UNIVERSIDAD DE MURCIA
Departamento de Ingeniería y Tecnología de Computadores

Mejora del Rendimiento y la Escalabilidad de los Multiprocesadores de Memoria Compartida-Distribuida Basados en Directorios

TESIS PROPUESTA PARA
LA OBTENCIÓN DEL GRADO DE

DOCTOR EN INFORMÁTICA

Presentada por:
Manuel Eugenio Acacio Sánchez

Supervisada por:
José Duato Marín
José Manuel García Carrasco

Murcia, Febrero 2004

UNIVERSIDAD DE MURCIA

Departamento de Ingeniería y Tecnología de Computadores

Improving the Performance and Scalability of Directory-based Shared-Memory Multiprocessors

A DISSERTATION
SUBMITTED IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

DOCTOR EN INFORMÁTICA

By
Manuel Eugenio Acacio Sánchez

Advisors
José Duato Marín
José Manuel García Carrasco

Murcia, February 2004

UNIVERSIDAD
DE MURCIA

DEPARTAMENTO DE INGENIERÍA Y TECNOLOGÍA DE COMPUTADORES

D. *José Duato Marín*, Catedrático de Universidad del Área de Arquitectura y Tecnología de Computadores en el Departament d'Informàtica de Sistemes i Computadors de la Universitat Politècnica de València

y

D. *José Manuel García Carrasco*, Profesor Titular de Universidad del Área de Arquitectura y Tecnología de Computadores en el Departamento de Ingeniería y Tecnología de Computadores de la Universidad de Murcia,

AUTORIZAN:

La presentación de la Tesis Doctoral titulada *"Mejora del Rendimiento y la Escalabilidad de los Multiprocesadores de Memoria Compartida-Distribuida Basados en Directorios"*, realizada por D. *Manuel Eugenio Acacio Sánchez*, bajo su inmediata dirección y supervisión, en el Departamento de Ingeniería y Tecnología de Computadores, y que presenta para la obtención del grado de Doctor por la Universidad de Murcia.

En Murcia, a 30 de Enero de 2003

Fdo: Dr. José Duato Marín

Fdo: Dr. José Manuel García Carrasco

U N I V E R S I D A D
D E M U R C I A

DEPARTAMENTO DE INGENIERÍA Y TECNOLOGÍA DE COMPUTADORES

D. *José Manuel García Carrasco*, Profesor Titular de Universidad del Área de Arquitectura y Tecnología de Computadores y Director del Departamento de Ingeniería y Tecnología de Computadores de la Universidad de Murcia,

INFORMA:

Que la Tesis Doctoral titulada *"Mejora del Rendimiento y la Escalabilidad de los Multiprocesadores de Memoria Compartida-Distribuida Basados en Directorios"*, ha sido realizada por D. *Manuel Eugenio Acacio Sánchez*, bajo la inmediata dirección y supervisión de D. *José Duato Marín* y D. *José Manuel García Carrasco*, y que el Departamento ha dado su conformidad para que sea presentada ante la Comisión de Doctorado.

En Murcia, a 11 de Febrero de 2003

Fdo: Dr. José Manuel García Carrasco

Facultad de Informática ❖ Campus de Espinardo ❖ 30100 Espinardo (Murcia) ❖ Telf.: 968 36 46 65 ❖ Fax: 968 36 41 51

To the memory of my grandpa Eugenio,
To Laura, my little princess,
To mum and dad,
To Pepe, thanks for everything my friend!

Abstract

Cache-coherent, nonuniform memory access or cc-NUMA is an attractive architecture for building a spectrum of shared memory multiprocessors (which are seeing widespread use in commercial, technical and scientific applications). Based on a directory-based cache coherence protocol, cc-NUMA designs offer a scalable performance path beyond symmetric multiprocessors (SMPs) by maintaining a compatible programming interface and allowing a large number of processors to share a single global address space over physically distributed memory.

Unfortunately, there are some factors which limit the maximum number of processors that can be offered at a good price/performance ratio. Two of these factors are the increased cost in terms of hardware overhead and the long L2 cache miss latencies.

The most important component of the hardware overhead is the amount of memory required to store directory information, particularly the sharing code. Depending on how the sharing code is organized, memory overhead for large-scale configurations of a multiprocessor could be prohibitive. On the other hand, the long L2 miss latencies that characterize cc-NUMA architectures are a consequence of the inefficiencies that the distributed nature of directory-based coherence protocols and the underlying scalable network imply.

In this thesis, we address these two important issues. First of all, we present a new directory architecture, called *two-level directory*, which significantly reduces the amount of memory required by the directory while keeping the performance of a non-scalable full-map directory. Two-level directories combine a small first-level directory, which provides precise sharing information for the most recently referenced memory lines, with a compressed second level, which holds sharing information for all the lines. Additionally, we propose *multi-layer clustering* as an effective approach to reduce the width of the directory entries and derive three new compressed sharing codes with less memory requirements than the existing ones.

Then, we present a *taxonomy of the L2 cache misses* found in cc-NUMA multipro-

cessors in terms of the actions executed by the directory to serve them, and propose a *novel node architecture* which significantly accelerates each one of the categories of the taxonomy. The node architecture that we introduce exploits current integration scale by including some key components of the system inside the processor chip. In particular, it extends the two-level directory architecture to a three-level directory in which the small first level is implemented into the processor die. Additionally, the design includes a small on-chip shared data cache that is accessed in those cases in which the home node must provide a copy of the memory line. This way, the time a particular L2 cache miss spends at the home node is significantly reduced.

Finally, we develop *owner prediction* and *sharers prediction*, two prediction-based techniques aimed at completely removing the access to the directory from the critical path of cache-to-cache transfer misses and upgrade misses, respectively. For these misses, the directory is accessed to find the identity of the nodes that have to receive the corresponding coherence messages (transfer requests, for cache-to-cache transfer misses, and invalidation messages, for upgrade misses). These techniques try to send the coherence messages directly from the requesting node (instead of the directory) as a means to significantly accelerating these misses. In both cases, this requires the development of a highly accurate predictor as well as the extension of the original coherence protocol to solve the new race conditions that arise.

Acknowledgments

I have benefited from the help and support of many people while completing this thesis. I would like to take this opportunity to thank them.

First, I have had the fortune of enjoying the guidance of excellent professionals during these years. I would like to thank my advisors Prof. José Duato and José Manuel García, for years of support and wise guidance which made work easier. A very special thanks to José González, whose technical contributions and advice have been essential to the success of this thesis. I am truly grateful to all of them for all they have taught me.

Also, I would like to thank all my friends and colleagues at the University of Murcia, especially Antonio, Diego, Ginés, Gregorio, Javi, Juan Luis, Juan Peinador, Juan Piernas, Lorenzo, Óscar, Paco (*UCLM*), Pedro Enrique and Pilar. Having them as workmates makes work an enjoyable place to be. Special thanks to Gregorio, my officemate, for supporting me during these long five years. Thanks also to Heather Collins for revising the English language of this thesis.

I cannot forget all the people that I met last summer at IBM T.J. Watson. Thanks lots to all the members of the BlueGene/L group, especially to Alan Gara and Burkhard Steinmacher-Burow. Also, I would like to express my gratitude to all the good friends I met there, especially Sung-Jun Cho, Roongroj Nopsuwanchai, Rogeli Grima, David Mrva, and Zhengyong Zhu. They contributed to making that summer a fun time, full of invaluable experiences.

My parents have been a wonderful source of support, inspiration, and encouragement throughout my education, and I would not be who I am today without them. Thanks to my brothers Óscar and Sergio, and to my sister Laura for their infinite patience. All the thanks in the world to Laura, my beloved girlfriend and future wife. She has been my inspiration during the last seven years, and without her unfailing love, support, and understanding I would not have been able to succeed in most of the things I have done during these years.

Also, I would like to thank my uncle José Eugenio and my aunt Paloma, as well as

Laura’s family, especially Carmen, Juan, Sole, Joaquín, Eva, Marisol, Bartolo, Amparo, José Luis, and Toñi.

Finally, this thesis would not have been possible without the influence that my grandpa Eugenio had on me. From him I managed to discover the thrill of researching.

Contents

Abstract xi

Acknowledgments xiii

Contents xv

List of Figures xxi

List of Tables xxv

0. Resumen de la Tesis **1**

0.1. Introducción 1

0.1.1. Contribuciones de la Tesis 5

0.2. Entorno de Evaluación 6

0.3. Agrupamiento Multicapa y Directorios de Dos Niveles 8

0.3.1. Agrupamiento Multicapa 10

0.3.2. Directorios de Dos Niveles 13

0.4. Una Nueva Arquitectura para Reducir la Latencia de los Fallos de L2 . . . 15

0.4.1. Clasificación de los Fallos de L2 16

0.4.2. Una Nueva Arquitectura de Nodo 16

0.5. Predicción de Propietario para Acelerar los Fallos *3-hop* 21

0.5.1. Predictor de Dos Etapas 23

0.5.2. Extensiones al Protocolo de Coherencia 24

0.6. Predicción de Compartidores para Acelerar los Fallos *Upgrade* 26

0.6.1. Diseño del Predictor 27

0.6.2. Extensiones al Protocolo de Coherencia 29

0.7. Conclusiones y Vías Futuras 30

1. Introduction **33**
1.1. Motivation and Foundations 33
1.2. Thesis Contributions 42
1.2.1. Techniques to Reduce Directory Memory Overhead 42
1.2.2. Techniques to Reduce L2 Miss Latency 42
1.3. Thesis Organization 43

2. Evaluation Methodology **45**
2.1. Introduction 45
2.2. Rice Simulator for ILP Multiprocessors 45
2.2.1. Processor Microarchitecture 48
2.2.2. Memory Hierarchy 49
2.2.3. Interconnection Network 51
2.2.4. Summary of the Simulation Parameters 52
2.3. Benchmarks 52
2.3.1. BARNES-HUT 53
2.3.2. CHOLESKY 54
2.3.3. EM3D 54
2.3.4. FFT 55
2.3.5. MP3D 55
2.3.6. OCEAN 56
2.3.7. RADIX 56
2.3.8. UNSTRUCTURED 56
2.3.9. WATER-NSQ 57
2.3.10. WATER-SP 57
2.3.11. Summary of Applications Used in Our Experiments 58

3. Multi-layer Clustering and Two-level Directories **61**
3.1. Introduction 61
3.2. Related Work 65
3.3. Multi-layer Clustering Concept 67
3.3.1. Binary Tree (*BT*) 68
3.3.2. Binary Tree with Symmetric Nodes (*BT-SN*) 69
3.3.3. Binary Tree with Subtrees (*BT-SuT*) 69
3.3.4. Performance Evaluation 72

3.4. Two-level Directories: Reducing Memory Overhead and Keeping Performance . . . 77
3.4.1. Two-Level Directory Architecture . . . 78
3.4.2. Implementation Issues . . . 78
3.4.3. Performance Evaluation . . . 80
3.5. Conclusions . . . 83

4. A Novel Architecture to Reduce the Latency of L2 Misses 85
4.1. Introduction . . . 85
4.2. Related Work . . . 87
4.3. Taxonomy of L2 Cache Misses . . . 89
4.3.1. *$-to-$* Misses . . . 90
4.3.2. *Mem* Misses . . . 90
4.3.3. *Inv* Misses . . . 90
4.3.4. *Inv+Mem* Misses . . . 91
4.4. A Novel Architecture to Reduce L2 Miss Latency . . . 94
4.4.1. Node Architecture . . . 95
4.4.1.1. Three-Level Directory Architecture . . . 95
4.4.1.2. Shared Data Cache (SDC) . . . 97
4.4.2. Directory Controller Operation . . . 98
4.4.3. Implementation Issues . . . 101
4.5. Performance Evaluation . . . 102
4.5.1. Impact on L2 Miss Latencies . . . 103
4.5.1.1. Impact on *$-to-$* Miss Latencies . . . 104
4.5.1.2. Impact on *Mem* Miss Latencies . . . 105
4.5.1.3. Impact on *Inv* Miss Latencies . . . 106
4.5.1.4. Impact on *Inv+Mem* Miss Latencies . . . 108
4.5.2. Impact on Execution Time . . . 109
4.6. Conclusions . . . 111

5. Owner Prediction for Accelerating Cache-to-Cache Transfer Misses 115
5.1. Introduction . . . 115
5.2. Related Work . . . 118
5.3. Predictor Design for Cache-to-Cache Transfer Misses . . . 121
5.3.1. First-step Predictor . . . 121

5.3.2. Second-step Predictor 122
5.3.3. Implementation Issues 123
5.4. Coherence Protocol Supporting Prediction 124
5.4.1. Extended Coherence Protocol Operation 125
5.4.2. Dealing with Race Conditions: the *NPT* Table 127
5.4.3. Directory Requirements 128
5.5. Performance Evaluation 129
5.5.1. Predictor Accuracy 130
5.5.2. Performance Analysis 133
5.6. Conclusions 137

6. Sharers Prediction for Accelerating Upgrade Misses 139
6.1. Introduction 139
6.2. Related Work 142
6.3. Predictor Design for Upgrade Misses 143
6.4. Coherence Protocol Supporting Prediction 146
6.4.1. Extended Coherence Protocol Operation 146
6.4.2. Dealing with Race Conditions: the *ILT* Table 148
6.4.3. Directory Requirements 150
6.5. Performance Evaluation 151
6.5.1. Predictor Accuracy 152
6.5.2. Performance Analysis 154
6.6. Conclusions 159

7. Conclusions and Future Directions 161
7.1. Conclusions 161
7.2. Future Directions 165

Bibliography 166

A. RSIM Coherence Protocol and Compressed Directories 177
A.1. Original Coherence Protocol 177
A.1.1. Protocol States 178
A.1.2. Handling Load Requests 180
A.1.3. Handling Store Requests 181

A.1.4. Handling Write-Back Requests and Replacements 183
A.2. Extensions to Support Compressed Directories 183
A.3. Differences with the SGI Origin 2000 Protocol 185

List of Figures

0.1. Arquitectura de un multiprocesador de memoria compartida distribuida escalable con coherencia de cachés . . . 3
0.2. Sobrecarga de memoria para el código de compartición *full-map* . . . 4
0.3. Latencia media de los fallos de L2 . . . 4
0.4. Organización del directorio para la familia de protocolos de coherencia asumidos en esta tesis . . . 9
0.5. Ejemplo de aplicación del agrupamiento multicapa . . . 10
0.6. Tiempos de ejecución normalizados para los códigos de compartición *coarse vector*, *gray-tristate*, *BT-SuT* y Dir_1B . . . 12
0.7. Arquitectura de un directorio de dos niveles . . . 13
0.8. Tiempos de ejecución normalizados para la configuración *BT-SuT* y directorio de primer nivel(*FM*) . . . 14
0.9. Arquitectura propuesta para los nodos de un multiprocesador cc-NUMA . 17
0.10. Latencia media de los fallos *$-to-$* . . . 19
0.11. Latencia media de los fallos *Mem* . . . 20
0.12. Latencia media de los fallos *Inv* . . . 20
0.13. Latencia media de los fallos *Inv+Mem* . . . 21
0.14. Proceso de resolución de un fallo *3-hop* en un multiprocesador cc-NUMA convencional (izquierda) y en uno que usa predicción de propietario (derecha) . . . 22
0.15. Anatomía del predictor de dos etapas propuesto para los fallos *3-hop* . . . 24
0.16. Latencia media normalizada para los fallos *3-hop* . . . 25
0.17. Proceso de resolución de un fallo *upgrade* en un multiprocesador cc-NUMA convencional (izquierda) y en uno que usa predicción de compartidores (derecha) . . . 27

0.18. Anatomía del predictor propuesto para los fallos *upgrade* 28
0.19. Latencia media normalizada para los fallos *upgrade* 30

1.1. Multiprocessor cache coherence 35
1.2. Architecture of a simple bus-based shared memory multiprocessor 37
1.3. Architecture of a scalable cache-coherent shared memory multiprocessor . 38
1.4. Memory overhead for full-map 41
1.5. Average L2 miss latency 42

2.1. RSIM 16-node cc-NUMA multiprocessor system 46
2.2. How parallel applications are transformed for simulation 47

3.1. Directory organization for the family of coherence protocols assumed in this thesis 62
3.2. Multi-layer clustering approach example 68
3.3. Memory overhead as a function of the number of nodes 71
3.4. Normalized number of messages per coherence event for *BT-SN*, *BT* and Dir_0B sharing codes 73
3.5. Normalized number of messages per coherence event for *coarse vector*, *gray-tristate*, *BT-SuT*, and Dir_1B sharing codes 73
3.6. Normalized execution times for *BT-SN*, *BT* and Dir_0B sharing codes . . . 75
3.7. Normalized execution times for *coarse vector*, *gray-tristate*, *BT-SuT*, and Dir_1B sharing codes 76
3.8. Two-level directory organization 79
3.9. Normalized execution times for Dir_0B and first-level directory(*FM*) . . . 81
3.10. Normalized execution times for *BT* and first-level directory(*FM*) 82
3.11. Normalized execution times for *BT-SN* and first-level directory(*FM*) . . . 82
3.12. Normalized execution times for *BT-SuT* and first-level directory(*FM*) . . 83

4.1. Alpha 21364 EV7 diagram 86
4.2. Proposed chip organization 86
4.3. Average *$-to-$* miss latency 93
4.4. Average *Mem* miss latency 93
4.5. Average *Inv* miss latency 94
4.6. Average *Inv+Mem* miss latency 94
4.7. Proposed node architecture 97

4.8. Average *$-to-$* miss latency 104
4.9. Average *Mem* miss latency 105
4.10. Average *Inv* miss latency 107
4.11. Average *Inv+Mem* miss latency 108
4.12. Average L2 miss latency 109
4.13. Normalized execution times 110

5.1. Coherence protocol operations for a 3-hop miss in a conventional cc-NUMA (left) and in a cc-NUMA including prediction (right) 116
5.2. Normalized average *cache-to-cache transfer miss* latency 117
5.3. Anatomy of the two-step predictor 122
5.4. How prediction is included into the original coherence protocol 127
5.5. First-step predictor accuracy 131
5.6. Second-step predictor accuracy 131
5.7. Percentage of 3-hop misses predicted 132
5.8. Normalized average 3-hop miss latency 134
5.9. Normalized average load/store miss latency 135
5.10. Normalized execution times 136

