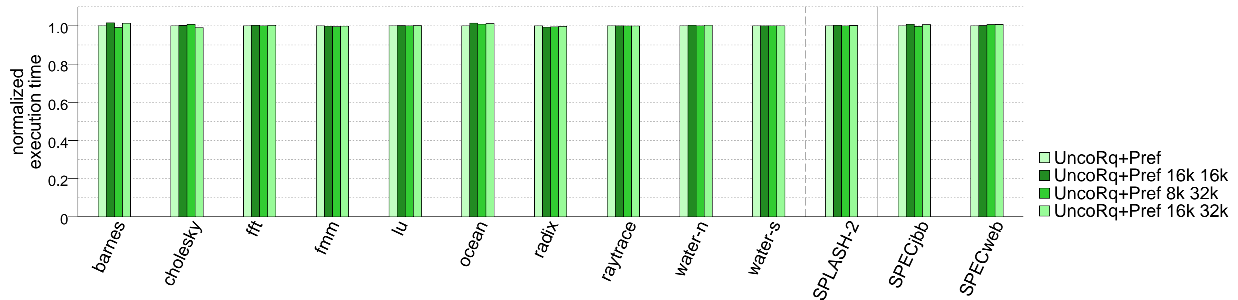


Figure 6.27: Execution time normalized to *UncoRq+Pref* for a Single-CMP system using the prefetching predictor with varying predictor size.

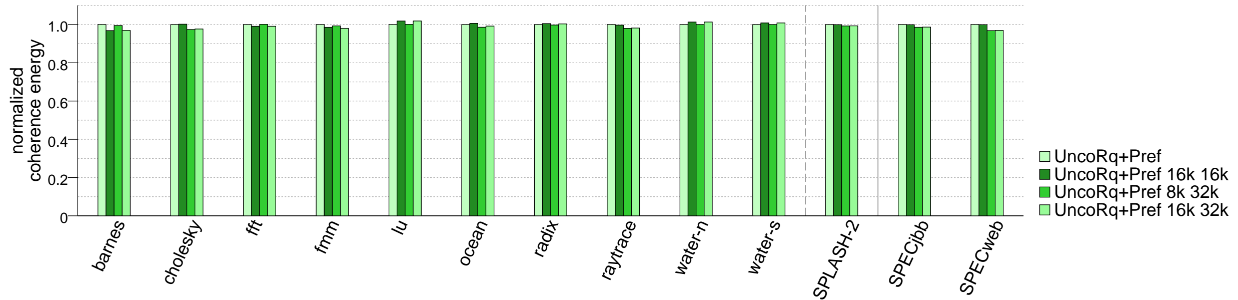


Figure 6.28: Coherence energy for a Single-CMP system using the prefetching predictor with varying predictor size. The bars are normalized to *UncoRq+Pref 8k 16k*.

The coherence energy for the same four prefetching predictors includes additional memory accesses caused by incorrect prefetches and is shown in Figure 6.28. We can observe that the coherence energy is reduced as the size of structures increase, but not by much, indicating that the predictor used as default is a good choice.

Chapter 7

Conclusions

In this thesis, we have proposed Embedded-Ring Cache Coherence, a very flexible approach that works on many types of point-to-point networks. It consists of embedding a logical unidirectional ring in an arbitrary network and using this ring to communicate a fraction of the control messages. Other messages can travel on any links of the network.

We show how Embedded-Ring Cache Coherence works and why it works, including details on the protocol and invariants it maintains. We provide evidence that the protocol maintains proper serialization of cache coherence transactions and that it guarantees forward progress, both at the system and at the node level.

By proposing optimizations to Embedded-Ring Cache Coherence, namely Flexible Snooping and Unconstrained Snoop Request Delivery, we show that this approach is a viable solution to cache coherence, especially with applications that show frequent cache-to-cache transfers.

Flexible Snooping is a family of adaptive forwarding and filtering snooping algorithms. We describe the primitive operations they rely on, analyze the design space of these algorithms and describe four general approaches, namely *Subset*, *Superset Con*, *Superset Agg*, and *Exact*. These approaches have different trade-offs in number of snoop operations and messages, snoop response time, energy consumption, and implementation difficulty.

We used SPLASH-2, SPECjbb, and SPECweb workloads to evaluate these approaches. Our analysis found some of the algorithms are more cost-effective than current ones. Specifically, our choice for a high-performance snooping algorithm (*Superset Agg* with a 7.3-Kbyte per-node predictor) was faster than the currently fastest algorithm (*Eager*), while consum-

ing 10-15% less energy; moreover, our choice for an energy-efficient algorithm (*Superset Con* with the same predictor) was 15-20% faster than *Lazy*, while consuming approximately the same energy.

With Unconstrained Snoop Request Delivery (*UncoRq*), we show that snoop requests can be delivered to multiple nodes *in parallel*, using *any path* in the network — as long as snoop responses (not data) use the logical ring. To exploit this observation, we propose a technique that adds little complexity to an existing embedded-ring multiprocessor.

We analyze the performance impact of *UncoRq*, and how it can be further optimized. Our results indicate that the proposed technique is very effective. On a 64-node CMP, this technique can improve performance over *Eager* by 20% for SPLASH-2 applications and 11% for commercial applications. With a simple additional optimization, we achieve an average performance improvement of 22% for SPLASH-2 applications and 18% for commercial applications.

Finally, we show that both *Superset Aggressive* and *UncoRq* outperform a high-performance partitioned, hierarchical shared bus by 25% to 44%, demonstrating that Embedded-Ring Cache Coherence is a competitive approach for the workloads considered.