6.1. Coherence protocol operations for an upgrade miss in a conventional cc-NUMA (left) and in a cc-NUMA including prediction (right) 140
6.2. Normalized average *upgrade miss* latency 141
6.3. Anatomy of the proposed predictor 144
6.4. Case in which the first scenario occurs (left) and how it is solved (right) . 148
6.5. Case in which the second scenario occurs (left) and how it is solved (right) 149
6.6. Percentage of upgrade misses predicted 152
6.7. Normalized average *upgrade miss* latency 155
6.8. Normalized average load/store miss latency 157
6.9. Normalized execution times 158

A.1. State transition diagram for a memory line at the directory in the coherence protocol implemented by RSIM 178
A.2. State transition diagram for a memory line at the caches in the coherence protocol implemented by RSIM 179

List of Tables

0.1. Resumen de los tamaños de problema, número máximo de procesadores que pueden usarse, número total de líneas de memoria compartida y patrones de compartición principales, para las aplicaciones empleadas en esta tesis 6
0.2. Valores de los parámetros usados en las simulaciones 7

2.1. Main processor parameters and their selected values 48
2.2. Main memory hierarchy parameters and their selected values 50
2.3. Network's main parameters and their selected values 51
2.4. Simulation parameters 52
2.5. Benchmarks and input sizes used in this work 53
2.6. Summary of the problem sizes, maximum number of processors that can be used, total number of memory lines that are referenced and dominant sharing patterns for the benchmarks used in this thesis 58

3.1. Behavior of the evaluated sharing codes for the proposed example 70
3.2. Number of bits required by each one of the sharing codes 71
3.3. Execution times, number of coherence events, number of events per cycle and messages per event for the applications evaluated, when full-map sharing code is used 72

4.1. Directory actions performed to satisfy load and store misses 91
4.2. Percentage of *$-to-$*, *Inv*, *Mem* and *Inv+Mem* misses found in the applications used in this thesis 92
4.3. Directory controller operation 100
4.4. How *$-to-$* misses are satisfied 104
4.5. How *Mem* misses are satisfied 105

4.6. How *Inv* misses are satisfied 106
4.7. How *Inv+Mem* misses are satisfied 108

5.1. Number of nodes included per prediction 133
5.2. Fraction of load and store misses that are 3-hop, and total percentage of 3-hop misses found in each application 135

6.1. How the coherence protocol works 150
6.2. Number of invalidations per upgrade miss (for *Base* case) and number of nodes included per prediction (for *UPT* and *LPT* cases) 156
6.3. Classification of the L2 misses according to the type of the instruction that caused them 157

Chapter 0

Resumen de la Tesis

0.1. Introducción

La mayoría de los computadores que utilizamos en el día a día (como el PC de casa o de la oficina) disponen de una única CPU. Estas máquinas, a menudo llamadas *máquinas monoprocesador*, ofrecen suficiente capacidad de cálculo para las tareas que realizamos cotidianamente la mayoría de los usuarios. Existen, sin embargo, algunas aplicaciones para las cuales la potencia ofrecida por un único microprocesador no basta. Por ejemplo, los grandes motores de búsqueda de Internet (como *Google*) deben ser capaces de satisfacer las peticiones que un gran número de usuarios están lanzando simultáneamente.

Hoy en día, la única alternativa viable para obtener más rendimiento del ofrecido por un único procesador consiste conectar varios microprocesadores de forma que puedan cooperar en la resolución de la misma tarea. Estas máquinas, conocidas como *multiprocesadores* (ó máquinas de tipo MIMD según la conocida clasificación de Flynn), pueden disponer de unos pocos microprocesadores (como los pequeños SMP con procesadores IA-32, cada vez más de moda) hasta miles de ellos (como el supercomputador BlueGene/L de IBM [86] que contará con hasta 65.536 procesadores para ofrecer un rendimiento pico de 360 teraFLOPS – billones de operaciones de punto flotante por segundo –).

En general, podemos distinguir dos familias de multiprocesadores: los multiprocesadores de *memoria compartida* y los multiprocesadores de *paso de mensajes*. Los multiprocesadores de memoria compartida ofrecen al programador un espacio de direcciones único que es compartido por todos los procesadores. En estas máquinas, la comunicación entre los distintos procesadores tiene lugar de forma implícita, mediante operaciones de carga y de almacenamiento a posiciones de memoria compartida. Por otro lado, en los

multiprocesadores de paso de mensajes cada procesador dispone de su propio espacio privado de direcciones. Dado que un determinado procesador no puede acceder al espacio de direcciones de otro, la comunicación entre procesadores en este tipo de máquinas se realiza mediante el envío de mensajes y ha de ser especificada de manera explícita por el programador.

Tradicionalmente, los multiprocesadores de memoria compartida han venido gozando de una mayor popularidad entre los programadores debido al sencillo modelo de programación que ofrecen (memoria compartida), que no es más que una extensión de la forma de programar en monoprocesadores.

Los multiprocesadores de memoria compartida se organizan a su vez de acuerdo a dos formas arquitectónicas distintas. Por un lado, en las *arquitecturas de memoria compartida centralizada* los procesadores comparten una memoria centralizada y se utiliza un bus para conectar éstos con la memoria. En estos multiprocesadores, que son conocidos como *multiprocesadores simétricos* o *multiprocesadores SMP*, cada procesador dispone de una gran memoria caché para reducir el número de accesos al bus y a la memoria compartida, y se utiliza un protocolo de coherencia de cachés basado en *fisgoneo* (*snooping*), a través del cual cada procesador observa las peticiones del resto y lleva a cabo las acciones apropiadas para el mantenimiento de la coherencia. Aún cuando pueden utilizarse varios buses (en lugar de uno), el número total de procesadores que pueden incorporarse en estos diseños es relativamente pequeño (hasta 64 en el caso del multiprocesador Sun Enterprise E10000 [18]).

Por otro lado, las *arquitecturas de memoria compartida distribuida* permiten incrementar de forma significativa el número total de procesadores del sistema (y, como consecuencia, la capacidad de cálculo) mediante la distribución de la memoria entre los distintos procesadores (para formar lo que se conoce como un *nodo*) y el uso de una red de interconexión punto a punto escalable, como una malla, para la conexión de los distintos nodos. Cada línea de la memoria compartida en estos diseños tiene asignado un nodo *home* en donde reside. El uso de memorias caché en este tipo de arquitecturas juega también un papel esencial a la hora reducir de forma significativa la latencia de acceso a la memoria. Sin embargo, a diferencia de los multiprocesadores SMP, la coherencia de las cachés en estas máquinas (que suelen ser conocidas como *multiprocesadores cc-NUMA*) se logra a través de un protocolo de coherencia basado en el uso de directorios. Cada línea de memoria tiene asociada una entrada de directorio que, entre otras cosas, informa sobre las cachés que almacenan una copia de la línea y que, por lo tanto, deberán recibir las transacciones

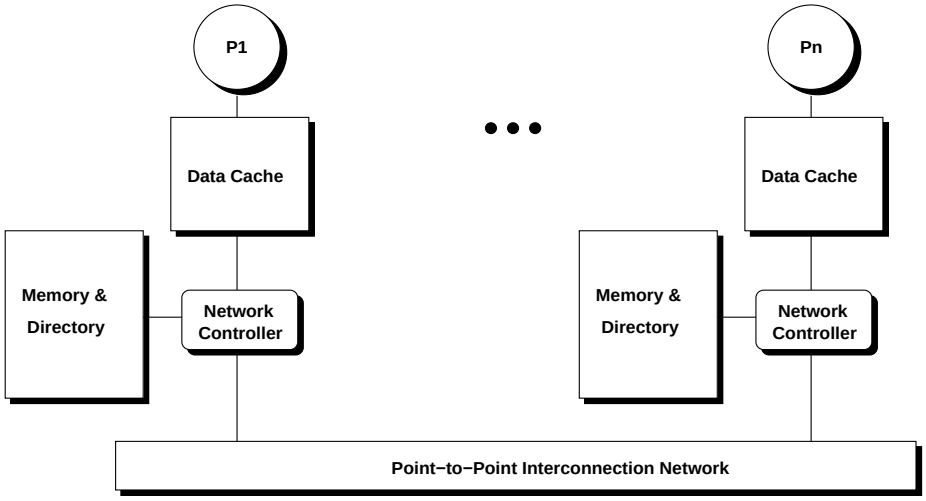


Figura 0.1: Arquitectura de un multiprocesador de memoria compartida distribuida escalable con coherencia de cachés

de coherencia correspondientes cuando sea preciso. Al igual que en el caso de la memoria, la información de directorio se distribuye entre los distintos nodos del sistema.

El multiprocesador Origin 2000 de Silicon Graphics [54] representa un ejemplo de un multiprocesador cc-NUMA actual que puede escalar desde unos pocos procesadores hasta 1024 (512 nodos$\times$2 procesadores/nodo). La Figura 0.1 muestra la arquitectura de una máquina cc-NUMA típica.

A pesar de que basándose en el uso de directorios se puede llevar a cabo el diseño de multiprocesadores de memoria compartida escalables, se han identificado varios factores que limitan la escalabilidad de estos diseños. Dos de los más importantes son: (1) el costo, en términos de circuitería adicional, que supone el uso de directorios, y, (2) el gran número de ciclos de reloj que se requieren para resolver cada fallo de L2.

El primero de los dos factores anteriores está fundamentalmente causado por la cantidad de memoria necesaria para almacenar la información de directorio, y más concretamente, la parte del directorio en la que se codifican los nodos que en un instante determinado mantienen una copia de cada línea (el *código de compartición*). La Figura 0.2 muestra la sobrecarga de memoria (calculada como tamaño del código de compartición dividido entre el tamaño de la línea de memoria) introducida por el código de compartición más sencillo, *full-map*, cuando se utilizan líneas de memoria de 128 bytes. Como puede observarse, conforme el número de nodos en el sistema aumenta, la sobrecarga de memoria debida a los directorios también lo hace, llegando a ser del 100 % para una configuración

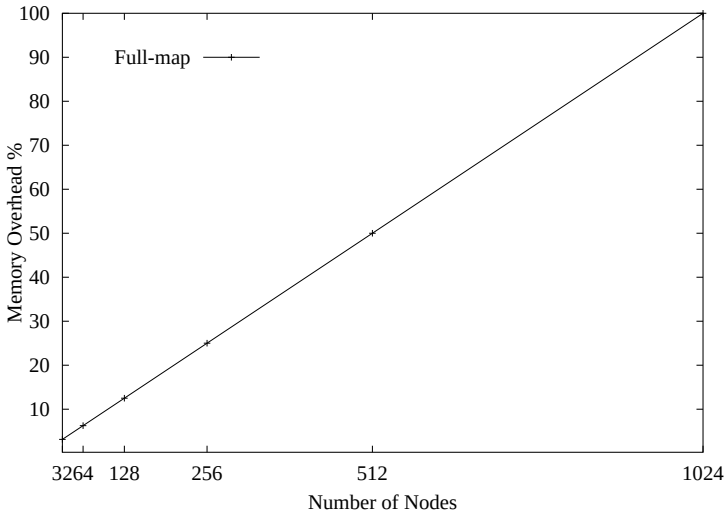


Figura 0.2: Sobrecarga de memoria para el código de compartición *full-map*

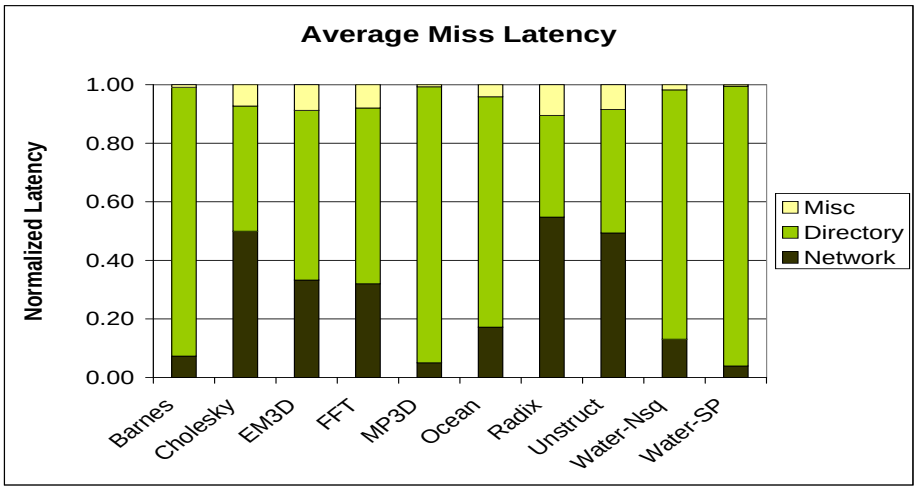


Figura 0.3: Latencia media de los fallos de L2

con 1024 nodos.

Por otro lado, las largas latencias de los fallos de L2 que caracterizan a las máquinas cc-NUMA están causadas fundamentalmente por la indirección que supone el uso de los directorios. Como consecuencia de la disparidad cada vez mayor existente entre las velocidades de las memorias y de los procesadores, y debido a que la información de directorio suele estar almacenada en la memoria principal, la componente más importante de la latencia de los fallos de L2, en la mayoría de los casos, es la correspondiente al directorio, esto es, el número de ciclos que los directorios tardan en resolver cada fallo de L2. La Figura

0.3 muestra las latencias medias de los fallos de L2 para las aplicaciones utilizadas en la presente tesis. Como puede observarse, para 7 de las 10 aplicaciones, la mayor parte de los ciclos que en promedio se requieren para resolver los fallos de L2 se consumen en el directorio.

0.1.1. Contribuciones de la Tesis

El objetivo de la presente tesis es el de proponer soluciones para los dos problemas que acabamos de presentar. En concreto, podemos identificar las contribuciones siguientes:

- **Técnicas para reducir la sobrecarga de memoria debida a los directorios**. En esta tesis presentamos el concepto de *agrupamiento multicapa* y lo utilizamos para obtener 3 nuevos códigos de compartición comprimidos con un tamaño menor que los ya existentes, y lo que es más importante, se propone una nueva arquitectura para los directorios, que hemos denominado *directorios de dos niveles*, a través de la cual se consigue reducir drásticamente la sobrecarga de memoria a la misma vez que se mantiene el rendimiento proporcionado por un directorio *full-map* no escalable.

- **Técnicas para reducir la latencia de los fallos de L2**. En este aspecto, tres son las contribuciones realizadas por esta tesis. Primero, proponemos una nueva arquitectura a través de la cual se consigue reducir de forma significativa la latencia de los fallos de L2. Para ello proponemos una clasificación de los fallos de L2 en términos de las acciones que el directorio realiza para servir cada fallo y hacemos uso de la posibilidad que hoy día existe de integrar componentes clave dentro del procesador (como, por ejemplo, se hizo en el diseño del Alpha 21364 EV7 [35]). Además de esta técnica general, a través de la cual conseguimos acelerar todos los tipos de fallos de L2, proponemos dos técnicas específicas para reducir la latencia de unos tipos de fallos muy particulares. Ambas técnicas, que son ortogonales a la anterior, están basadas en el uso de predicción y, por lo tanto, requieren del desarrollo de predictores eficientes así como la introducción de ciertas extensiones en el protocolo de coherencia del multiprocesador. Primero presentamos la predicción de propietario: esta técnica es aplicable a los fallos *3-hop*, los cuales ocurren muy frecuentemente en las aplicaciones actuales (más del 60 % de los fallos de L2 son *3-hop* en algunas ocasiones). La segunda de las técnicas, denominada predicción de compartidores, trata de acelerar los fallos *upgrade*. Ambos tipos de fallos se caracterizan por no solicitar del

directorio una copia de la línea de memoria sino, simplemente, el envío de ciertos mensajes de coherencia.

0.2. Entorno de Evaluación

A la hora de llevar a cabo la evaluación de las propuestas presentadas en la presente tesis, decidimos hacer uso del simulador RSIM v.1.0 (***R**ice **S**imulator for **I**LP **M**ultiprocessors*), el cual está disponible de forma pública [43]. RSIM es un simulador guiado por la ejecución (*execution-driven*) que permite llevar a cabo la simulación de un multiprocesador cc-NUMA actual con un alto grado de detalle, y ofrece la posibilidad de seleccionar los valores de muchos de los parámetros de la arquitectura. El simulador modela un procesador *superescalar* en cada uno de los nodos (muy parecido a un procesador R10000 [91]), la jerarquía de memoria (2 niveles de memoria caché, el bus del sistema y el directorio, que se encuentra en memoria principal) así como una red de interconexión escalable con topología de malla 2-D. La Tabla 0.2 resume los valores de los parámetros que hemos utilizado en nuestras simulaciones. Estos valores son representativos de los multiprocesadores cc-NUMA actuales. Hemos elegido la consistencia secuencial como modelo de memoria siguiendo las recomendaciones dadas por Hill [40].

El simulador RSIM ha de utilizarse conjuntamente con algunos programas de prueba (*benchmarks*) que son ejecutados por el mismo. En nuestro caso particular, hemos seleccionado diez programas de prueba: BARNES-HUT, CHOLESKY, FFT, OCEAN, RADIX,

Benchmark	Problem Size	Max Processors	Memory lines	Dominant Sharing Patterns
BARNES-HUT	8192 bodies	64	22.32 K	Wide sharing
CHOLESKY	tk15.O	64	249.74 K	Migratory
EM3D	38400 nodes	64	120.55 K	Producer-consumer
FFT	256K complex doubles	64	200.87 K	Producer-consumer
MP3D	48000 nodes	64	30.99 K	Migratory
OCEAN	258x258 ocean	32	248.48 K	Producer-consumer
RADIX	2M keys	64	276.30 K	Producer-consumer
UNSTRUCTURED	Mesh.2K	32	13.54 K	Producer-consumer and migratory
WATER-NSQ	512 molecules	64	71.81 K	Migratory
WATER-SP	512 molecules	64	5.85 K	Wide sharing

Tabla 0.1: Resumen de los tamaños de problema, número máximo de procesadores que pueden usarse, número total de líneas de memoria compartida y patrones de compartición principales, para las aplicaciones empleadas en esta tesis

ILP Processor	
Processor Speed	1 GHz
Max. fetch/retire rate	4
Active List	64
Functional Units	2 integer arithmetic
	2 floating point
	2 address generation
Branch Predictor	2-bit history predictor
Counters in the Branch Predictor Buffer	512
Shadow Mappers	8
Memory queue size	32 entries
Memory Hierarchy Parameters	
Cache line size	64 bytes
L1 cache (on-chip, WT)	Direct mapped, 32KB
L1 request ports	2
L1 hit time	2 cycles
L2 cache (on-chip, WB)	4-way associative, 512KB
L2 request ports	1
L2 hit time	15 cycles, pipelined
Number of MSHRs	8 per cache
Memory access time	70 cycles (70 ns)
Memory interleaving	4-way
Cache-coherence protocol	MESI
Consistency model	Sequential consistency
Directory Cycle	10 cycles
First coherence message creation time	4 directory cycles
Next coherence messages creation time	2 directory cycles
Bus Speed	1 GHz
Bus width	8 bytes
Network Parameters	
Router speed	250 MHz
Channel width	32 bits
Channel speed	500 MHz
Number of channels	1
Flit size	8 bytes
Non-data message size	16 bytes
Router's internal bus width	64 bits
Arbitration delay	1 router cycle

Tabla 0.2: Valores de los parámetros usados en las simulaciones

WATER-SP y WATER-NSQ, de la *suite* SPLASH-2 [90], EM3D, que es una implementación para memoria compartida de la versión Split-C [22], MP3D, de la *suite* SPLASH [82], y, finalmente, UNSTRUCTURED, que es una aplicación para dinámica de fluidos [67]. La Tabla 0.1 muestra los programas de evaluación que hemos utilizado en esta tesis, junto con los tamaños de problema usados en cada caso. Estos tamaños han sido seleccionados en función del número máximo de procesadores disponibles en cada caso y que se muestra en la tercera columna de la tabla.

0.3. Agrupamiento Multicapa y Directorios de Dos Niveles

Existen dos alternativas fundamentales a la hora de almacenar la información de directorio [23]: los esquemas basados en memoria con nodos *home* fijos (*flat, memory-based directories*) y los esquemas basados en caché con nodos *home* fijos (*flat, cache-based directories*). En los protocolos de coherencia que utilizan el primer tipo de esquemas, el nodo *home* que cada línea de memoria tiene asignado almacena información sobre el estado de la línea así como la lista de nodos que mantienen una copia de la misma. Por el contrario, para los protocolos de coherencia basados en la segunda de las alternativas, el nodo *home* almacena el estado de la línea y un puntero al primero de los compartidores (en lugar de la lista completa de compartidores almacenada en el caso anterior). El resto de nodos que mantienen una copia de la línea de memoria se disponen formando una lista doblemente enlazada mediante el uso de punteros adicionales asociados a cada *línea de cache* en cada nodo.

Aunque los directorios basados en caché con nodos *home* fijos pueden reducir de forma significativa la cantidad de memoria necesaria para el directorio y han sido utilizados en algunos multiprocesadores comerciales como el multiprocesador Sequent NUMA-Q [59], los inconvenientes de este tipo de esquemas han provocado que la mayoría de los multiprocesadores de memoria compartida que se construyen hoy en día hagan uso de protocolos pertenecientes al primer grupo, especialmente a partir de la introducción del multiprocesador Origin 2000 [54]. Algunos ejemplos son los multiprocesadores Piranha [12], AlphaServer GS320 [28] y Cenju-4 [42].

Por lo tanto en esta tesis nos centramos en los protocolos de coherencia pertenecientes a este primer grupo. La Figura 0.4 muestra cómo se organiza el directorio en estos protocolos. Cada línea de memoria tiene asociada una *entrada de directorio* que se almacena en el nodo *home* correspondiente, al lado de la línea (normalmente en memoria principal). La entrada de directorio de una determinada línea de memoria se utiliza para almacenar el *estado* de la misma así como su *código de compartición*, el cual indica los nodos que mantienen una copia de la línea. Este último elemento consume la mayoría de los bits de cada entrada de directorio y su elección determina, de forma directa, la cantidad de memoria requerida para el directorio (por ejemplo, la sobrecarga de memoria introducida por el código de compartición *full-map* para una configuración de 1024 procesadores y para líneas de memoria de 128 bytes es del 100 %).

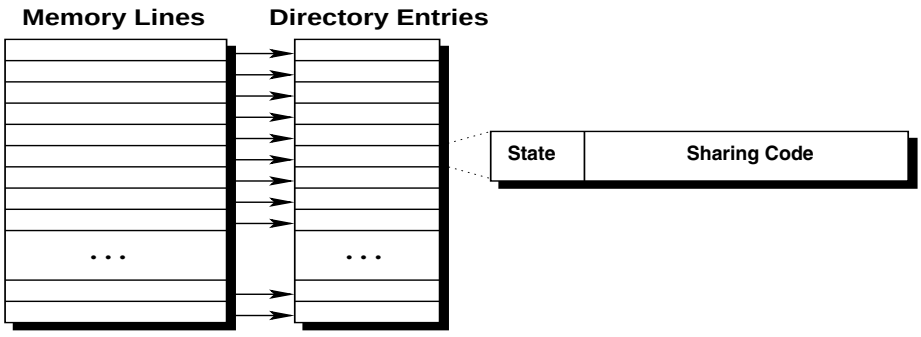


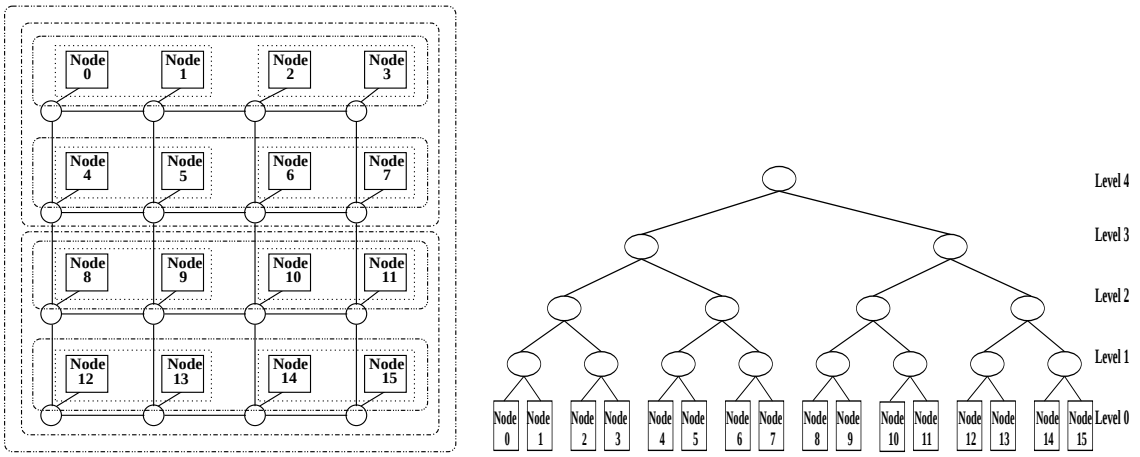
Figura 0.4: Organización del directorio para la familia de protocolos de coherencia asumidos en esta tesis

Con el objetivo de reducir la *anchura* de cada una de las entradas del directorio en configuraciones con un número considerable de nodos y, por lo tanto, la sobrecarga de memoria debida al directorio, se han propuesto los *códigos de compartición comprimidos*. Estos códigos de compartición requieren menos bits debido a que almacenan una representación *en exceso* de los compartidores de una línea de memoria en un instante determinado (se almacena un *superconjunto* del conjunto de los compartidores reales). De esta forma, comparados con códigos de compartición *precisos* como *full-map* ó *punteros limitados* que identifican exactamente qué nodos mantienen una copia de la línea, los códigos de compartición comprimidos introducen una pérdida de precisión que tiene consecuencias negativas en el rendimiento de las aplicaciones paralelas. Existen dos motivos fundamentales por los que los códigos de compartición comprimidos pueden aumentar el tiempo de ejecución de las aplicaciones:

1. Aparición de *mensajes de coherencia innecesarios*. Es decir, mensajes de coherencia que aparecen como consecuencia de la codificación en exceso que se realiza de los compartidores. Estos mensajes consumen recursos del sistema (congestionan la red y los controladores tanto de caché como de directorio) y son inútiles, es decir, no serían enviados si se dispusiera de un código de compartición preciso.

2. Se necesitan más mensajes para detectar ciertas condiciones de carrera.

Otra forma de reducir la cantidad de memoria requerida por los directorios consiste en disminuir la *altura* de los mismos. Es decir, en lugar de tener una entrada de directorio por cada línea de memoria, se dispone de una caché que almacena información de directorio para las líneas de memoria más recientemente usadas. El problema de utilizar un



(a) Sistema *físico* (b) Sistema *lógico*

Figura 0.5: Ejemplo de aplicación del agrupamiento multicapa

directorio de este tipo es la aparición de *invalidaciones innecesarias* como consecuencia de los reemplazos que tienen lugar en este tipo de estructuras. Estas invalidaciones innecesarias pueden aumentar drásticamente el número de fallos de L2 y, por lo tanto, afectar negativamente al rendimiento final.

En esta tesis presentamos una nueva arquitectura de directorio, denominada *directorio de dos niveles*, que ofrece las ventajas de ambos tipos de soluciones en un único diseño. Una organización de directorio de dos niveles tiene la capacidad de reducir de forma sustancial los requerimientos de memoria sin afectar negativamente al rendimiento final.

0.3.1. Agrupamiento Multicapa

El objetivo del *agrupamiento multicapa* [1] es el de reducir el tamaño del código de compartición. Se realiza una distinción entre el sistema *físico* (tal y como es) y el sistema *lógico* (tal y como es visto por los directorios). La idea es la de formar un sistema lógico en el que todos los nodos se agrupan recursivamente en grupos de igual tamaño (en nuestro caso, 2) hasta que se obtenga un único grupo. La compresión se alcanza especificando el nivel del grupo mínimo en la jerarquía en el que están contenidos todos los compartidores. En nuestro caso este agrupamiento recursivo en grupos produce como resultado un árbol binario lógico en el que los nodos se encuentran en las hojas del árbol.

Por ejemplo, para un multiprocesador de 16 nodos que utiliza una malla como red de

interconexión, la Figura 5(a) muestra el sistema físico, mientras que la Figura 5(b) muestra el sistema lógico que se obtiene al aplicar el agrupamiento multicapa. A partir de este sistema lógico derivamos tres nuevos códigos de compartición comprimidos:

1. **Árbol binario (*binary tree* ó *BT*)**. Los nodos que almacenan una copia de una determinada línea de memoria se expresan codificando el nivel de la raíz del subárbol mínimo que contiene a todos los compartidores. Esto puede ser expresado utilizando un total de $\lceil \log_2 (\log_2 N + 1) \rceil$ bits.

2. **Árbol binario con nodos simétricos (*binary tree with symmetric nodes* ó *BT-SN*)**. Se introduce el concepto de nodos simétricos de un determinado nodo *home*. Si asumimos un total de 3 nodos simétricos por cada nodo home, éstos son codificados a través de las diferentes combinaciones de los 2 bits más significativos del identificador del nodo *home* (observar que una de las cuatro combinaciones representa al propio nodo *home*). Ahora el proceso de elegir el subárbol mínimo que incluya a todos los compartidores se repite para los nodos simétricos. Se elige entonces el mínimo de los 4 subárboles para representar a los compartidores (el calculado desde el nodo *home* y los calculados desde los 3 simétricos). El tamaño del código de compartición es el mismo que el del caso anterior más los 2 bits necesarios para codificar el simétrico que se usa.

3. **Árbol binario con subárboles (*binary tree with subtrees* ó *BT-SuT*)**. Este esquema resuelve el caso común de tener un único compartidor codificando directamente la identidad del nodo (para lo que hacen falta, al menos, $\log_2 N$ bits). Cuando varios nodos tienen copia de la línea se utiliza una representación alternativa: se emplean 2 subárboles (en lugar de 1) para incluir a todos los compartidores. Uno de ellos se calcula desde el nodo *home*, mientras que el otro se calcula desde uno de los 3 nodos simétricos. Por lo tanto, el tamaño total de este código de compartición es de $\text{máx}\{(1+\log_2 N), (1+2+2\lceil \log_2 (\log_2 N) \rceil)\}$ bits.

De los tres códigos de compartición comprimidos que se proponen en esta tesis, *BT-SuT* es el que obtiene los mejores resultados en términos de número de mensajes de coherencia innecesarios introducidos. Los otros dos códigos de compartición, *BT* y *BT-SN*, son demasiado agresivos (es decir, comprimen en demasía la información de compartición) y, como resultado, incrementan en gran medida el tiempo de ejecución de las aplicaciones.

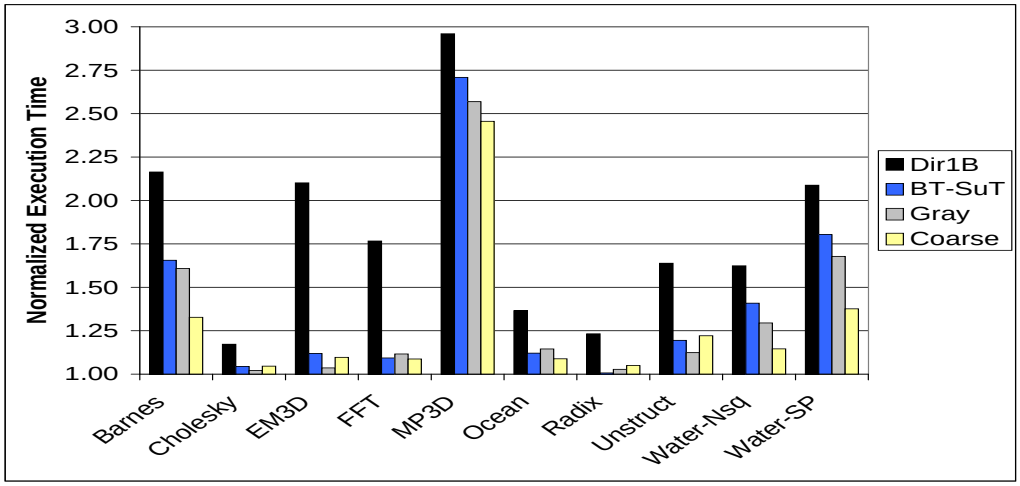


Figura 0.6: Tiempos de ejecución normalizados para los códigos de compartición *coarse vector*, *gray-tristate*, *BT-SuT* y Dir_1B

La Figura 0.6 compara *BT-SuT* con *coarse vector* [33], *gray-tristate* [65] y Dir_1B, en términos del tiempo ejecución obtenido para las aplicaciones usadas en esta tesis. El tiempo de ejecución obtenido en cada caso ha sido normalizado con respecto al que se conseguiría con un directorio *full-map*. Los dos primeros códigos de compartición comprimidos introducen una sobrecarga de memoria mayor, mientras que la inducida por Dir_1B es ligeramente inferior.

Cuando comparamos nuestro esquema más elaborado con Dir_1B, podemos observar en la Figura 0.6 cómo *BT-SuT* obtiene tiempos de ejecución inferiores para todas las aplicaciones. A pesar de que ambos códigos de compartición tienen un tamaño similar, *BT-SuT* realiza un uso más eficiente de los bits que están disponibles. Por otro lado, para las aplicaciones CHOLESKY, EM3D, FFT, OCEAN, RADIX and UNSTRUCTURED, los resultados obtenidos por *coarse vector*, *gray-tristate* y *BT-SuT* son muy parecidos, incluso *BT-SuT* consigue mejores resultados que estos dos códigos de compartición en algunos casos. Para el resto de las aplicaciones, *BT-SuT* logra tiempos de ejecución similares a los obtenidos con *gray-tristate* aunque los mejores resultados se obtienen al utilizar *coarse vector* como código de compartición. Es importante hacer notar que las diferencias, sin embargo, no son excesivamente importantes. El código de compartición *BT-SuT* requiere aproximadamente la mitad de los bits que son necesarios para *gray-tristate* y no tiene los problemas de escalabilidad que *coarse vector* presenta. Por lo tanto, podemos concluir que *BT-SuT* logra la mejor relación entre la sobrecarga de memoria introducida y la degradación del

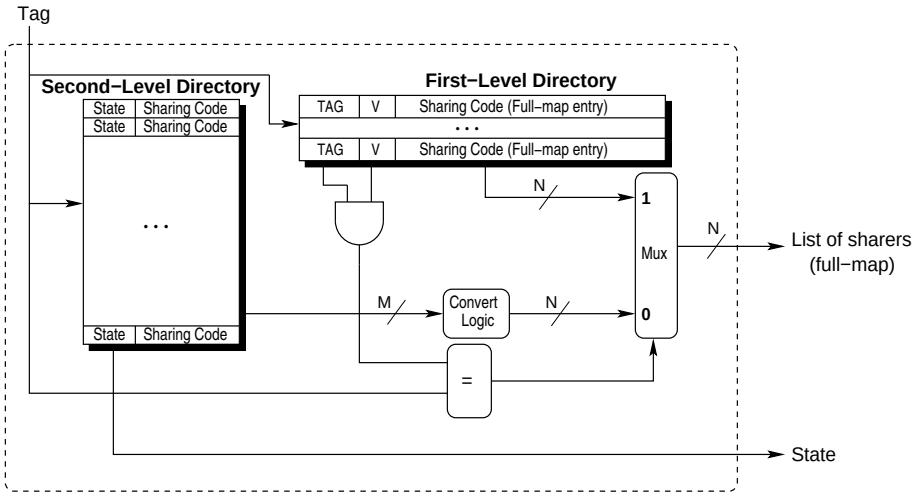


Figura 0.7: Arquitectura de un directorio de dos niveles

rendimiento experimentada como consecuencia de la compresión de la información de directorio.