Overall, this thesis has shown that Embedded-Ring Cache Coherence is a practical solution for cache coherence, and that it is particularly suitable to systems that target applications with high cache-to-cache transfer rates.

Appendix A

Exhaustive Enumeration of Message Interleavings

In this appendix, we list the operating rules of both the baseline embedded-ring protocol and the *UncoRq* protocol and show that, on a collision between any two nodes, these protocols maintain the single supplier invariant.

To achieve this goal, we analyze the message exchange events of two colliding transactions and all of their possible interleavings. There are four key message exchange events associated with each node involved in the collision: (a) when a node starts a transaction and sends the corresponding request and response; (b) when a node receives the combined response corresponding to its transaction back; (c) when a node receives the request corresponding to the transaction started by the other node involved in the collision; (d) when a node receives the combined response corresponding to the transaction started by the other node involved in the collision.

For each of the two nodes involved in the collision, we construct all possible interleavings of the above key events these nodes could observe individually. Some of them might be impossible, such as one in which a node receives a combined response back even before having started the corresponding transaction and sent a request and response.

Based on the rules mentioned above, we eliminate invalid interleavings. Next, we take each of the remaining interleavings for a node and combine with each of the remaining interleavings for the other node, creating combinations of all feasible interleavings.

We then analyze these interleaving combinations and again discard a few combinations using the rules mentioned above. This results in a list of all viable event combinations when the two nodes are involved in a collision together. Finally, for each of these combinations,

we show that the protocols are capable of detecting the collision and maintaining the single supplier invariant, be it just the effect of the event interleaving combination in question (supplier status goes to one node, which then transfers it to the other node) or by forcing one of the nodes to retry its transaction.

A.1 Regular Embedded-Ring Protocol

Table A.1 lists operating rules of the baseline embedded-ring protocol. These rules stem from the protocol operating principles (2, 3, 5), same-address FIFO order on ring links (1, 4), properties of the logical unidirectional ring (6).

Rule Number	Rule
1	A response never passes its corresponding request.
2	While a node is processing a transaction from another node, the node cannot start a transaction on the same address.
3	A node processes requests in the order it receives them.
4	Messages do not pass other messages on the same address in the ring links.
5	A node sends newly combined responses in the order their corresponding requests are processed by the node.
6	A combined response is received by all nodes on the ring before it is received back by its corresponding requester node.

Table A.1: Rules derived from the baseline embedded-ring protocol operating principles, same-address FIFO order on ring links, and properties of the logical unidirectional ring.

Table A.2 lists all the event interleavings for each of the nodes. The first column shows event interleavings for node A, and the second column shows event interleavings for node B, duals of A's interleavings. The third column shows the feasibility of each event interleaving (i.e., whether a particular event interleaving is possible) given the rules in Table A.1 and if not feasible, which rule is violated by that particular event interleaving. To represent different event interleavings, we adhere to the following notation: for node X, R_X represents the moment node X starts a transaction and sends its request and response, r_X represents

the moment node X receives its combined response back after the response has circulated the ring, R_Y represents the moment node X receives a request originated at node Y and r_Y represents the moment node X receives a combined response corresponding to the request R_Y. To determine the dual of an event interleaving, we substitute subscripts X for Y and Y for X. Time grows from left to right.

A number of interleavings violate rule 1 because they represent cases in which a combined response is received before its corresponding request. Since rule 1 dictates that a response never passes its corresponding request, these interleavings are impossible. A few other event interleavings violate rule 2 because they represent cases in which a node starts a transaction while it is still processing messages corresponding to another transaction on the same address (i.e., after the node received a request from another node on the same address but before it receives a combined response corresponding to the received request), which is prohibited by rule 2. Finally, a number of other interleavings violate causality: these are interleavings represent cases in which a combined response is received even before a node has started its corresponding transaction. Only four message interleavings are valid for each of the nodes, and we only consider those in our case-by-case analysis.

In our case-by-case analysis, we take each of the interleavings identified in Table A.2 as feasible for node A and combine each of them with each of the interleavings identified as feasible for node B. Table A.3 lists all these combinations. By using the rules in Table A.1 once again, we now determine which *combinations* are feasible, this time taking into account the first and second columns together.

In Table A.3, a few event interleaving combinations violate rule 6. This is because they represent situations in which each of the nodes receives their own responses back before they receive each other's responses. Since rule 6 requires that a response is received by all nodes before it can be received back by its corresponding requester, these combinations are impossible. A few other event interleaving combinations violate either rule 3, 4 or 5. This is because these interleavings represent situations in which responses pass other messages