0.3.2. Directorios de Dos Niveles

Aunque los directorios comprimidos (esto es, aquellos que utilizan un código de compartición comprimido) consiguen reducir de forma considerable la cantidad de memoria necesaria para almacenar la información de directorio, las consecuencias negativas sobre el tiempo de ejecución de las aplicaciones que implican reduce la utilidad de los mismos.

En esta tesis presentamos una *arquitectura de directorio de dos niveles* [1] para reducir de manera significativa la cantidad de memoria dedicada al directorio y, a la misma vez, obtener el rendimiento de un directorio *full-map* que por definición no es escalable. Como se muestra en la Figura 0.7, un directorio de dos niveles consta de:

1. *Directorio de primer nivel*, que es una caché de directorio que mantienen información de directorio precisa para las líneas de memoria que han sido solicitadas más recientemente y que, por lo tanto, serán accedidas en un futuro cercano con más probabilidad. El código de compartición que hemos utilizado para este nivel ha sido *full-map*.

2. *Directorio de segundo nivel*, que es un directorio comprimido en el que se mantiene la información de directorio actualizada para todas las líneas de memoria que tiene asignadas el nodo *home*. Este nivel proporcionará la información de directorio sólo

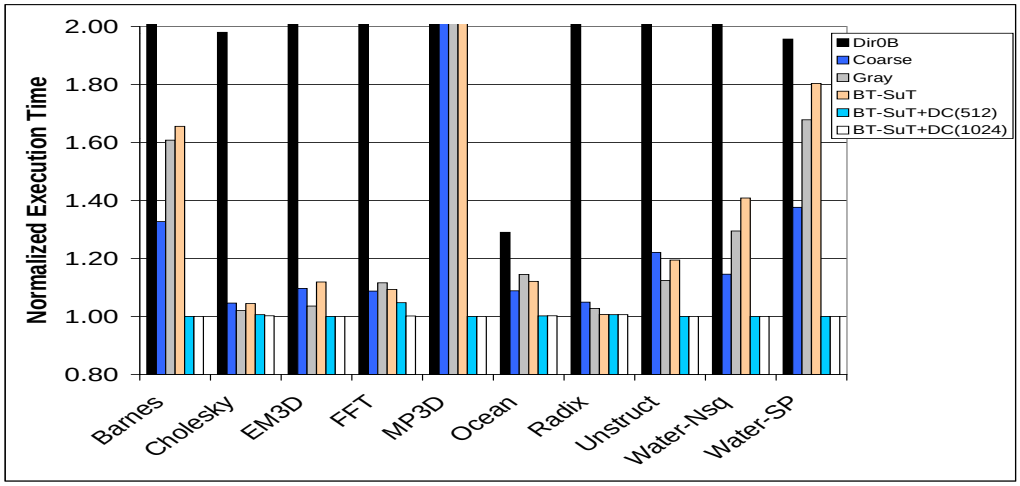


Figura 0.8: Tiempos de ejecución normalizados para la configuración *BT-SuT* y directorio de primer nivel(*FM*)

en aquellos casos en los que la entrada correspondiente no se encuentre en el directorio de primer nivel. Para este segundo nivel podría utilizarse cualquiera de los códigos de compartición comprimidos previamente presentados. En concreto, nos hemos decantado por *BT-SuT*, ya que es el que obtiene la mejor relación entre el costo en términos de requerimientos de memoria y el rendimiento.

La Figura 0.8 muestra los resultados que se obtienen para la arquitectura de directorio de dos niveles previamente explicada cuando el directorio de primer nivel dispone de 512 y 1024 entradas (barras *BT-SuT+DC(512)* y *BT-SuT+DC(1024)*, respectivamente). Se muestran, además, los tiempos de ejecución obtenidos con los distintos códigos de compartición comprimidos. De nuevo, los tiempos de ejecución han sido normalizados con respecto a los obtenidos con un directorio *full-map*. Podemos observar cómo el rendimiento de un directorio *full-map* puede alcanzarse cuando cada uno de los nodos del multiprocesador incluye un directorio de dos niveles en el que se combinan un pequeño directorio de primer nivel (4 KB para la configuración de 512 entradas) y un directorio completo (una entrada por cada línea de memoria) y comprimido (utilizando *BT-SuT* como código de compartición).

0.4. Una Nueva Arquitectura para Reducir la Latencia de los Fallos de L2

Como ya apuntamos en la Sección 0.1, las largas latencias de los fallos de L2 en las que normalmente incurren los multiprocesadores cc-NUMA constituyen otro de los factores que limitan la escalabilidad de este tipo de arquitecturas. El motivo de estas importantes latencias es la indirección que supone el acceso a la información de directorio y, más concretamente, el número de ciclos que el directorio *home* requiere para llevar a cabo el procesamiento de cada fallo de L2 (la *componente de directorio* de la latencia). Esto incluye el tiempo necesario para acceder a memoria principal, lugar en el que normalmente se ubica la información de directorio.

Desafortunadamente, la tendencia hacia microprocesadores cada vez más rápidos no se sigue en igual medida en el caso de la memoria, cuya velocidad crece mucho más lentamente [36]. Este hecho acentúa aún más el problema y tiene especial repercusión sobre aquellos fallos de L2 en los que el directorio *home* no tiene que proporcionar la línea de memoria, sino únicamente enviar mensajes de coherencia a los nodos que mantienen una copia de la línea. Al contrario que para las máquinas cc-NUMA, los multiprocesadores SMP no requieren el acceso a la memoria principal para este tipo de fallos, lo que reduce de manera considerable su latencia y, como consecuencia, aumenta la popularidad de los mismos.

Por otro lado, el número de transistores que pueden proporcionarse en un único chip aumenta cada vez más como consecuencia de las mejoras tecnológicas. Esto permite a los arquitectos de computadores incluir componentes claves dentro del chip del procesador principal. Por ejemplo, el procesador Alpha 21364 EV7 diseñado por Compaq [35] incluye dentro del procesador el controlador de memoria, la circuitería de coherencia así como el interfaz de red y el encaminador.

En esta tesis hacemos uso de esta escala de integración con el objetivo de derivar una *nueva arquitectura* [2] capaz de reducir, de forma considerable, la latencia de los fallos de L2. Nuestra propuesta, cuyo punto de partida es una organización como la del Alpha 21364 EV7, reemplaza el directorio tradicional por un directorio de tres niveles (que es una extensión de la arquitectura de directorios de dos niveles presentada con anterioridad) y la adición de una caché para datos compartidos. El directorio de primer nivel y la caché para datos compartidos se incluyen dentro del chip del procesador, junto con la circuitería de coherencia.

0.4.1. Clasificación de los Fallos de L2

Considerando las acciones realizadas por el directorio *home* para resolver los fallos de L2, éstos pueden clasificarse en cuatro categorías (suponiendo los estados MESI para las cachés):

1. Fallos *$-to-$*: el directorio tiene que re-enviar el fallo al único nodo que mantiene una copia de la línea de memoria.
2. Fallos *Mem*: el directorio tiene que proporcionar directamente la línea de memoria.
3. Fallos *Inv*: el directorio tiene que invalidar todas las copias de la línea de memoria salvo la que el nodo peticionario mantiene.
4. Fallos *Inv+Mem*: el directorio tiene que invalidar primero todas las copias de la línea de memoria para enviar, a continuación, una copia de la misma al nodo peticionario.

0.4.2. Una Nueva Arquitectura de Nodo

Partiendo de una arquitectura como la del Alpha 21364 EV7, la arquitectura de nodo que presentamos en esta tesis y que mostramos en la Figura 0.9, añade los siguientes elementos:

- **Directorio de tres niveles**. Este directorio reemplaza al directorio tradicional que se ubica en memoria principal. El directorio de tres niveles que proponemos constituye una extensión de la arquitectura de directorio de dos niveles que presentamos con anterioridad [4]. En particular, distinguimos los siguientes niveles de directorio:
  - *Directorio de primer nivel*. Es una pequeña caché de directorio que ubicamos dentro del chip del procesador. Cada entrada en este directorio dispone de un número limitado de punteros como código de compartición. En concreto, hemos observado que disponer de 3 punteros por cada entrada de directorio es suficiente para la mayoría de los casos y, por lo tanto, cada entrada de directorio en este nivel puede almacenar la identidad de hasta 3 compartidores.
  - *Directorio de segundo nivel*. Este nivel constituye una *caché de víctimas* del primero. Es decir, en este nivel vamos a almacenar todas las entradas de directorio que fueron desalojadas del primer nivel, bien como consecuencia de

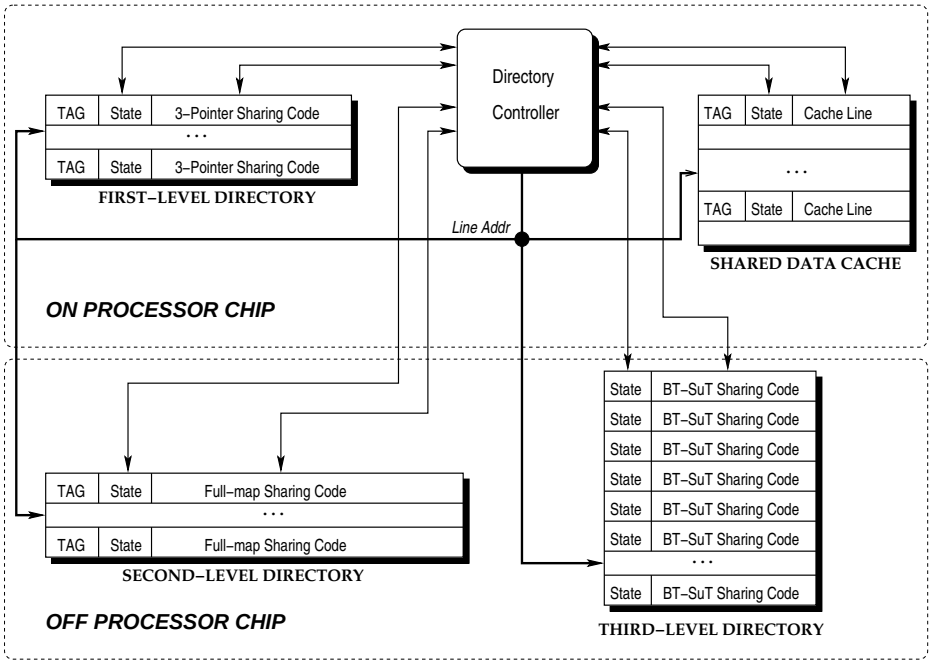


Figura 0.9: Arquitectura propuesta para los nodos de un multiprocesador cc-NUMA

un reemplazo, bien debido a que el número de compartidores en un instante determinado supera el límite de los de los 3 que pueden almacenarse en cada entrada del directorio de primer nivel. Este directorio de segundo nivel, que usa como código de compartición *full-map*, lo implementamos haciendo uso de una caché de directorio y lo ubicamos fuera del chip del procesador. Es importante reseñar que la información de directorio para una determinada línea de memoria sólo puede residir en uno de los dos primeros niveles, nunca en ambos.

- *Directorio de tercer nivel*. Es un directorio comprimido que se ubica al lado de la memoria principal. El directorio de tercer nivel mantiene información de compartición actualizada para todas las líneas de memoria y es accedido cuando la entrada de directorio que se busca no está disponible en los dos niveles inferiores. Este nivel de directorio emplea *BT-SuT* como código de compartición comprimido.

Mientras que a través del directorio de primer nivel se consiguen reducciones sustanciales de la latencia de los fallos de L2, muchos de los cuales pueden ser satisfechos rápidamente desde el propio chip del procesador, el cometido fundamental de los dos

niveles restantes es el de reducir la sobrecarga de memoria debida al directorio. Como puede observarse, la arquitectura de directorio de tres niveles que acabamos de describir se deriva de la arquitectura de dos niveles que propusimos con anterioridad, añadiendo únicamente lo que ahora denominamos el primer nivel.

- **Caché para datos compartidos**. Esta estructura es una pequeña caché para datos que ubicamos dentro del chip del procesador, junto al directorio de primer nivel. El propósito de la caché para datos compartidos es el de almacenar aquellas líneas de memoria que el directorio *home* probablemente tendrá que proporcionar y que, de otra forma, tendrían que ser buscadas en memoria principal.

A través del diseño que acabamos de presentar, podemos reducir de forma significativa el número de ciclos que el directorio *home* requiere, para cada una de las categorías de los fallos de L2 anteriormente presentadas. Para los fallos *$-to-$* (también conocidos como fallos *3-hop*), el directorio *home* puede re-enviar rápidamente la petición al propietario actual de la línea de memoria cuando la entrada de directorio correspondiente se encuentra en el directorio de primer nivel y, en menor medida, cuando lo hace en el directorio de segundo nivel. De manera similar, el proceso de invalidación para los fallos *Inv* podrá comenzar antes cuando el directorio de primer o segundo nivel proporciona la información de compartición. Como comentamos con anterioridad, para los fallos *Mem* el directorio debe proporcionar la línea de memoria y, por lo tanto, los fallos pertenecientes a esta categoría serán acelerados cuando ésta se encuentre en la caché para datos compartidos. Finalmente, hemos observado cómo el directorio emplea la mayor parte de los ciclos necesarios para resolver un fallo *Inv+Mem* realizando el proceso de invalidación de los compartidores. De ésta forma, el encontrar la línea de memoria en la caché para datos compartidos no supone ningún beneficio en cuanto a reducción de la latencia de estos fallos, ya que la línea no es requerida antes de que el proceso de invalidación haya concluido. Igualmente, el no hallar la línea en la caché para datos compartidos no supone perjuicio alguno, debido a que el acceso a memoria principal ocurriría en paralelo con la invalidación de los compartidores. Como para el caso *Inv*, los beneficios que este tipo de fallos pueden obtener de la arquitectura propuesta radican en que el proceso de invalidación puede comenzar antes cuando la información de directorio se encuentra en uno de los dos primeros niveles.

Las Figuras 0.10, 0.11, 0.12 y 0.13 reflejan cuantitativamente los beneficios (en términos de reducción de las latencias medias normalizadas) conseguidos para cada una de las categorías de los fallos de L2. En todos los casos se comparan el sistema base (*Base*),

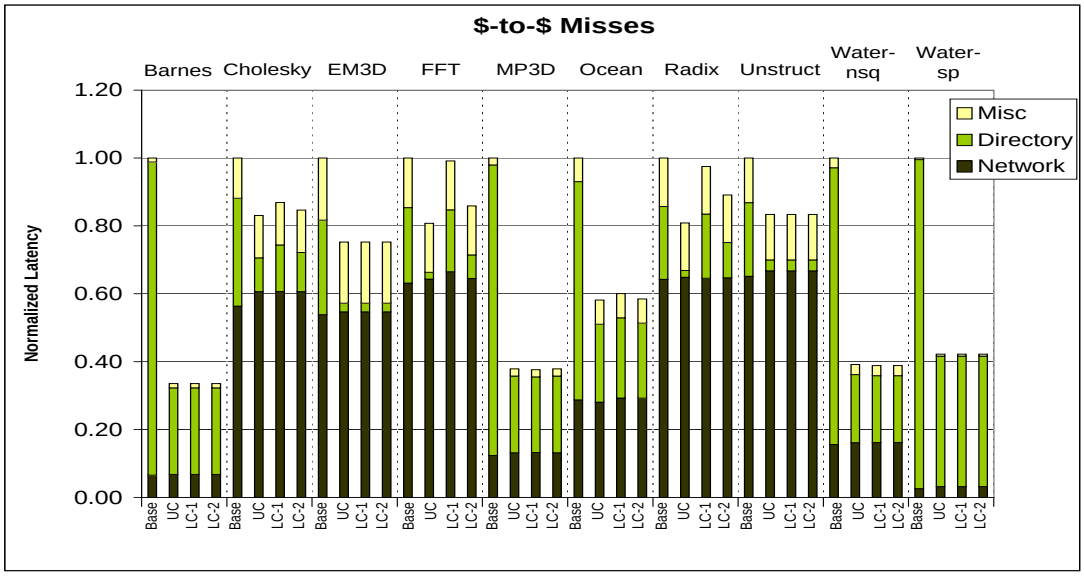


Figura 0.10: Latencia media de los fallos *$-to-$*

dos configuraciones basadas en la arquitectura presentada con tamaños diferentes para los directorios de primer y segundo nivel y para la caché para datos compartidos (*LC-1* y *LC-2*) y, por último, una configuración que nos proporciona una aproximación al potencial de nuestra propuesta mediante la utilización de un número ilimitado de entradas en las cachés de directorio y en la de datos compartidos (*UC*). Las latencias medias normalizadas mostradas en cada una de las figuras aparecen divididas en tres componentes: latencia de red, latencia de directorio y latencia miscelánea (ciclos empleados en las cachés, el bus...).

La Figura 0.10 muestra la latencia media normalizada de los fallos pertenecientes a la primera de las cuatro categorías anteriores (es decir, los fallos *$-to-$*). Como puede observarse, las reducciones de latencia que se obtienen cuando se usa la configuración *LC-2* van desde un 11 % para RADIX hasta un 66 % para BARNES. Estas reducciones están muy próximas a las que se consiguen con cachés ilimitadas para los directorios de primer y segundo nivel y para la caché para datos compartidos (configuración *UC*).

La Figura 0.11 presenta las latencias medias normalizadas para los fallos *Mem*. En este caso, las reducciones de latencia van desde un 10 % para RADIX hasta un 45 % para BARNES cuando utilizamos la configuración *LC-2*. Para todas las aplicaciones salvo para FFT, los resultados que obtiene esta configuración coinciden prácticamente con los mostrados para la configuración *UC*.

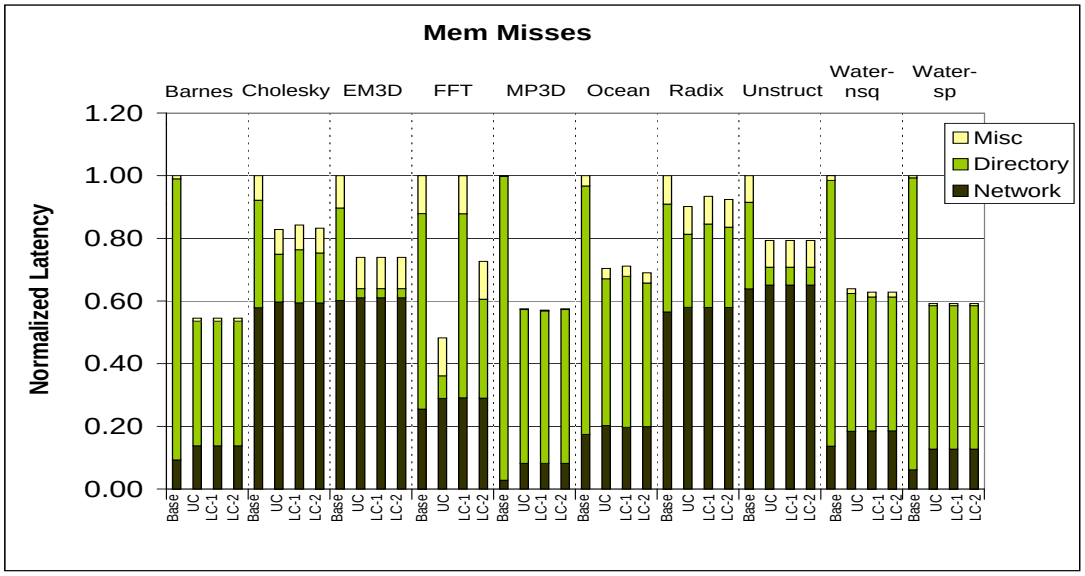


Figura 0.11: Latencia media de los fallos *Mem*

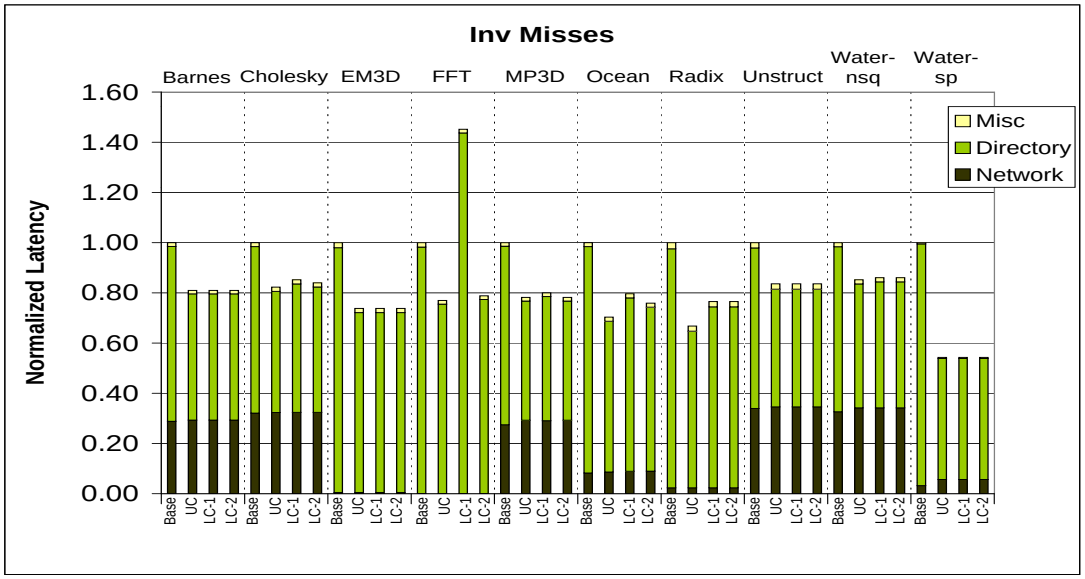


Figura 0.12: Latencia media de los fallos *Inv*

Como se presenta en la Figura 0.12, las reducciones en términos de latencia media normalizada para los fallos *Inv* van desde un 15 % para CHOLESKY y WATER-NSQ, hasta el 45 % observado para WATER-SP, sobre todo cuando usamos la configuración *LC-2*. Es importante reseñar que la degradación encontrada en FFT para la configuración *LC-1* es consecuencia del gran número de fallos *Inv* para los que se necesita acceder al directorio

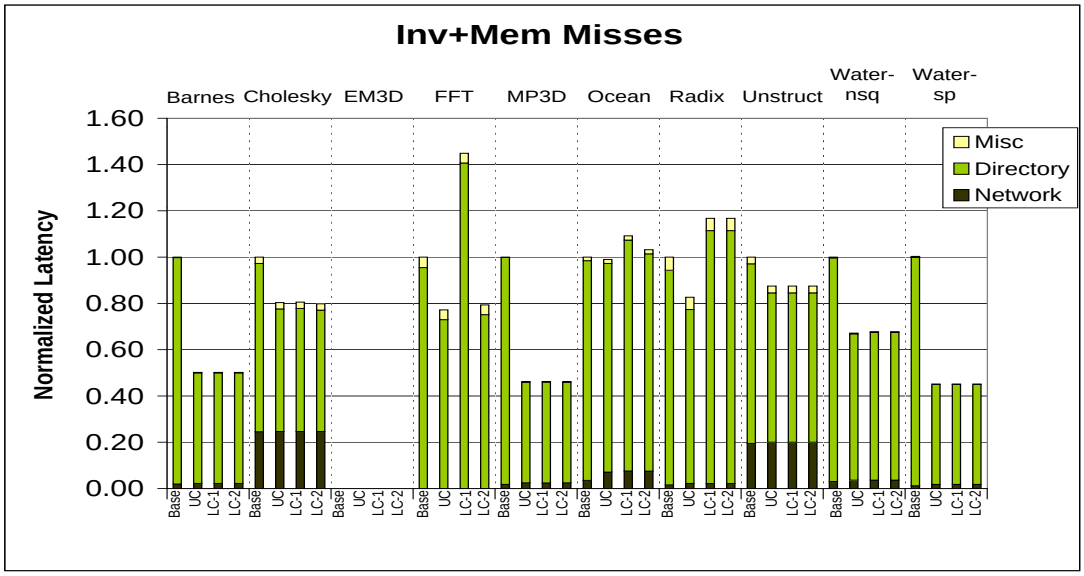


Figura 0.13: Latencia media de los fallos *Inv+Mem*

comprimido de tercer nivel. Esto es debido a que la entrada de directorio correspondiente no se encuentra en los dos primeros niveles y, como puede verse, desaparece completamente cuando pasamos a utilizar la configuración *LC-2*. Podemos ver también cómo los resultados conseguidos por la configuración *LC-2* están muy próximos a los de la configuración *UC*.

Por último, la Figura 0.13 presenta las latencias medias normalizadas para todas las aplicaciones en las que ocurren fallos *Inv+Mem* (es decir, todas salvo EM3D). Las reducciones obtenidas en este caso cuando se emplea la configuración *LC-2* van desde un 13 % para UNSTRUCTURED hasta un 55 % para WATER-SP. Como puede observarse, para algunas aplicaciones como OCEAN y especialmente RADIX esta configuración no sólo no es capaz de reducir la latencia de este tipo de fallos, sino que la aumenta ligeramente debido a que para muchos de estos fallos es preciso llegar al directorio de tercer nivel para encontrar la información de directorio.

0.5. Predicción de Propietario para Acelerar los Fallos *3-hop*

Los fallos *$-to-$* (también conocidos como *3-hop* ó transferencias de caché a caché) aparecen frecuentemente en aplicaciones científicas y, sobre todo, en aplicaciones comer-

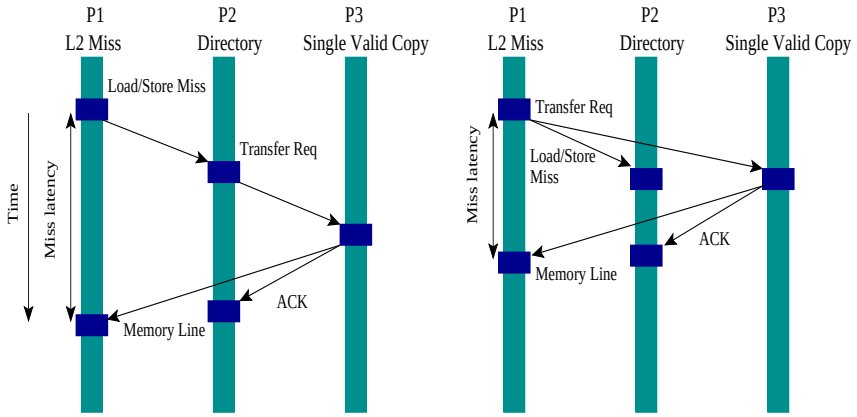


Figura 0.14: Proceso de resolución de un fallo *3-hop* en un multiprocesador cc-NUMA convencional (izquierda) y en uno que usa predicción de propietario (derecha)

ciales [11]. Es por este motivo por el que estos fallos están siendo objeto de un número importante de esfuerzos de investigación.

Para este tipo de fallos el directorio *home* no dispone de una copia actualizada de la línea de memoria, lo que provoca que éste tenga que re-enviar el fallo (como una *orden de transferencia*) al nodo propietario de la línea. Para determinar la identidad del nodo propietario, el directorio *home* hace uso de la información de directorio. Finalmente, cuando el nodo propietario recibe la orden de transferencia desde el directorio *home*, envía una copia de la línea de memoria al nodo que generó el fallo, actualiza el estado de su copia local e informa al directorio de que la transferencia fue realizada (dependiendo de si el fallo estaba causado por una instrucción de carga – *load* – o de almacenamiento – *store* –, este mensaje incluirá o no una copia de la línea de memoria). Todo este proceso se muestra en la parte izquierda de la Figura 0.14.

El hecho de que en una arquitectura cc-NUMA sea preciso acceder a la información de directorio para conocer la identidad del propietario penaliza a los fallos *3-hop* comparado con los diseños SMP, en los que los éstos van directamente a la caché donde reside la única copia válida de la línea. Con el objetivo de eliminar completamente el acceso al directorio de la ruta crítica de los fallos *3-hop*, en esta tesis presentamos una nueva técnica conocida como *predicción de propietario* [3]. Como se muestra en la parte derecha de la Figura 0.14, nuestra técnica se basa en el uso de predictores para identificar aquellos fallos de L2 que son *3-hop* e intentar averiguar la identidad del nodo propietario. De esta forma, el nodo peticionario podría enviar directamente la orden de transferencia al nodo que se predice como propietario. En paralelo y con el objetivo de comprobar la predicción, el

nodo peticionario podría enviar también el fallo al directorio *home*. Si la predicción fue correcta (que es el caso presentado en la figura), el directorio esperaría a que llegara la confirmación que el nodo propietario envía anunciando que la transferencia fue realizada y, entonces, actualizaría la información de directorio. Observar que, en este caso, el acceso a la información de directorio sería completamente eliminado de la ruta crítica del fallo de L2. En otro caso, si la predicción fue incorrecta, el directorio realizaría la acción correctiva adecuada (por ejemplo, si no se acertó la identidad del propietario, el directorio enviaría la orden de transferencia al propietario real).

La técnica de predicción de propietario presentada en esta tesis consta de dos componentes clave: (1) un predictor capaz de obtener frecuencias de acierto elevadas y (2) un protocolo de coherencia extendido para soportar el uso de la predicción y, por lo tanto, resolver las nuevas condiciones de carrera que surgen. Hemos observado que los fallos *3-hop* se comportan normalmente de manera repetitiva, es decir, el número de instrucciones que los provocan es reducido, al igual que el número de propietarios diferentes desde los que el nodo peticionario recibe la línea de memoria. Este comportamiento, por lo tanto, podríamos capturarlo a través del uso de un predictor sencillo, que es lo que motiva la presente propuesta.

0.5.1. Predictor de Dos Etapas

El predictor propuesto para la técnica de predicción de propietario, y que se muestra en la Figura 0.15, realiza una predicción de dos etapas: en una primera etapa decide si un determinado fallo de L2 es *3-hop* o no. Si el fallo es predicho como *3-hop*, en una segunda etapa el predictor especula acerca de la identidad del propietario, proporcionando una lista de posibles candidatos.

De esta forma, el denominado *predictor de la primera etapa* se encarga principalmente de la identificación de los fallos *3-hop*. Este predictor se implementa sin hacer uso de *tags*, sino que se accede a él directamente utilizando el PC de la instrucción que provocó el fallo de L2. Cada una de las entradas de este primer predictor está formada por el campo *$-to-$ Confidence*, que es un contador de confianza de 2 bits que se usa para determinar si un determinado fallo es *3-hop* (cuando almacena valores de 2 o 3) o no (en otro caso), así como de un puntero capaz de almacenar la identidad de un posible propietario (*Pointer 1/1 1st Step*), junto con su contador de confianza (*Confidence Pointer 1*). El hecho de disponer de este puntero en este predictor está motivado por la observación de que, en algunos casos,

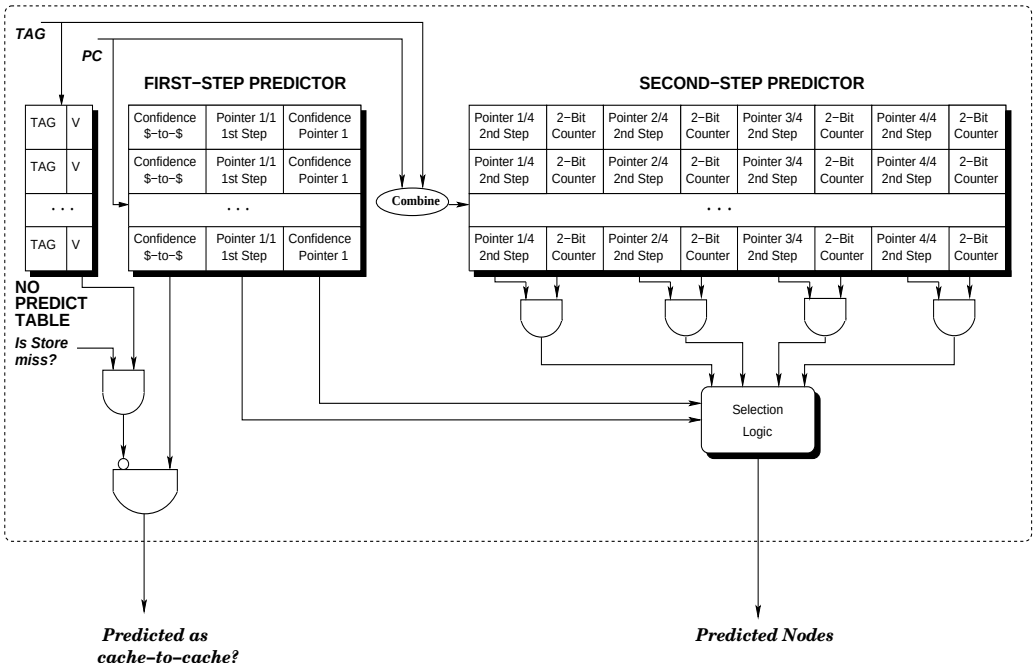


Figura 0.15: Anatomía del predictor de dos etapas propuesto para los fallos *3-hop*

los fallos *3-hop* causados por una misma instrucción reciben la línea de memoria desde el mismo nodo. En estos casos, como puede verse, el predictor de la primera etapa sería el responsable de ambas predicciones.

Sin embargo, en la mayoría de los casos es necesario disponer de un predictor más complejo para llevar a cabo la segunda de las predicciones. Al igual que para el predictor anterior, la implementación del *predictor de la segunda etapa* se lleva a cabo sin hacer uso de *tags*. En este caso, el acceso al predictor lo realizamos combinando el PC de la instrucción que causó el fallo y la dirección de la línea de memoria. Cada una de las entradas de este predictor dispone de hasta 4 punteros (cada uno de los campos *Pointer X/4 2nd Step*), junto con sus contadores de confianza respectivos. A la hora de elaborar la lista de posibles propietarios para un fallo de L2 que es predicho como *3-hop*, se tendrán en cuenta aquellos punteros para los que el contador de confianza correspondiente almacena los valores 2 ó 3.

0.5.2. Extensiones al Protocolo de Coherencia

Además de ofrecer un predictor con ratios de acierto altos, la técnica de predicción de propietario requiere la inclusión de una serie de cambios en el protocolo de coherencia que

garanticen que las nuevas situaciones que surgen como consecuencia de la predicción son resueltas favorablemente.

Nuestro punto de partida fue el protocolo de coherencia implementado en RSIM y, en todo momento, tratamos de reducir al mínimo el número de cambios efectuados. En todo el desarrollo asumimos un modelo de consistencia secuencial como anteriormente justificamos.

Básicamente, la mayoría de los cambios realizados se concentran sobre el directorio, el cual es el encargado de recibir las respuestas a las órdenes de transferencia enviadas desde el nodo peticionario y, en caso de que la predicción sea errónea, de tomar las acciones correctivas necesarias para poder satisfacer el fallo. Además, se introducen una serie de extensiones en las cachés L2, como la inclusión de una tabla *NPT* para evitar predecir ciertos fallos *3-hop* que, de hacerlo, podrían conducir al sistema a un estado de bloqueo.