A	B	Feasibility
R_A, r_A, R_B, r_B R_A, R_B, r_A, r_B R_A, R_B, r_B, r_A	R_B, r_B, R_A, r_A R_B, R_A, r_B, r_A R_B, R_A, r_A, r_B	feasible
R_A, r_A, r_B, R_B R_A, r_B, r_A, R_B R_A, r_B, R_B, r_A	R_B, r_B, r_A, R_A R_B, r_A, r_B, R_A R_B, r_A, R_A, r_B	impossible (violates rule 1)
R_B, r_B, R_A, r_A	R_A, r_A, R_B, r_B	feasible
R_B, R_A, r_B, r_A R_B, R_A, r_A, r_B	R_A, R_B, r_A, r_B R_A, R_B, r_B, r_A	impossible (violates rule 2)
R_B, r_B, r_A, R_A R_B, r_A, r_B, R_A R_B, r_A, R_A, r_B r_A, R_A, R_B, r_B r_A, R_B, R_A, r_B r_A, R_B, r_B, R_A r_A, R_A, r_B, R_B r_A, r_B, R_A, R_B r_A, r_B, R_B, R_A	R_A, r_A, r_B, R_B R_A, r_B, r_A, R_B R_A, r_B, R_B, r_A r_B, R_B, R_A, r_A r_B, R_A, R_B, r_A r_B, R_A, r_A, R_B r_B, R_B, r_A, R_A r_B, r_A, R_B, R_A r_B, r_A, R_A, R_B	impossible (violates causality)
r_B, R_B, R_A, r_A r_B, R_A, R_B, r_A r_B, R_A, r_A, R_B	r_A, R_A, R_B, r_B r_A, R_B, R_A, r_B r_A, R_B, r_B, R_A	impossible (violates rule 1)
r_B, R_B, r_A, R_A r_B, r_A, R_B, R_A r_B, r_A, R_A, R_B	r_A, R_A, r_B, R_B r_A, r_B, R_A, R_B r_A, r_B, R_B, R_A	impossible (violates causality)

Table A.2: Message interleavings at nodes involved in a collision and their feasibility for baseline embedded-ring protocol.

illegally on their way from A to B or from B to A, which necessarily violates one of these rules. Finally, one of the message interleaving combinations violates causality. In this case, node A sends its request R_A after receiving node B's request R_B. However, node B sends its request R_B after receiving node A's request R_A, a clearly impossible situation.

Table A.4 shows all feasible event interleaving combinations as determined in Table A.3. We analyze each of them for the case in which there is a supplier node. Without loss of generality, we assume nodes are positioned as shown in Figure A.1. The first column shows event interleavings at node A. The second column shows the sequence in which requests are

A	B	Feasibility
R_A, r_A, R_B, r_B	R_B, r_B, R_A, r_A	impossible (violates rule 6)
R_A, r_A, R_B, r_B	R_B, R_A, r_B, r_A	
R_A, r_A, R_B, r_B	R_B, R_A, r_A, r_B	impossible (violates rule 3, 4 or 5)
R_A, r_A, R_B, r_B	R_A, r_A, R_B, r_B	feasible
R_A, R_B, r_A, r_B	R_B, r_B, R_A, r_A	impossible (violates rule 6)
R_A, R_B, r_A, r_B	R_B, R_A, r_B, r_A	
R_A, R_B, r_A, r_B	R_B, R_A, r_A, r_B	impossible (violates rule 3, 4 or 5)
R_A, R_B, r_A, r_B	R_A, r_A, R_B, r_B	feasible
R_A, R_B, r_B, r_A	R_B, r_B, R_A, r_A	impossible (violates rule 3, 4 or 5)
R_A, R_B, r_B, r_A	R_B, R_A, r_B, r_A	
R_A, R_B, r_B, r_A	R_B, R_A, r_A, r_B	feasible
R_A, R_B, r_B, r_A	R_A, r_A, R_B, r_B	impossible (violates rule 3, 4, or 5)
R_B, r_B, R_A, r_A	R_B, r_B, R_A, r_A	feasible
R_B, r_B, R_A, r_A	R_B, R_A, r_B, r_A	
R_B, r_B, R_A, r_A	R_B, R_A, r_A, r_B	impossible (violates rule 3, 4, or 5)
R_B, r_B, R_A, r_A	R_A, r_A, R_B, r_B	impossible (violates causality)

Table A.3: Feasibility of interleaving combinations between two nodes involved in a collision for baseline embedded-ring protocol.

processed by the supplier node S, which we derive from the response reception interleaving at node B and rules 3, 4 and 5: according to these rules, the order in which node B (the node involved in the collision positioned after the supplier node S in ring order) receives responses should be the same order in which S receives and processes requests, and sends combined responses. In addition, these rules determine that the order in which S receives and processes requests should be the same as the order of requests in the interleaving of node A (the node involved in the collision positioned before the supplier node in ring order). The third column shows event interleavings at node B. The fourth column shows what type of serialization these event interleaving combinations represent, the fifth column shows how the embedded-ring protocol detects and can still recover from collisions, if necessary, and the sixth column shows which node is the winner for each combination (i.e., which node gets the supplier status and has its transaction ordered first in the total order of transactions).

We can identify two different cases: natural and forced serialization. With natural se-

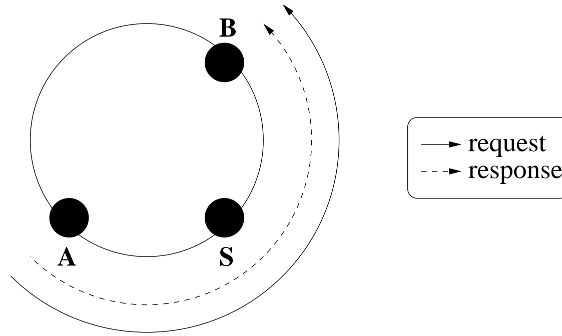


Figure A.1: Assumed relative position of nodes for baseline embedded-ring protocol exhaustive enumeration.

rialization, both nodes have the same event interleaving. This means that they "agree" on the order of their transactions, and no further action is needed unless the node whose transaction is ordered first is not ready to service the transaction ordered second when it receives the corresponding request, in which case the first node causes the second node to retry its transaction (first and fourth combinations).