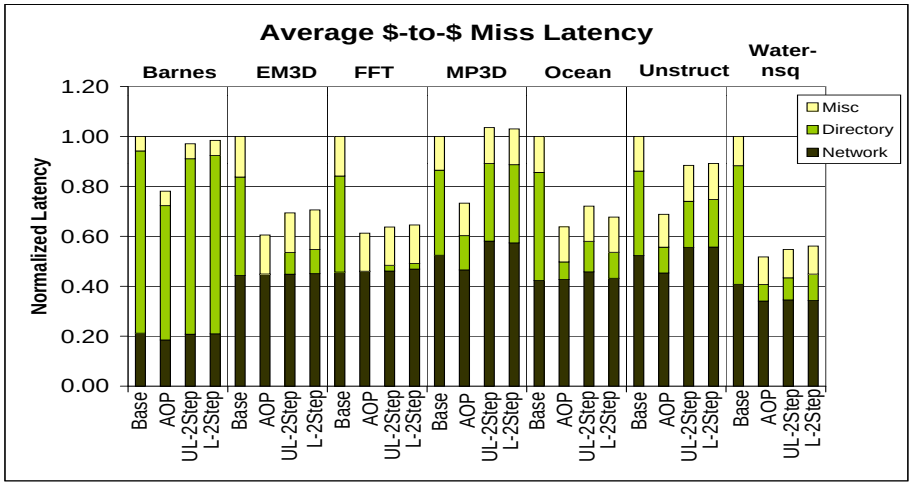


Figura 0.16: Latencia media normalizada para los fallos *3-hop*

Por último, en la Figura 0.16 mostramos las reducciones en la latencia media de los fallos *3-hop* que se obtienen al utilizar la técnica de predicción de propietario para aquellas aplicaciones en las que más del 16 % de los fallos de L2 son *3-hop*. En concreto, la figura muestra las latencias medias normalizadas para el sistema base (*Base*), para un sistema que hace uso de nuestra técnica pero que incluye un predictor casi perfecto y que, de esta forma, nos da una aproximación al potencial de la técnica (*AOP*) y, finalmente, para dos sistemas que hacen uso de dos configuraciones del predictor presentado anteriormente (*UL-2Step* y *L-2Step*). El predictor incluido en la configuración *UL-2Step* dispone de un número ilimitado de entradas en las tablas de predicción así como en la tabla *NPT*, con lo

que se consiguen eliminar las interferencias que surgen al tener tablas de predicción que no hacen uso de *tags*. Por otro lado, la configuración *L-2Step* constituye un ejemplo de un predictor realista, que puede implementarse utilizando menos de 64 KB. En todos los casos, las latencias medias aparecen divididas en las 3 categorías que identificamos con anterioridad: latencia de red, latencia de directorio y latencia miscelánea.

Podemos observar cómo la técnica presentada en esta tesis puede reducir la latencia media de los fallos *3-hop* para todas las aplicaciones salvo para BARNES y MP3D. Las reducciones que se consiguen van desde el 12 % observado para UNSTRUCTURED hasta el 45 % conseguido en WATER-NSQ. Es importante notar que estas reducciones pueden ser obtenidas con un predictor realista (menos de 64 KB).

0.6. Predicción de Compartidores para Acelerar los Fallos *Upgrade*

Los fallos *upgrade* se producen cada vez que una instrucción de almacenamiento encuentra una copia de sólo lectura de la línea de memoria correspondiente en la caché L2. En estos casos, el nodo peticionario posee una copia válida de la línea y necesita únicamente disponer de la propiedad de la misma. Para ello, el fallo se envía al directorio *home* al que la línea está asignada y éste, después de invalidar el resto de copias de la misma, otorga la propiedad al nodo peticionario. Todo este proceso se ilustra en la parte izquierda de la Figura 0.17.

De esta forma, al igual que sucedía para los fallos *3-hop*, el directorio no tiene que proporcionar una copia de la línea sino que ha de realizar únicamente el proceso de invalidación del resto de compartidores. Para esto se ha de obtener primero la entrada de directorio correspondiente con la finalidad de conocer la identidad de los compartidores y, después, enviar mensajes de invalidación a los mismos y recoger las respuestas que éstos devuelven. Es importante observar cómo a través de la arquitectura presentada en la Sección 0.4 se consigue reducir únicamente la latencia de la primera de las dos acciones anteriores (la determinación de los compartidores de la línea). Sin embargo, hemos podido comprobar que, en la mayoría de los casos, la segunda de las acciones consume la mayoría de los ciclos que se requieren para satisfacer un fallo *upgrade*.

Para incrementar el paralelismo entre las acciones que se realizan para resolver un fallo *upgrade*, presentamos en esta tesis la técnica denominada *predicción de compartidores* [5].

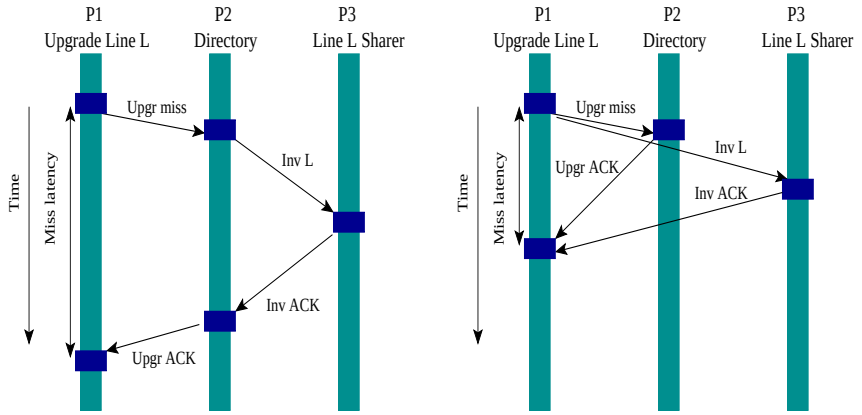


Figura 0.17: Proceso de resolución de un fallo *upgrade* en un multiprocesador cc-NUMA convencional (izquierda) y en uno que usa predicción de compartidores (derecha)

En la parte derecha de la Figura 0.17 resumimos cómo se resolvería un fallo *upgrade* mediante esta técnica. De forma parecida a como hicimos para los fallos *3-hop*, a través de la predicción de compartidores el nodo peticionario especularía acerca de la identidad de los compartidores cada vez que sucede un fallo *upgrade*, enviando directamente los mensajes de invalidación a los mismos. A la misma vez, el fallo se enviaría al directorio para llevar a cabo la comprobación de la predicción. Si el directorio encuentra que el nodo peticionario invalidó a todos los compartidores (que es el caso mostrado en la figura), entonces respondería inmediatamente otorgándole a dicho nodo la propiedad de la línea. En otro caso, cuando alguno o ninguno de los compartidores fueron incluidos en la predicción, el directorio llevaría a cabo la invalidación de los mismos antes de proporcionar la propiedad de la línea.

Como para el caso anterior, la técnica de predicción de compartidores se basa en el desarrollo de un predictor capaz de capturar el comportamiento repetitivo que, según hemos podido comprobar, presentan los fallos *upgrade*, así como la extensión del protocolo de coherencia para poder soportar el uso de la predicción.

0.6.1. Diseño del Predictor

Es importante notar que para decidir si un fallo es *upgrade* o no, el nodo peticionario no ha de hacer uso de ningún predictor. A diferencia de los fallos *3-hop*, que tenían que ser identificados mediante un predictor, el nodo peticionario sabe que un fallo es *upgrade* cuando encuentra una copia de sólo lectura de la línea de memoria para una instrucción de

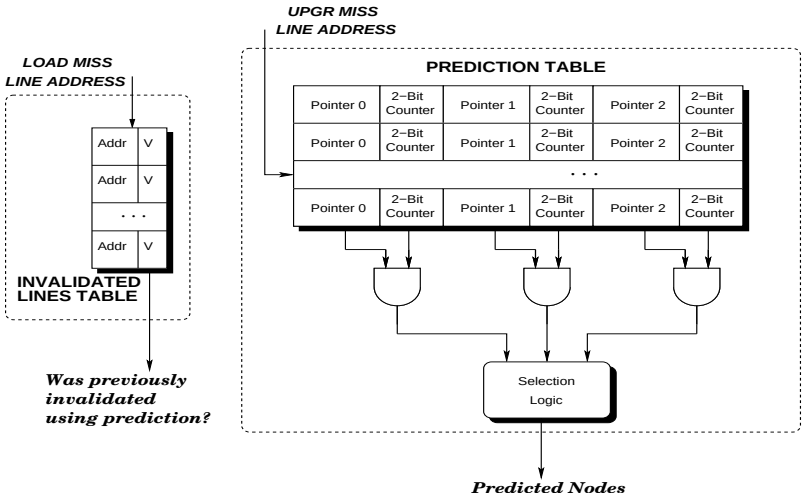


Figura 0.18: Anatomía del predictor propuesto para los fallos *upgrade*

almacenamiento[1].

De este modo, el predictor utilizado para los fallos *upgrade* no tiene que ser accedido para todos los fallos de L2, sólo para aquéllos que son *upgrade*, y se utiliza para determinar la identidad del resto de los compartidores de la línea de memoria.

La Figura 0.18 muestra la arquitectura del predictor que hemos desarrollado. Como en el caso del predictor de dos etapas que propusimos para los fallos *3-hop*, el predictor se implementa sin hacer uso de *tags* y se accede a él utilizando la dirección de la línea de memoria para la que el fallo tiene lugar. Cada una de las entradas del predictor consta de 3 punteros para almacenar la identidad de hasta 3 compartidores (cada uno de los campos *Pointer X*). Además, cada puntero tiene asociado un contador de confianza a través del cual se determina si se enviará un mensaje de invalidación o no al nodo identificado por el puntero (cada uno de los campos *2-Bit Counter*). Según hemos podido constatar, el disponer de 3 punteros por cada entrada resulta más que suficiente para la mayoría de los casos, ya que el número de mensajes de invalidación que se envían por cada fallo *upgrade* es bastante reducido. Este hecho ya había sido comprobado con anterioridad en varios estudios [23].

[1]Esto es cierto para la mayoría de los casos, aunque otro fallo *upgrade* distinto que estuviera siendo procesado por el directorio en ese instante podría hacer que el primero no fuera resuelto como un fallo *upgrade*.

0.6.2. Extensiones al Protocolo de Coherencia

Para poder hacer uso de la predicción es necesario introducir algunos cambios en el protocolo de coherencia original, que al igual que para el caso anterior fue el incluido en RSIM. El propósito de dichos cambios es el de resolver las nuevas situaciones que surgen como consecuencia de enviar los mensajes de invalidación directamente desde el nodo peticionario a los compartidores.

Además de enviar los mensajes de invalidación, el nodo peticionario está encargado de recoger todas las respuestas que se generan como consecuencia del uso de la predicción. En concreto, ha de recibir las respuestas a los mensajes de invalidación que los nodos que han sido predichos envían (*ACK* ó *NACK*), así como la respuesta del directorio que otorga la propiedad de la línea.

Como se muestra en la Figura 0.18, además del predictor, cada uno de los nodos del multiprocesador incluye una tabla *ILT*. Debido a que el protocolo de coherencia que RSIM proporciona no envía notificaciones cuando una determinada caché reemplaza una línea en estado compartido (*Shared*), necesitamos esta tabla para evitar que el directorio procese un fallo de lectura antes que el fallo *upgrade* que haciendo uso de la predicción lo provocó.

En algunos casos y para evitar incoherencias, algunos de los mensajes de invalidación tendrán que ser enviados dos veces por parte del nodo peticionario. La primera vez, cuando el predictor considera que un determinado nodo es uno de los compartidores y, la segunda vez, cuando se obtiene la respuesta del directorio y se observa que el mencionado nodo respondió con un *NACK* cuando el directorio consideró que debía de haber remitido un *ACK*.

Para finalizar, mostramos en la Figura 0.19 las reducciones en la latencia media de los fallos *upgrade* que se obtienen al emplear la técnica de predicción de compartidores para las mismas aplicaciones que fueron utilizadas en el caso anterior. En concreto, mostramos las latencias medias normalizadas para el sistema base (*Base*), para un sistema en el que el directorio realiza el proceso de invalidación sin requerir ciclo alguno y que, de esta forma, constituye una aproximación al potencial de la técnica (*POT*) y, finalmente, para dos sistemas que hacen uso de dos configuraciones del predictor presentado anteriormente (*UPT* y *LPT*). Con el fin de eliminar completamente las interferencias que surgen al no hacer uso de *tags*, el predictor incluido en la primera de las configuraciones (*UPT*) dispone de un número ilimitado de entradas en la tabla de predicción. Además de la tabla de predicción, las tablas *ILT* utilizadas en esta configuración disponen también de un número ilimitado de entradas. Por otro lado, la configuración *LPT* representa un ejemplo de un predictor realis-

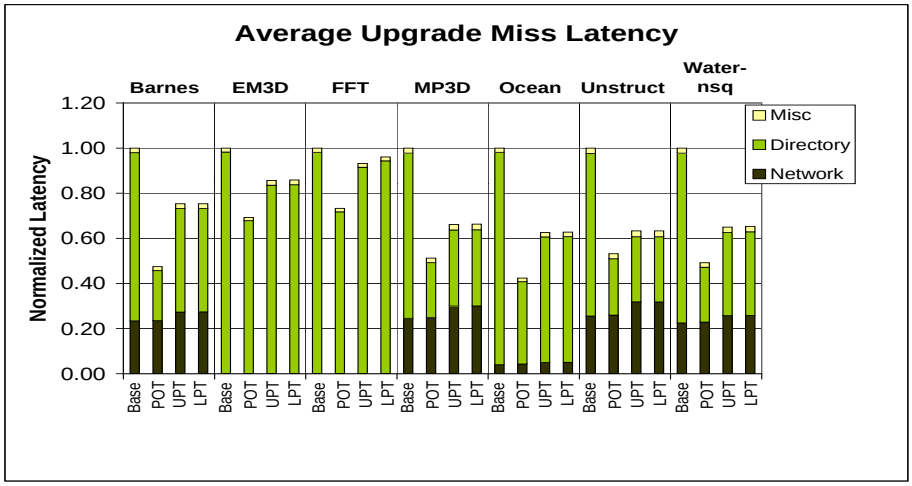


Figura 0.19: Latencia media normalizada para los fallos *upgrade*

ta, que puede implementarse utilizando menos de 48 KB. En todos los casos, las latencias medias aparecen divididas en las 3 categorías que identificamos con anterioridad: latencia de red, latencia de directorio y latencia miscelánea.

Podemos observar cómo la técnica presentada en esta tesis consigue reducir la latencia media de los fallos *upgrade* para todas las aplicaciones. Las reducciones que se consiguen son muy importantes para todas las aplicaciones (hasta un 37 % para OCEAN y UNSTRUCTURED) salvo para FFT (4 %). Finalmente, es preciso reseñar que estas reducciones han sido obtenidas con el predictor realista (menos de 48 KB).

0.7. Conclusiones y Vías Futuras

A pesar de que a través del uso de protocolos de coherencia de caché basados en directorios podemos llevar a cabo la realización de multiprocesadores de memoria compartida escalables y de alto rendimiento, más allá de los límites de los protocolos basados en *fisgoneo*, existen una serie de factores que limitan el número máximo de nodos que una máquina cc-NUMA puede ofrecer con una buena relación coste/redimiento.

Dos de estos factores son la sobrecarga de circuitería que supone el uso de los directorios, principalmente la cantidad de memoria necesaria para almacenar esta información, y las largas latencias de los fallos de L2 que caracterizan a los multiprocesadores cc-NUMA actuales. Las propuestas que hemos presentado en esta tesis han estado dirigidas a paliar estos dos problemas.

En primer lugar presentamos una nueva organización para el directorio, que hemos denominado *directorio de dos niveles*, a través de la cual se consigue reducir de forma significativa la sobrecarga de memoria introducida por la información de directorio sin que ello suponga una pérdida de rendimiento. Además, presentamos el concepto de *agrupamiento multicapa* y derivamos tres nuevos códigos de compartición comprimidos con menores requerimientos en cuanto a memoria se refiere que los propuestos con anterioridad.

A continuación añadimos a la arquitectura de directorio de dos niveles un tercer nivel que integramos, junto con una caché para datos compartidos, dentro de chip del procesador como una forma de acelerar significativamente los fallos de L2. Esta nueva *arquitectura de nodo* que derivamos a partir de una *clasificación de los fallos de L2* en función del trabajo realizado por el directorio para resolverlos, consigue obtener importantes reducciones en la componente de la latencia debida al directorio para todas las categorías de la clasificación.

Por último, hemos presentado también dos técnicas basadas en el uso de predictores que tratan de eliminar completamente el acceso al directorio de la ruta crítica de los fallos *3-hop* y *upgrade*. A través de la primera de las técnicas, que hemos denominado *predicción de propietario*, tratamos de enviar directamente la orden de transferencia requerida para un fallo *3-hop*, desde el propio nodo peticionario hasta el nodo propietario. La segunda de las técnicas, llamada *predicción de compartidores*, trata de hacer lo propio con los mensajes de invalidación que se envían en cada fallo *upgrade*. Ambas técnicas requieren el desarrollo de los predictores adecuados para cada uno de los casos, así como la introducción de una serie de cambios en el protocolo de coherencia que garanticen la correcta resolución de las nuevas situaciones que surgen al usar la predicción.

Como vías futuras del trabajo presentado en esta tesis se estudiará la posibilidad de combinar varias de las técnica propuestas para tratar de conseguir multiprocesadores de memoria compartida escalables que puedan competir con los multiprocesadores SMP incluso a mediana escala (de 64 a 256 nodos). Además, consideramos interesante el estudiar la combinación de directorios comprimidos con mensajes de coherencia multidestino (*multicast*) como modo de reducir, aún más, la sobrecarga de memoria debida a la información de directorio.

Otras líneas de investigación dentro del área en la que se enmarca la presente tesis y que están siendo objeto recientemente de importantes esfuerzos de investigación son la tolerancia de fallos (principalmente para multiprocesadores cc-NUMA [76, 83] y la reducción del consumo de energía (principalmente para multiprocesadores SMP [64]).

Chapter 1

Introduction

1.1. Motivation and Foundations

Most of the computer systems used nowadays have a single microprocessor as a central processor. Such systems, usually known as *uniprocessors*, satisfy the expectations of most users. Although microprocessor performance has been improving at a rate of about 50% per year, at any given time, the available technology limits the performance achievable by a single processor. Unfortunately, the computational power offered by a single microprocessor is not enough for some other applications. For example, online transaction processing (OLTP) systems must fulfill the requests that a large number of users are simultaneously issuing; weather forecasts require an increased computational power to obtain more accurate predictions; obtaining good-quality visual effects involve gigaFLOPS – or even teraFLOPS – of performance.

One approach to go beyond the performance offered by current microprocessors is to try building a faster processor using more gates in parallel and/or with a more exotic technology. This was the approach used for building the supercomputers of the 1970s and 1980s. The CRAY-1 [78], for example, had a 12.5 nanosecond clock (80 MHz) in 1976. Microprocessors, however, did not reach this clock rate until the early 1990s [41]. Unfortunately, this approach is no longer economically viable.

On the other hand, current microprocessors can integrate hundreds of millions of transistors on a single chip. Transistors are grouped together using short wires to form complex pipelines and integrated memory systems. This way, microprocessors constitute fast computing engines.