With forced serialization, nodes have different interleavings. Two combinations (second and fifth combinations) are interleavings in which the request corresponding to the loser transaction passes the positive response corresponding to the winner transaction in one of the nodes positioned between the loser node and the winner node on the ring, causing the winner node to receive the other node's request before receiving its own combined response. In this case, when the winner node receives the other node's request, it detects a conflict; when it receives its own positive response, it learns it is the winner; when it receives the other node's combined response, it sends a signal to the other node forcing it to retry[1]. In the other case of forced serialization (third combination), the node succeeding the supplier node S in ring order (i.e. B) receives r_A+, from which it can tell A's transaction is the winner and conclude it needs to retry its own transaction.

[1]Another option is to reexecute the local snoop operation at the winner node for the request corresponding to the loser transaction when the winner node receives the loser node's combined response, and then incorporate the outcome of this operation into the combined response corresponding to the loser transaction, which at this point has not been sent by the winner node.

A	S	B	Type of serializ.	Outcome	Win-ner
R_A, r_A, R_B, r_B	R_A, R_B	R_A, r_A, R_B, r_B	natural	If A is not ready to service R_B when A receives R_B, A sends r_B-retry when A receives r_B	A
R_A, R_B, r_A, r_B		R_A, r_A, R_B, r_B	forced	A detects collision when it receives R_B, learns A is the winner when it receives r_A+ and sends r_B-retry when A receives r_B	
R_A, R_B, r_B, r_A		R_B, R_A, r_A, r_B	forced	B detects collision when it receives R_A, learns A is the winner when it receives r_A+ and retries when it receives r_B	
R_B, r_B, R_A, r_A	R_B, R_A	R_B, r_B, R_A, r_A	natural	If B is not ready to service R_A when B receives R_A, B sends r_A-retry when B receives r_A	B
R_B, r_B, R_A, r_A		R_B, R_A, r_B, r_A	forced	B detects collision when it receives R_A, learns B is the winner when it receives r_B+ and sends r_A-retry when B receives r_A	

Table A.4: Outcomes for feasible interleaving combinations between two nodes involved in a collision when there is a single supplier node for baseline embedded-ring protocol.

Table A.5 shows all feasible event interleaving combinations as determined in Table A.3. We now analyze each of them for the case in which there is no supplier node. The first column shows event interleavings at node A. The second column shows event interleavings at node B. The third column shows what type of serialization these event interleaving combinations represent, the fourth column shows how the embedded-ring protocol detects and can still recover from collisions, if necessary, and the fifth column shows which node is the winner for each combination.

Once again, we identify two cases: natural and forced serialization. The natural serialization cases (first and fourth combinations) are serviced identically to when there is a supplier

A	B	Type of Serializ.	Outcome	Winner
R_A, r_A, R_B, r_B	R_A, r_A, R_B, r_B	natural	If A is not ready to service R_B when A receives R_B, A sends r_B-retry when A receives r_B	A
R_A, R_B, r_A, r_B	R_A, r_A, R_B, r_B	forced	A detects collision and uses distributed arbitration algorithm to decide which node retries; if A, A retries immediately, if B, A sends r_B-retry when A receives r_B	A or B
R_A, R_B, r_B, r_A	R_B, R_A, r_A, r_B	forced	A and B use distributed arbitration algorithm to decide which node retries	A or B
R_B, r_B, R_A, r_A	R_B, r_B, R_A, r_A	natural	If B is not ready to service R_A when B receives R_A, B sends r_A-retry when B receives r_A	B
R_B, r_B, R_A, r_A	R_B, R_A, r_B, r_A	forced	B detects collision and uses distributed arbitration algorithm to decide which node retries; if B, B retries immediately, if A, B sends r_A-retry when B receives r_A	A or B

Table A.5: Outcomes for feasible interleaving combinations between two nodes involved in a collision when there is no supplier node for baseline embedded-ring protocol.

node: no further action is needed unless the node whose transaction is ordered first is not ready to service the transaction ordered second when it receives the request corresponding to the second transaction, in which case the first node causes the second node to retry its transaction via the response corresponding to the second transaction.

With forced serialization, there are again two subcases. The first subcase (second and fifth combinations) stems from interleavings that would be trivial cases of natural serialization if the request corresponding to one of the transactions had not passed the combined response corresponding to the other transaction. In the second combination, for example,

R_B passes r_A on its way from B to A. This causes node A to detect a conflict when it receives R_B and use the distributed arbitration algorithm to decide whether to retry its own transaction or to send a response that causes node B to retry its transaction[2]. The second subcase (third combination) results from a symmetrical interleaving and, because there is no supplier node to order the transactions, there is a tie, which is resolved by the distributed arbitration algorithm locally at each of the nodes. Since the distributed arbitration algorithm is guaranteed to pick the same one and only winner in each of the nodes, only one of the nodes will get the supplier status.

A.2 Embedded-Ring with Unconstrained Snoop Request Delivery

Table A.6 lists operating rules of the embedded-ring protocol with *UncoRq*. Compared to the baseline embedded-ring protocol, the only rules that change are rules 1 and 5. Since *UncoRq* does not use the ring to deliver requests, but uses it to deliver responses, requests and responses follow different paths to reach the same node. It may be the case that a combined response originated at a node reaches another node before its corresponding request. The *UncoRq* protocol does not take any action regarding a response before it receives its corresponding request, resulting in rule 1. The consequence is that an interleaving in which a response originated at another node appears before its corresponding request is equivalent to an interleaving in which the same response is received right after its corresponding request. For example, event interleaving R_A, r_B, r_A, R_B, observed by node A is equivalent to R_A, r_A, R_B, r_B. For this reason, we also call this rule the equivalence rule. Rule 5 is a product of how *UncoRq* handles the reordering of responses: positive responses should never be overtaken by other responses at any node.

[2]Since node A receives r_A with no retry signal before it receives r_B, this node could initiate its access to memory and send B a response that causes it to retry its transaction (r_B-retry) when it receives r_B, instead of using the distributed arbitration algorithm to decide which transaction should retry.

Rule Number	Rule
1	A node never takes any action regarding a response it receives before it receives the corresponding request. (equivalence rule)
2	While a node is processing a transaction from another node, the node cannot start a transaction on the same address.
3	A node processes requests in the order it receives them.
4	Messages do not pass other messages on the same address in the ring links.
5	Positive responses/outcomes are never overtaken by other responses at any node.
6	A combined response is received by all nodes on the ring before it is received back by its corresponding requester node.

Table A.6: Rules derived from the embedded-ring network properties and the *UncoRq* protocol properties.

Table A.7 lists all event interleavings for both nodes involved in a collision. The first column shows event interleavings for node A, and the second column shows event interleavings for node B, again duals of A's interleavings. To determine feasibility, we examine each event interleaving independently and this time determine whether they respect the rules in Table A.6. The interleaving feasibility, shown in the third column, applies to both first and second column when examined separately. If an event interleaving is not feasible, we also list which rules it violates. We use the same convention as in Section A.1 to represent event interleavings.

Many of the event interleavings listed in Table A.7 violate causality. This is because nodes receive their own combined response before even sending their own requests, a clearly impossible situation. Another set of event interleavings are not feasible because they represent a situation in which a node starts a transaction while it is still processing messages corresponding to another transaction on the same address, which violates rule 2. Many of the event interleavings, according to rule 1, are equivalent to others, so we choose only one event interleaving from each equivalence class to determine the combinations we examine in our case-by-case analysis[3]. We are left with four interleavings per node.

[3]Being equivalent means that the effective behavior of equivalent message interleavings would be the

A	B	Feasibility
R_A, r_A, R_B, r_B R_A, R_B, r_A, r_B R_A, R_B, r_B, r_A	R_B, r_B, R_A, r_A R_B, R_A, r_B, r_A R_B, R_A, r_A, r_B	feasible
R_A, r_A, r_B, R_B R_A, r_B, r_A, R_B	R_B, r_B, r_A, R_A R_B, r_A, r_B, R_A	feasible (equivalent to 1st interleaving)
R_A, r_B, R_B, r_A	R_B, r_A, R_A, r_B	feasible (equivalent to 3rd interleaving)
R_B, r_B, R_A, r_A	R_A, r_A, R_B, r_B	feasible
R_B, R_A, r_B, r_A R_B, R_A, r_A, r_B	R_A, R_B, r_A, r_B R_A, R_B, r_B, r_A	impossible (violates rule 2)
R_B, r_B, r_A, R_A R_B, r_A, r_B, R_A R_B, r_A, R_A, r_B r_A, R_A, R_B, r_B r_A, R_B, R_A, r_B r_A, R_B, r_B, R_A r_A, R_A, r_B, R_B r_A, r_B, R_A, R_B r_A, r_B, R_B, R_A	R_A, r_A, r_B, R_B R_A, r_B, r_A, R_B R_A, r_B, R_B, r_A r_B, R_B, R_A, r_A r_B, R_A, R_B, r_A r_B, R_A, r_A, R_B r_B, R_B, r_A, R_A r_B, r_A, R_B, R_A r_B, r_A, R_A, R_B	impossible (violates causality)
r_B, R_B, R_A, r_A	r_A, R_A, R_B, r_B	feasible (equivalent to 7th interleaving)
r_B, R_A, R_B, r_A	r_A, R_B, R_A, r_B	feasible (equivalent to 3rd interleaving)
r_B, R_A, r_A, R_B	r_A, R_B, r_B, R_A	feasible (equivalent to 1st interleaving)
r_B, R_B, r_A, R_A r_B, r_A, R_B, R_A r_B, r_A, R_A, R_B	r_A, R_A, r_B, R_B r_A, r_B, R_A, R_B r_A, r_B, R_B, R_A	impossible (violates causality)

Table A.7: Message interleavings at nodes involved in a collision and their feasibility for *UncoRq* protocol.

Again for our case-by-case analysis, we take each of the remaining interleavings of Table A.7 for each of the nodes and list all their combinations in Table A.8. We examine the pairs of interleavings as units to determine if, together, they are feasible combinations according to the rules in Table A.6. Compared with Table A.3, Table A.8 has many more feasible event interleaving combinations. This is because now responses are allowed to pass another message (unless the other message is a positive response), which makes all combinations originally violating rule 3, 4 or 5 of Table A.1 now feasible. As before, a few

same, so it is sufficient to choose only one message interleaving to represent an entire equivalence class.

combinations violate rule 6, and one violates causality.

A	B	Feasibility
R_A, r_A, R_B, r_B	R_B, r_B, R_A, r_A	impossible (violates rule 6)
R_A, r_A, R_B, r_B	R_B, R_A, r_B, r_A	
R_A, r_A, R_B, r_B	R_B, R_A, r_A, r_B	feasible
R_A, r_A, R_B, r_B	R_A, r_A, R_B, r_B	
R_A, R_B, r_A, r_B	R_B, r_B, R_A, r_A	impossible (violates rule 6)
R_A, R_B, r_A, r_B	R_B, R_A, r_B, r_A	
R_A, R_B, r_A, r_B	R_B, R_A, r_A, r_B	feasible
R_A, R_B, r_A, r_B	R_A, r_A, R_B, r_B	
R_A, R_B, r_B, r_A	R_B, r_B, R_A, r_A	
R_A, R_B, r_B, r_A	R_B, R_A, r_B, r_A	
R_A, R_B, r_B, r_A	R_B, R_A, r_A, r_B	
R_A, R_B, r_B, r_A	R_A, r_A, R_B, r_B	
R_B, r_B, R_A, r_A	R_B, r_B, R_A, r_A	
R_B, r_B, R_A, r_A	R_B, R_A, r_B, r_A	
R_B, r_B, R_A, r_A	R_B, R_A, r_A, r_B	
R_B, r_B, R_A, r_A	R_A, r_A, R_B, r_B	impossible (violates causality)

Table A.8: Feasibility of interleaving combinations between two nodes involved in a collision for *UncoRq* protocol.