Creating a new microprocessor is not straightforward. This task involves a great num-

ber of professionals to first design the microprocessor and then carefully check its functionality, as well as an expensive fabrication technology to finally implement it. Therefore, a microprocessor's sales must reach millions of units to adequately amortize up-front costs [39] and it is certain that a customized microprocessor would not ship in enough volume to amortize the design and testing costs for complex pipelines and integrated memory system.

One can expect to surpass the computational power offered by a single microprocessor by employing a handful of them and connecting them in some way so that they can cooperate on common computational tasks. Today, this constitutes the only viable alternative for obtaining high-performance at a sustainable price and is the philosophy adopted by computer architects to create supercomputers with gigaFLOPS and teraFLOPS of performance. For example, the IBM BlueGene/L Supercomputer [86], scheduled to be operational in 2004-2005, will deliver target peak processing power of up to 360 teraFLOPS by having 65,536 PowerPC 440 processors.

As defined by Almansi and Gottlieb, a *parallel computer* is a "collection of processing elements that communicate and cooperate to solve large problems fast" [9]. This way, a multiprocessor can be seen as a "collection of communicating processors" [23]. Therefore, a multiprocessor is a computing system composed of multiple processors and memory modules connected together through an interconnection network.

Multiprocessors typically support either the message passing or the shared memory programming paradigms (sometimes both) [88]. In the traditional message passing model, processors communicate with each other by accessing the interconnection network directly to send and receive messages. This paradigm allows the programmer to optimize data movement among the nodes in the system and to use the characteristics of the network optimally (e.g., optimize for delivery speed or optimize for throughput). However, the message passing model forces programmers to explicitly specify any communication required among the nodes in the system. Such specification may be challenging to implement both efficiently and correctly, and may, ultimately, restrict the parallelization strategy.

In contrast, most programmers find that the shared memory model is an attractive programming paradigm for multiprocessors. This model provides programmers with an intuitively appealing programming model as well as affords reasonably efficient implementations. A programmer's intuition developed on single-processor systems still holds on shared memory multiprocessors, since a load from any node of a (shared) memory location still returns the value last written in that location. This model implicitly supports com-

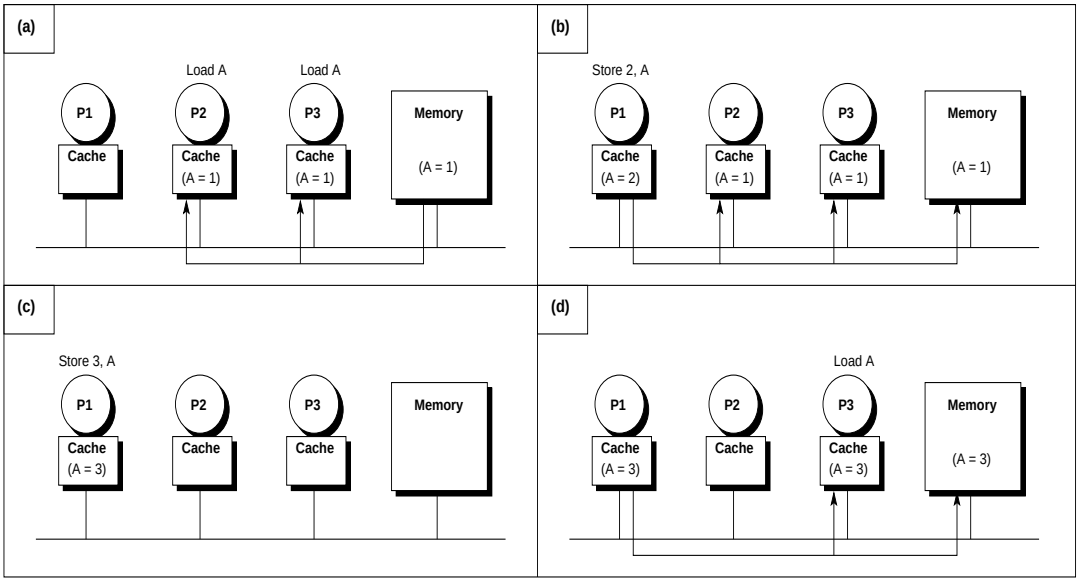


Figure 1.1: Multiprocessor cache coherence

munication among processors in the multiprocessor through conventional load and store instructions. The underlying memory system will satisfy a processor's request to read or write a memory location by sending and receiving messages to the appropriate memory module on the processor's behalf.

Just as in uniprocessor systems, caching of memory has become a key element in reducing memory latency in shared memory multiprocessor systems. In fact, caching offers even more benefit in multiprocessors since memory latencies are usually higher [57]. Unfortunately, multiprocessor caches are more complex because they introduce coherence problems between the independent processor caches.

Some shared memory multiprocessors avoid the cache coherence problem by restricting the kind of the addresses that a certain data cache can hold. For example, the Cray T3E multiprocessor [10] only allows local memory to be cached. Unfortunately, these restrictions usually result in lower processor utilization and force programmers to be much more cognizant of locality and interprocessor communication.

Figure 1.1 presents the two major coherence problems that caches introduce. First, when multiple processors read the same location they create *shared* copies of memory in their respective caches (see Figure 1.1(a)). By itself, this is not a problem. However, if the location is written, some action must be taken to ensure that the other processor caches do not supply stale data. These cached copies can either be updated or simply eliminated

through invalidations (Figure 1.1(b)). Whether writes should cause other cached copies to be updated or invalidated was the subject of considerable debate in the past. However, the increased bandwidth demand introduced by update-based protocols, the complexity of supporting updates, the trend towards larger cache blocks, and the "*pack rat*" phenomenon[1] with the important case of multiprogrammed sequential workloads, tipped the balance in favor of invalidation-based protocols, which are nowadays found in almost every commercial multiprocessor. After completing the write, the writing processor has an exclusive copy of the cache line (i.e., it holds the line *dirty* in its cache). This allows to subsequently write the line by only updating its cached copy, and further accesses to main memory or the other processor caches are not needed (Figure 1.1(c)). After a write invalidation, other processors that re-read the line get an updated copy from memory or the writing processor's cache. The misses caused by invalidations are referred to as coherence misses.

The second coherence problem arises when a processor holds a dirty item in its cache. When lines are dirty, simply reading a location may return a stale value from memory. To eliminate this problem, reads also require interaction with other processor caches. If a cache holds the requested line dirty, it must override memory's response with its copy of the cache line (see Figure 1.1(d)). If memory is updated when the dirty copy is provided, then both the dirty and requesting caches enter a shared state (as in Figure 1.1(d)). This state is equivalent to the initial state after multiple processors have read the line (Figure 1.1(a)). Thus, three cache states provide the basis for a simple multiprocessor cache coherence scheme: *Invalid* (after a write by another processor), *Shared* (after a read), and *Private* (or dirty, after a write by this processor) [23, 57].

Currently, most small-scale shared memory multiprocessors maintain cache coherence based on snoopy caches [31]. The structure of a typical snoopy-based shared memory multiprocessor is shown in Figure 1.2. In these systems, a shared bus is typically employed to interconnect all the processors and memory modules. The bus is located between the processor's private caches (or cache hierarchies) and the shared main memory subsystem. This way, symmetric access to all of the main memory from any processor is provided and these designs are often called *symmetric multiprocessors* or SMPs [23, 39, 41]. In these designs, cache coherence is achieved by broadcasting every memory reference to each processor cache and by every processor monitoring all requests to memory in the

[1] This phenomenon arises when the processors whose lines are being updated are never going to use them again [23]. In this case, the update traffic is useless and consumes interconnect and controller resources. Invalidations would clean out the old copies and eliminate the apparent sharing.

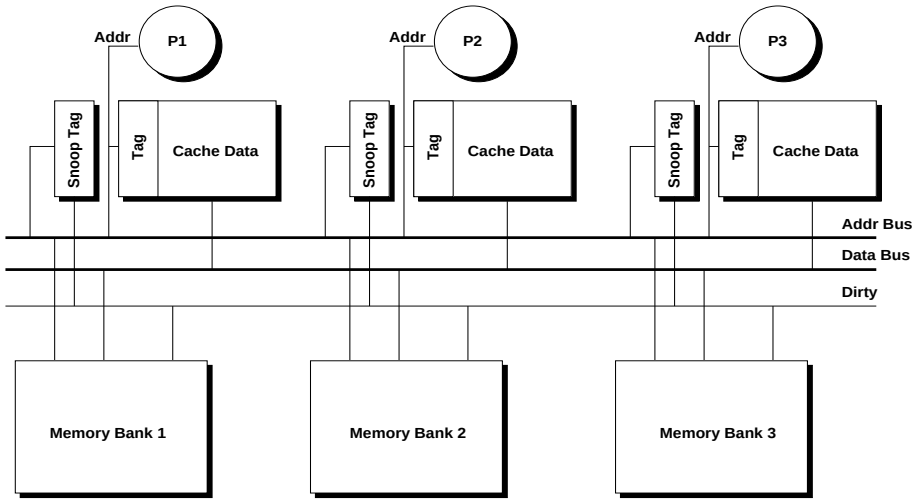


Figure 1.2: Architecture of a simple bus-based shared memory multiprocessor

order in which they appear on the bus. This monitoring, or *snooping*, allows each cache to independently determine whether accesses made by another processor require it to perform any action (i.e., update its caching state or provide the line). Figure 1.2 also shows that each processor maintains a duplicate set of snoop tags. The primary tags are used for local processor accesses while the snoop tags are dedicated to monitoring the bus [23]. The duplicate snoop tags filter memory traffic that does not affect this processor cache, thus reducing contention for the processor caches.

In the previous example, a snoopy-based system would work as follows: First, as processors 2 and 3 read location A from memory, all processors perform snoops, but since no one has a copy of the data, they do not interact with these reads. Second, processor 1 issues the store to A, which is translated into a read-exclusive bus operation to retrieve an exclusive copy of the line. When processors 2 and 3 see this read-exclusive matching their snoop tags, they invalidate their cached copies of A. Third, processor 1 performs several writes to A and, since it holds the block dirty, completes these operations locally in its cache. Finally, processor 3 re-reads the line. Processor 1 snoops the request and sees that it has the line dirty. Processor 1 flags this using the dirty signal on the bus, which disables memory and enables it to provide the memory line. Processor 1 then transitions to a shared state (along with processor 3, which reads the data). The memory controller also triggers a memory write in order to update main memory. This action is required because both processors are now in the *Shared* state and expect memory to supply an up-to-date copy of the line on the next read access.

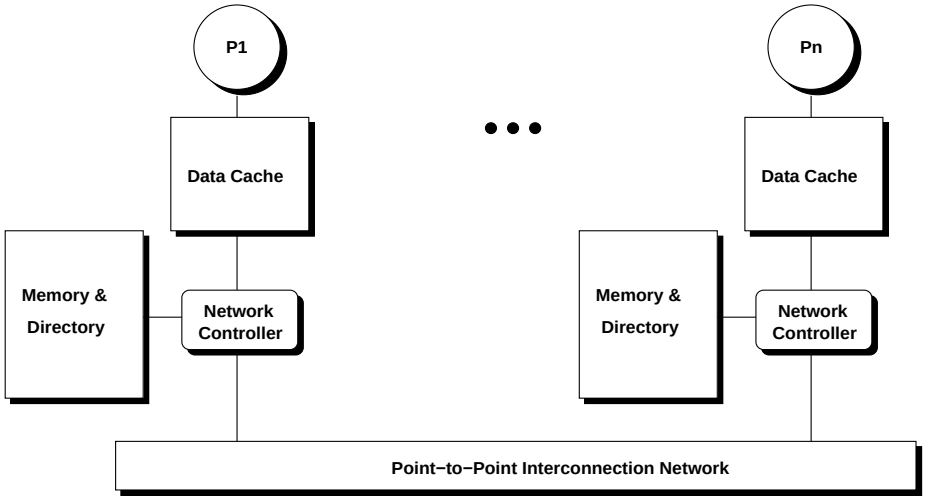


Figure 1.3: Architecture of a scalable cache-coherent shared memory multiprocessor

Snoopy coherence schemes are very popular. Dozens of commercial machines rely on variations of the scheme outlined above. One reason for the popularity of snoopy schemes is that they put most of the burden for cache coherence on the caches themselves. The only requirement on other portions of the memory system is that the processor must be able to override responses from memory in order to respond with dirty data. This property has allowed existing buses to be upgraded to support cache coherence, and for coherence protocols to be improved as new processors are introduced. Unfortunately, the broadcast nature of snoopy-based coherence protocols makes them unsuitable for larger systems. The bus used in a snoopy-based cache-coherent multiprocessor to broadcast coherence transactions (and to order them) has a fixed amount of bandwidth which usually limits it to less than 32 processors.

Some snoopy-based multiprocessors have recently extended the number of processors by replacing the bus with more sophisticated interconnection network organizations. For instance, the Sun UE10000 [18], which can accommodate up to 64 processors, separates address and data onto distinct wires. In turn, the address network, used to broadcast coherence transactions, uses four interleaved buses. Each of them is implemented as a pipelined broadcast tree. This complex address network, designed to perform ordered broadcasts, significantly increases the final cost of the system. Besides, the energy consumed by snoop requests, snoop bandwidth limitations, and the need to act upon all transactions at every processor, make snoopy-based designs extremely challenging, especially in light of aggressive processors with multiple outstanding memory requests.

On the other hand, scalable shared memory multiprocessors are constructed based on scalable point-to-point interconnection networks, such as a mesh or a torus [25], rather than the completely ordered networks required by snoopy-based cache-coherent multiprocessors. Besides, main memory is physically distributed in order to ensure that the bandwidth needed to access main memory scales with the number of processors. In these designs, accessing main memory has nonuniform access costs to a processor and in this way, architectures of this type are often called *cache-coherent, nonuniform memory access* or cc-NUMA architectures. Figure 1.3 illustrates the structure of a scalable shared memory multiprocessor. In these organizations, totally ordered message delivery becomes unfeasible and cache coherence is based on the concept of a *directory*. A state-of-the-art example is the SGI Origin 2000 [54] which can scale to several hundreds of processors.

Directory-based cache coherence [15] removes the scalability restrictions that snoopy-based protocols impose by eliminating the need for broadcast messages. Since the state of a memory line in the caches can no longer be determined implicitly by placing a request on a shared bus and having it snooped by the cache controllers, the idea is to maintain this state explicitly in a place, called a *directory*, where requests go and look it up. Instead of having a centralized directory, directory entries are distributed along with the memory, so that different directory accesses can go to different locations, just as different memory requests go to different memories. This prevents the directory from becoming the bottleneck. Each memory line is assigned to a directory (the *home directory*), which keeps a directory entry for the memory line. A natural way to organize directories is to maintain the directory information for each memory line together with the line in main memory at the home node for the line.

Each directory entry is comprised of two main fields: a state field, used to store the state of the line, and a *sharing code* [65] field, used to identify those caches currently holding a copy of the line. The majority of the bits of each directory entry are devoted to codifying the sharing code and, therefore, its election directly affects the memory required for the directory. A simple organization for the sharing code is as a bit-vector of N presence bits, which indicate for each of the N nodes[2] whether that node has a cached copy of the line or not (note that each node may be a uniprocessor or a multiprocessor system). This organization of the sharing code is called *Full-map*, Dir_N or *Bit-vector* [23] and some current multiprocessors, such as the SGI Origin 2000 [54] or the Stanford DASH Multiprocessor [56], directly use it or some variations on it.

[2]We use the terms *processor* and *node* interchangeably along this thesis.

In this configuration, all requests for a certain memory line are sent to the corresponding home directory. On receiving a request for a line, the directory controller accesses directory information to obtain the directory entry associated with the memory line. Depending on the state of the line and the type of the request (i.e., read-only or read-exclusive), the directory controller satisfies the request by performing some coherence actions before replying (for example, invalidation of the sharers for a read-exclusive request when the line is in the *Shared* state) or by directly providing the memory line.

For the example presented in Figure 1.1, a simple directory-based cache coherence protocol would operate as follows. First, as processors read a memory line (Figure 1.1(a)), their identities are registered through the sharing code of the corresponding directory entry and the state becomes shared. Upon a write to a line that the directory indicates is shared (Figure 1.1(b)), invalidation messages are sent to those processors caching the line (i.e., whose identifiers are indicated by the corresponding sharing code). The write also updates the state field in the directory to indicate that memory is not up-to-date, and the identity of the writing processor (*owner* processor) is stored in the sharing code field. As before, once a processor has an exclusive copy of a memory line, it can perform writes within its cache without interacting with the directory (Figure 1.1(c)). Upon a subsequent read by another processor, the directory forwards the request directly to the dirty processor. The dirty cache supplies the data to memory, which then passes it to the requesting processor (Figure 1.1(d)). This updates the sharing code and state for the line.

The main aim of using directory protocols is to allow cache coherence to scale beyond the number of processors that may be sustained by a bus. Although the use of directories allows multiprocessor designers to orchestrate shared memory multiprocessors with hundreds of nodes, the implementations of the shared memory paradigm have limited scalability, then becoming unfeasible for very large-scale systems, which use the message-passing paradigm. Examples of such machines are the ASCI Red, the ASCI Blue Pacific and the ASCI White multiprocessors.

There are several factors limiting the scalability of cc-NUMA designs. Two of the most important of these issues are, first, the cost in terms of the hardware overhead that the use of directories implies, and, second, the increased distance to memory, which is the reason for the higher L2 miss latencies that are currently being observed in cc-NUMA architectures.