Table A.9 shows all feasible event interleaving combinations from Table A.8. We analyze each of them for the case in which there is a supplier node. Once again, without loss of generality, we assume nodes are positioned as shown in Figure A.2. The first column shows event interleavings at node A. The second column shows the sequence in which requests are processed by the supplier node S, which we derive from the response interleaving at node B and rules 3, 4, and 5: the order of responses in B's interleaving is the order in which the supplier node S processes their corresponding requests. The third column shows event interleavings at node B, the fourth column shows what type of serialization each event interleaving combination represents, the fifth column shows how the *UncoRq* protocol detects and can still recover from collisions, if necessary, and the sixth column shows which node is the winner node (i.e., which node gets the supplier status and has its transaction ordered first in the total order of transactions).

While deriving the sequence in which requests are processed by the supplier node S, we find one event interleaving combination that is not feasible when a supplier is present in the position shown in Figure A.2 (eighth combination). From the event interleaving at B, we determine that the supplier processes messages in the sequence R_A, R_B. We also observe that node B starts its transaction and sends the corresponding request and response after it sends r_A+. However, the same combination also indicates that node A receives r_B before receiving r_A+, which violates rule 5, rendering this combination impossible under the circumstances we assume.

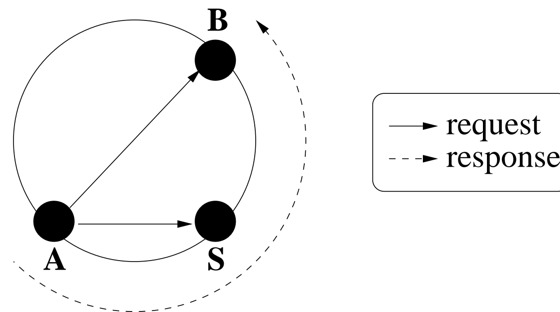


Figure A.2: Assumed relative position of nodes for *UncoRq* protocol exhaustive enumeration.

For the other combinations we can again identify cases of natural and forced serialization. With natural serialization (second and ninth combinations), both nodes have the same interleaving, so no further action is needed unless the node whose transaction is ordered first is not ready to service the transaction ordered second when it receives the corresponding request, in which case the first node forces the second node to retry its transaction.

All other event interleaving combinations lead to forced serialization. For all of them, the *UncoRq* protocol is capable of detecting a collision and recovering from it, i.e., it is always the case that at least one of the nodes identifies which node involved in the collision has to retry its transaction, and that always happens in time to take appropriate action and cause this node to indeed retry its transaction. For some event interleaving combinations (first, third, seventh and eleventh combinations), the loser node determines it should retry its transaction

A	S	B	Type of Serial.	Outcome	Winner
R_A, r_A, R_B, r_B	R_A, R_B	R_B, R_A, r_A, r_B	forced	B detects collision when it receives R_A, learns A is the winner when it receives r_A+ and retries when it receives r_B	A
R_A, r_A, R_B, r_B		R_A, r_A, R_B, r_B	natural	If A is not ready to service R_B when A receives R_B, A sends r_B-retry when A receives r_B	
R_A, R_B, r_A, r_B		R_B, R_A, r_A, r_B	forced	B detects collision when it receives R_A, learns A is the winner when it receives r_A+ and retries when it receives r_B	
R_A, R_B, r_A, r_B		R_A, r_A, R_B, r_B	forced	A detects collision when it receives R_B, learns A is the winner when it receives r_A+ and sends r_B-retry when A receives r_B	
R_A, R_B, r_B, r_A	R_B, R_A	R_B, r_B, R_A, r_A	forced	If B is not ready to service R_A when B receives R_A, B sends r_A-retry when B receives r_A	B
R_A, R_B, r_B, r_A		R_B, R_A, r_B, r_A	forced	B detects collision when it receives R_A, learns B is the winner when it receives r_B+ and sends r_A-retry when B receives r_A	
R_A, R_B, r_B, r_A	R_A, R_B	R_B, R_A, r_A, r_B	forced	B detects collision when it receives R_A, learns A is the winner when it receives r_A+ and retries when it receives r_B	A
R_A, R_B, r_B, r_A	–	R_A, r_A, R_B, r_B	–	impossible (violates rule 5)	–
R_B, r_B, R_A, r_A	R_B, R_A	R_B, r_B, R_A, r_A	natural	If B is not ready to service R_A when B receives R_A, B sends r_A-retry when B receives r_A	B
R_B, r_B, R_A, r_A		R_B, R_A, r_B, r_A	forced	B detects collision when it receives R_A, learns B is the winner when it receives r_B+ and sends r_A-retry when B receives r_A	
R_B, r_B, R_A, r_A	R_A, R_B	R_B, R_A, r_A, r_B	forced	B detects collision when it receives R_A, learns A is the winner when it receives r_A+ and retries when it receives r_B	A

Table A.9: Outcomes for feasible interleaving combinations between two nodes involved in a collision when there is a single supplier node for *UncoRq* protocol.

because it receives a positive response corresponding to the winner transaction on its way back to the winner node, indicating that the winner node will get supplier status. The loser node then detects it should retry, and it does it when it receives its own combined response. For other combinations (fourth, sixth and tenth combinations), thanks to rule 5, the winner node receives the positive response corresponding to its own transaction after it has started processing the loser transaction, but before it receives the combined response corresponding to the loser transaction (the *UncoRq* mechanism does not allow this response to pass the positive response), so the winner node can still send a retry signal to the loser node via the loser node's combined response, forcing it to retry when it receives its combined request.

Finally, Table A.10 shows all feasible combinations from Table A.8. We analyze each of the combinations, this time for the case in which there is no supplier node. The first column shows event interleavings at node A, the second column shows event interleavings at node B, the third column what type of serialization these event interleaving combinations represent, the fourth column shows how the *UncoRq* protocol detects and recovers from collisions, if necessary, and the fifth column shows which node is the winner for each combination.

For combinations resulting in natural serialization (second and ninth combinations), transactions are processed identically to when there is a supplier node: no further action is needed unless the node whose transaction is ordered first is not ready to service the transaction ordered second when it receives the request corresponding to the second transaction, in which case the first node causes the second node to retry its transaction via the response corresponding to the second transaction.

With forced serialization, at least one of the nodes is able to detect the collision. Even if only one of them detects it, it still does it in time to use the distributed arbitration algorithm, determine a winner transaction and cause the other node to retry, if necessary. The first combination is an example of that. Although A does not detect the collision, B does. When B receives R_A, which is before B receives and sends r_A, B uses the distributed arbitration algorithm to determine which transaction should proceed and which should retry

so, according to the result, it can still send a retry signal to A using r_A-retry if necessary.

A	B	Type of Serial.	Outcome	Winner
R_A, r_A, R_B, r_B	R_B, R_A, r_A, r_B	forced	B detects collision when it receives R_A and uses the distributed arbitration algorithm to determine winner; if A, B retries when it receives r_B; if B, B sends r_A-retry when B receives r_A	A or B
R_A, r_A, R_B, r_B	R_A, r_A, R_B, r_B	natural	If A is not ready to service R_B when A receives R_B, A sends r_B-retry when A receives r_B	A
R_A, R_B, r_A, r_B	R_B, R_A, r_A, r_B	forced	Same as 1st combination; also, same as 4th combination	A or B
R_A, R_B, r_A, r_B	R_A, r_A, R_B, r_B	forced	A detects collision when it receives R_B and uses the distributed arbitration algorithm to determine winner; if B, A retries when it receives r_A; if A, A sends r_B-retry when A receives r_B	
R_A, R_B, r_B, r_A	R_B, r_B, R_A, r_A	forced	Same as 4th combination	
R_A, R_B, r_B, r_A	R_B, R_A, r_B, r_A	forced	Same as 1st combination; also, same as 4th combination	
R_A, R_B, r_B, r_A	R_B, R_A, r_A, r_B	forced	Same as 1st combination; also, same as 4th combination	
R_A, R_B, r_B, r_A	R_A, r_A, R_B, r_B	forced	Same as 4th combination	
R_B, r_B, R_A, r_A	R_B, r_B, R_A, r_A	natural	If B is not ready to service R_A when B receives R_A, B sends r_A-retry when B receives r_A	B
R_B, r_B, R_A, r_A	R_B, R_A, r_B, r_A	forced	Same as 1st combination	A or B
R_B, r_B, R_A, r_A	R_B, R_A, r_A, r_B	forced	Same as 1st combination	

Table A.10: Outcomes for feasible interleaving combinations between two nodes involved in a collision when there is no supplier node for *UncoRq* protocol.

References

[1] M. E. Acacio, J. González, J. M. García, and J. Duato. Owner Prediction for Accelerating Cache-to-Cache Transfer Misses in a cc-NUMA Architecture. In *High Performance Computing, Networks and Storage Conference (SC)*, Nov 2002.

[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, Dec 1996.

[3] A. Agarwal, R. Bianchini, D. Chaiken, F. T. Chong, K. L. Johnson, D. Kranz, J. D. Kubiatowicz, L. Beng-Hong, K. Mackenzie, and D. Yeung. The MIT Alewife Machine. *Proceedings of the IEEE*, Mar 1999.

[4] J. Archibald and J.-L. Baer. An Economical Solution to the Cache Coherence Problem. In *International Symposium on Computer Architecture (ISCA)*, June 1984.

[5] L. Barroso and M. Dubois. The Performance of Cache-Coherent Ring-based Multiprocessors. In *International Symposium on Computer Architecture*, May 1993.

[6] B. Bloom. Space/time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 11(7):422–426, July 1970.

[7] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. In *International Symposium on Computer Architecture*, June 2005.

[8] J. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache System. *IEEE Transaction on Computers*, Dec 1978.

[9] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *International Symposium on Computer Architecture*, June 2007.

[10] P. Conway and B. Hughes. The AMD Opteron Northbridge Architecture, Present and Future. *IEEE Micro Magazine*, Mar/Apr 2007.

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

[12] N. Jouppi D. Tarjan, S. Thoziyoor. CACTI 4.0. Technical Report HPL-2006-86, HP Labs, June 2006.

[13] N. Eisley, L.-S. Peh, and L. Shang. In-network cache coherence. *Computer Architecture Letters*, March 2006.

[14] M. Ekman, F. Dahlgren, and P. Stenström. Evaluation of Snoop-Energy Reduction Techniques for Chip-Multiprocessors. In *Workshop on Duplicating, Deconstructing, and Debunking*, May 2002.

[15] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *International Symposium on Computer Architecture*, May 1999.

[16] J. R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *International Symposium on Computer Architecture*, June 1983.

[17] R. Grindley, T. Abdelrahman, S. Brown, S. Caranci, D. DeVries, B. Gamsa, A. Grbic, M. Gusat, R. Ho, O. Krieger, G. Lemieux, K. Loveless, N. Manjikian, P. McHardy, S. Srbljic, M. Stumm, Z. Vranesic, and Z. Zilic. The NUMAchine Multiprocessor. In *International Conference on Parallel Processing*, Aug 2000.

[18] HyperTransport Technology Consortium. *HyperTransport I/O Link Specification*, 2.00b edition, April 2005.

[19] International Technology Roadmap for Semiconductors 2005 Edition. http://www.itrs.net/Links/2005ITRS/Home2005.htm.

[20] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway. The AMD Opteron Processor for Multiprocessor Servers. *IEEE Micro Magazine*, Mar/Apr 2003.

[21] N. Kırman, M. Kırman, R.K. Dokania, J.F. Martínez, A.B. Apsel, M.A. Watkins, and D.H. Albonesi. Leveraging Optical Technology in Future Bus-based Chip Multiprocessors. In *International Symposium on Microarchitecture*, Dec 2006.

[22] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling. In *International Symposium on Computer Architecture*, June 2005.

[23] D. Lenosky, J. Laudon, K. Ghararchorloo, A. Gupta, and J. Hennessy. The Directory-based Cache Coherence Protocol for the DASH Multiprocessor. In *International Symposium on Computer Architecture (ISCA)*, May 1990.

[24] 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 *International Symposium on Computer Architecture*, June 2003.

[25] M. Martin, M. Hill, and D. Wood. Token Coherence: Decoupling Performance and Correctness. In *International Symposium on Computer Architecture*, June 2003.

[26] M. Marty, J. Bingham, M. Hill, A. Hu, M. Martin, and D. Wood. Improving Multiple-CMP Systems Using Token Coherence. In *International Symposium on High-Performance Computer Architecture*, Feb 2005.

[27] M. R. Marty and M. D. Hill. Coherence Ordering for Ring-based Chip Multiprocessors. In *International Symposium on Microarchitecture*, Dec 2006.

[28] Micron Technology, Inc. *System-Power Calculator*. http://www.micron.com/products/dram/syscalc.html.

[29] A. Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. In *International Symposium on Computer Architecture*, June 2005.

[30] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers. In *International Symposium on High-Perfomance Computer Architecture*, Jan 2001.

[31] C. Saldanha and M. Lipasti. Power Efficient Cache Coherence. In *Workshop on Memory Performance Issues*, June 2001.

[32] Silicon Graphics. Silicon Graphics Altix 3000 Scalable 64-bit Linux Platform. http://www.sgi.com/products/servers/altix/.

[33] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 System Microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–522, 2005.

[34] Standard Performace Evaluation Corporation (SPEC). http://www.spec.org.

[35] Sun Microsystems. Sun Enterprise 10000 Server Overview. http://www.sun.com/servers/highend/e10000/.

[36] K. Strauss, X. Shen, and J. Torrellas. Unconstrained Snoop Request Delivery. Submitted for publication.

[37] K. Strauss, X. Shen, and J. Torrellas. Flexible Snooping: Adaptive Forwarding and Filtering of Snoops in Embedded-Ring Multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, pages 327 – 338, June 2006.

[38] C. K. Tang. Cache Design in the Tightly Coupled Multiprocessor System. In *National Computer Conference*, June 1976.

[39] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. In *IBM Journal of Research and Development*, Jan 2002.

[40] Virtutech. Virtutech Simics. http://www.virtutech.com/products.

[41] Z. Vranesic, M. Stumm, D. Lewis, and R. White. Hector: A Hierarchically Structured Shared-Memory Multiprocessor. In *IEEE Computer Magazine*, Jan 1991.

[42] H. S. Wang, X. P. Zhu, L.-S. Peh, and S. Malik. Orion: A Power-Performance Simulator for Interconnection Networks. In *International Symposium on Microarchitecture*, Nov 2002.

[43] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for Store-wait-free Multiprocessors. In *International Symposium on Computer Architecture*, June 2007.

[44] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *International Symposium on Computer Architecture*, June 1995.

Author's Biography

Karin Strauss was born in São Paulo, Brazil. She wanted to be an architect, but was encouraged to go to Engineering School. During High School, she started playing with computers, and ended up getting her Bachelor's and Master's degrees in Electrical Engineering from the Polytechnic School of the University of São Paulo, in Brazil. During her Master's program, she had the opportunity of spending a year in the US, at the IBM T. J. Watson Research Center in Yorktown Heights, NY. During that time, she decided she wanted to pursue a PhD. When she started her PhD program, she finally became an architect, only a computer architect.

Karin has just finished her PhD program in the Department of Computer Science at the University of Illinois Urbana-Champaign (UIUC). Her doctoral research has focused on multiprocessor computer architectures with an emphasis on novel cache coherence protocol support. She has co-authored over 15 papers in computer architecture and system software. She has also participated in the Blue Gene and PERCS projects at IBM. She has received awards from UIUC for service, research and academic accomplishments, and from IBM for contribution to the Blue Gene project. She has held Intel and IBM PhD Fellowships during the program.