The most important component of the hardware overhead is the amount of memory required to store the directory information, particularly the sharing code. Depending on

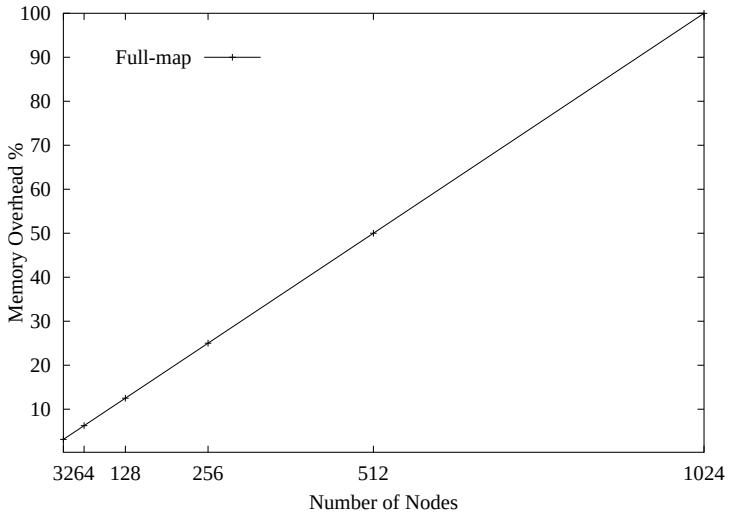


Figure 1.4: Memory overhead for full-map

how the sharing code is organized, memory overhead for large-scale configurations of a parallel machine could be intolerable. For example, for a simple full-map sharing code and for a 128-byte line size, Figure 1.4 draws directory memory overhead (measured as sharing code size divided by memory line size) as a function of the number of system nodes. As observed, directory memory overhead for a system with 128 nodes is 12.50%, which is not too bad. However, when the node count reaches 1024, this overhead becomes 100%, which is definitely prohibitive.

On the other hand, long miss latencies of directory protocols are caused by the inefficiencies that the distributed nature of the protocols and the underlying scalable network imply. Currently, and as a consequence of the increased distance to memory, the most important of such inefficiencies is the indirection introduced by the access to the directory, which is usually stored in main memory. This represents a unique feature of directory protocols, not present in SMPs. The consequences of such indirection are particularly serious for cache-to-cache transfer misses and upgrade misses, which constitute the most important fraction of the total L2 miss rate [2, 11]. Figure 1.5 presents the normalized average miss latency obtained when running several applications on the base system assumed in this thesis. Average miss latency is split into network latency, directory latency and miscellaneous latency (buses, cache accesses...). Further details about the evaluation methodology are given in Chapter 2. As can be observed, the most important fraction of the time needed to satisfy a certain request is spent at the directory in almost every application. Therefore, techniques for reducing the directory component of the miss latency will

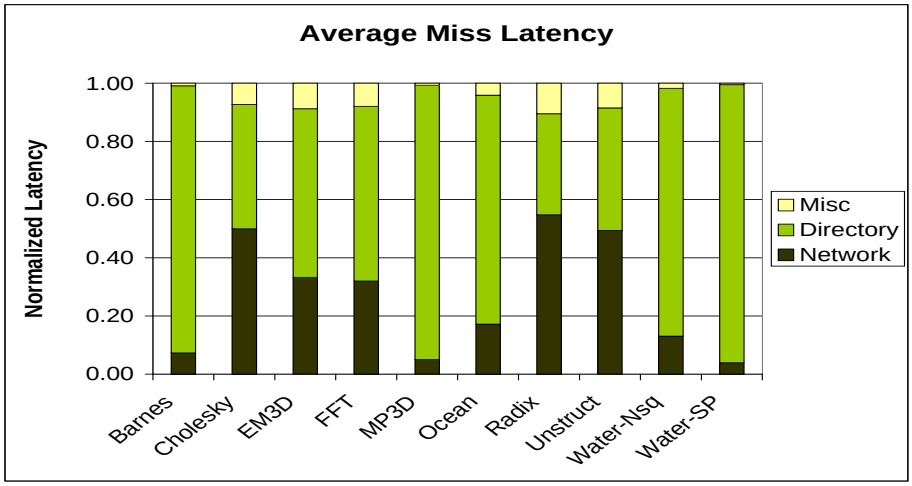


Figure 1.5: Average L2 miss latency

be rewarding.

1.2. Thesis Contributions

We can classify the primary contributions of this thesis into two main groups: techniques to reduce directory memory overhead and techniques to reduce L2 miss latency.

1.2.1. Techniques to Reduce Directory Memory Overhead

The main objective of these techniques is to reduce directory memory overhead without degrading performance. The contributions this thesis makes in this category are twofold:

- First, we introduce the ***multi-layer clustering concept*** and apply it to derive three new sharing codes with a better trade-off between directory memory overhead and performance than the existing ones.
- Second, we present ***multi-level directories***, a novel directory architecture significantly reducing directory memory overhead without hurting performance.

1.2.2. Techniques to Reduce L2 Miss Latency

Directory component has been observed as representing the most important fraction of the average L2 miss latency. This thesis proposes several techniques that accelerate

L2 misses by significantly reducing the time needed by directories to satisfy them. In particular, we present:

- ***A novel node architecture*** which exploits on-chip integration as the means of reducing the latency of each one of the L2 miss types found in cc-NUMA multiprocessors. We first propose a classification of the L2 misses in terms of the actions done by the directory to serve them. Then, we use this taxonomy to distill a new architecture especially designed to reduce the cycles spent in the directory while servicing L2 misses. This can be seen as a general technique, since every L2 miss type is accelerated.

- Two specific techniques that apply prediction to reduce the latency of the two most frequent L2 miss types:

  - ***Owner Prediction*** for accelerating cache-to-cache transfer misses.
  - ***Sharers Prediction*** for accelerating upgrade misses.

  Different from other proposals, the size of the predictor used in each one of the prediction-based techniques presented in this thesis is a key factor. Our objective is to provide prediction at a reasonable cost. In particular, we propose predictors with total sizes of less than 48 and 64 KB, for sharers prediction and owner prediction, respectively, which constitute a small percentage of the L2 cache size (512 KB, in our case). The access to the predictors do not add cycles, since their latency can be hidden from the critical.

1.3. Thesis Organization

The organization of the remainder of this thesis is as follows:

- **Chapter 2** describes the evaluation methodology employed throughout this thesis.

- **Chapter 3** faces the directory memory overhead problem. Multi-layer clustering is proposed as an effective approach to reducing the width of the directory entries, and it is applied to derive 3 new sharing codes. A detailed comparison in terms of memory overhead and performance between these new sharing codes and some others previously proposed is also included. Furthermore, this chapter presents a

two-level directory architecture and shows how it can significantly reduce memory overhead whilst maintaining performance.

- **Chapter 4** exploits current integration scale presenting a novel node architecture aimed at reducing both the long L2 miss latencies and the memory overhead of using directories that characterize cc-NUMA multiprocessors. It first presents a taxonomy of the L2 misses found in cc-NUMA multiprocessors and identifies directory component of the latency as representing a very important fraction of the average latency for each one of the categories of the taxonomy. The traditional directory is then replaced with a three-level directory architecture and a small shared data cache is also added to each one of the nodes of the multiprocessor. The small first-level directory and the shared data cache are integrated into the processor chip in every node, enhancing performance. The chapter includes a detailed evaluation study of the proposal.

- **Chapter 5** studies the use of prediction as a means of providing cc-NUMA multiprocessors with a more efficient support to cache-to-cache transfer misses. The proposal consists of both a prediction scheme (which can be implemented using less than 64 KB) and a coherence protocol properly extended to support the use of prediction. This chapter contains a detailed description and evaluation of the proposal.

- **Chapter 6** proposes the use of prediction for accelerating upgrade misses by invalidating sharers from the missing node instead of the directory node as is done in traditional directory-based coherence protocols. As in the previous case, the proposal comprises a prediction scheme (which can be implemented using less than 48 KB) and an extended coherence protocol, which are described and evaluated in this chapter.

- Finally, **Chapter 7** summarizes the main conclusions and suggests future directions to explore.

Chapters 3 to 6 also include descriptions of related works that can be found in the literature on each one of the topics.

Chapter 2

Evaluation Methodology

2.1. Introduction

This section presents the experimental methodology used for all the evaluations carried out in this thesis. Experiments have consisted on running several parallel programs on a simulation platform.

We have selected the Rice Simulator for ILP Multiprocessors Version 1.0 (**RSIM** for short), a detailed execution-drive simulator [43], as the simulation platform. We have configured RSIM to evaluate the impact that the proposals presented in this thesis have on the performance of a state-of-the-art cc-NUMA multiprocessor system.

Several shared memory applications are run on top of RSIM simulator for our performance studies. Originally, all the parallel programs employed in this thesis are written in C, extended with annotations that add parallel shared memory semantics to C. In all cases, the programs use the annotations proposed by Boyle *et al.* (usually known as PARMACS macros) [14] and should be properly transformed.

The rest of the chapter is structured as follows. Section 2.2 describes the RSIM simulation environment used for the performance evaluations carried out in this thesis. Descriptions of the applications employed to feed RSIM simulator appear in Section 2.3.

2.2. Rice Simulator for ILP Multiprocessors

RSIM is a publicly available execution-driven simulator primarily designed to study cc-NUMA multiprocessor architectures built from processors that aggressively exploit instruction-level parallelism (ILP). As shown in Figure 2.1, RSIM models state-of-the-

art ILP processors, an aggressive memory system as well as a multiprocessor coherence protocol and the network used to interconnect the multiprocessor nodes. RSIM simulator is written in a modular fashion using C++ and C to allow ease of extensibility and portability. Through a great number of configuration parameters, RSIM allows the simulation of a variety of shared memory multiprocessor and uniprocessor systems.

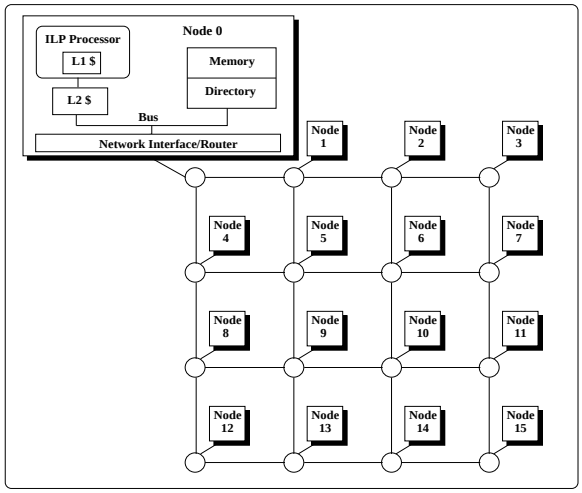


Figure 2.1: RSIM 16-node cc-NUMA multiprocessor system

From all the simulators publicly available, we selected RSIM because of its key advantage: it supports a processor model which aggressively exploits instruction-level parallelism (ILP) and is more representative of current and near-future processors.

Many shared memory multiprocessor simulation studies use simple processor models such as in-order issue, single-instruction issue per cycle, blocking-demand read misses and no speculative execution. To model the benefits obtained using ILP, researchers typically speed-up the simulated simple processor's clock rate and primary cache access time by a *clock multiplier* factor *N*, which can range from 1 to the issue width of the ILP processor modeled. Unfortunately, the appropriate value of *N* depends on both the application and the system, and currently no known technique can determine *N* a priori.

As demonstrated in [75], the use of these simple shared memory multiprocessor simulators can potentially produce large and unpredictable errors in applications that exploit hardware support for parallel read misses. As processors become more aggressive, modeling read miss parallelism becomes increasingly important [26].

A disadvantage of the increased accuracy and detail of RSIM is, however, that it is slower than simulators that do not model the processor.

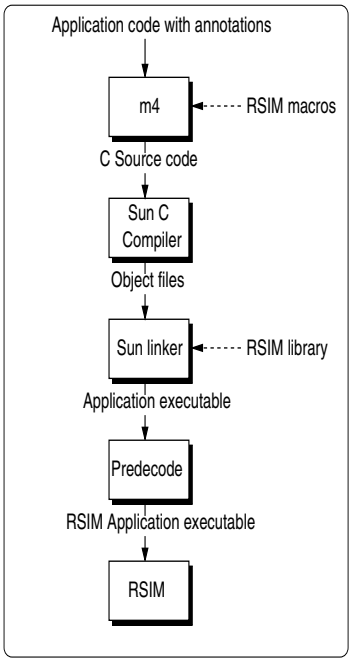


Figure 2.2: How parallel applications are transformed for simulation

RSIM simulates applications compiled and linked for SPARC V9/Solaris using ordinary SPARC compilers and linkers. Figure 2.2 illustrates the steps needed to transform a parallel application written in C (for example, one of the SPLASH-2 applications) into a convenient form ready to run onto the simulator on a target platform supporting ELF[1], such as the SGI Origin 2000 used for our simulations. First, the *m4* preprocessor transforms parallel programs into plain C sources substituting the parallel shared memory annotations into C statements or suitable library calls, using the set of PARMACS macros included in the RSIM distribution. Next, the Sun SPARC C compiler for Solaris is used to generate the relocatable object files that the Sun linker links together (along with the RSIM library, C library and math library), to create an executable SPARC application. For faster processing and portability, the RSIM simulator actually interprets applications in an expanded, loosely encoded instruction set format. In this way, a predecoder is finally used to convert the executable SPARC application into this internal format, which is then fed into the simulator.

The next three sections briefly describe the main parts of the RSIM simulator. We enumerate the most important configurable parameters available in each case as well as

[1]Users intending to run RSIM on target platforms that do not support ELF will need to firstly process the application executables to be simulated with the *unelf* utility provided by the distribution.

ILP Processor	
Processor Speed	1 GHz
Max. fetch/retire rate	4
Active List	64
Functional Units	2 integer arithmetic 2 floating point 2 address generation
Branch Predictor	2-bit history predictor
Counters in the Branch Predictor Buffer	512
Shadow Mappers	8
Memory queue size	32 entries

Table 2.1: Main processor parameters and their selected values

the particular values we have chosen for them for the performance evaluations carried out in this thesis. Refer to the RSIM Reference Manual [74] for a more detailed explanation.

2.2.1. Processor Microarchitecture

As mentioned previously, one of the most important characteristics of RSIM is that it models a processor microarchitecture that aggressively exploits ILP, incorporating features from a variety of current processors. However, the processor microarchitecture modelled by RSIM is closest to the MIPS R10000 processor [91]. Specifically, RSIM models the R10000's *active list* (which holds the currently active instructions and is also known as *reorder buffer*), *register map table* (which holds the mapping from the logical to physical registers) and *shadow mappers* (which allow single-cycle state recovery on a mispredicted branch). The pipeline parallels the *Fetch*, *Decode*, *Issue*, *Execute* and *Complete* stages of the dynamically scheduled R10000 pipeline. Instructions are graduated (i.e., *retired* or *committed*) from the active list after passing through this pipeline. Instructions are fetched, decoded and graduated in program order but they can issue, execute and complete out-of-order. In-order graduation enables precise interrupts.

The RSIM processor supports static branch prediction, dynamic branch prediction using either a 2-bit history scheme or a 2-bit agree predictor, and prediction of return instructions using a return address stack. Each hardware prediction scheme uses only a single level of prediction hardware. The processor may include multiple predicted branches at a time, as long as there is at least one shadow mapper for each outstanding branch. These branches may also be resolved out-of-order.

Most processor parameters are user-configurable, including the number of functional units, the latencies and repeat rates of the functional units, the instruction issue width, the size of the active list, the number of shadow mappers for branch speculation, and the size of the branch prediction structures. In this thesis, we model an aggressive version of the R10000, increasing the active list to 64 entries at a processor frequency of 1 GHz. Table 2.1 presents some of the processor parameters.

2.2.2. Memory Hierarchy

RSIM simulates a hardware cache-coherent distributed shared memory system (a cc-NUMA multiprocessor), with variations of a full-map (or bit-vector) invalidation-based directory coherence protocol. Each processing node consists of a processor, a two-level cache hierarchy (with a coalescing write buffer if the first-level cache is write-through), a portion of the distributed physical memory and its associated directory, and a network interface. A pipelined split-transaction bus connects the secondary cache, the memory and directory modules, and the network interface.

Both cache levels are lockup-free and store the state of outstanding requests using miss status holding registers (MSHRs). The first-level cache can either be a write-through cache with a non-allocate policy on writes, or a write-back cache with a write-allocate policy. RSIM allows for a multiported and pipelined first-level cache. Lines are replaced only on incoming replies. The size, line size, set associativity, cache latency, number of ports, and number of MSHRs can be varied. The coalescing write buffer is implemented as a buffer with cache-line-sized entries. All writes are buffered there and sent to the second-level cache as soon as this cache is free to accept a new request. The number of entries in the write buffer is configurable. The second-level cache is a pipelined write back cache with write-allocate. Lines are replaced only on incoming replies. The secondary cache maintains inclusion with respect to the first-level cache. The size, set associativity, cache latency, and number of MSHRs can be varied.

The memory is interleaved, with multiple modules available in each node. The memory is accessed in parallel with an interleaved directory, which originally implements a full-map based cache coherence protocol. The memory access time, the memory interleaving factor, the minimum directory access time, and the time to create coherence packets at the directory are all configurable parameters.

The directory can support either a four-state MESI protocol with *Modified*, *Exclusive*,

Memory Hierarchy Parameters	
Cache line size	64 bytes
L1 cache (on-chip, WT)	Direct mapped, 32KB
L1 request ports	2
L1 hit time	2 cycles
L2 cache (on-chip, WB)	4-way associative, 512KB
L2 request ports	1
L2 hit time	15 cycles, pipelined
Number of MSHRs	8 per cache
Memory access time	70 cycles (70 ns)
Memory interleaving	4-way
Cache-coherence protocol	MESI
Consistency model	Sequential consistency
Directory Cycle	10 cycles
First coherence message creation time	4 directory cycles
Next coherence messages creation time	2 directory cycles
Bus Speed	1 GHz
Bus width	8 bytes

Table 2.2: Main memory hierarchy parameters and their selected values

Shared and *Invalid* states (similar to the one implemented in the SGI Origin 2000 [54]), or a three-state MSI protocol. The RSIM directory protocol and cache controllers support cache-to-cache transfers. The protocols also include transient states at the directory and caches.

For local communication within a node, RSIM models a pipelined split-transaction bus connecting the L2 cache, the local memory, and the local network interface. The bus speed, bus width, and bus arbitration delay are all configurable.

The interface for memory in a shared memory multiprocessor is called the *memory consistency model* [6]. RSIM supports three memory consistency models configurable at compile-time: sequential consistency, processor consistency and release consistency.

Sequential consistency (SC) provides the most intuitive programming interface for shared memory multiprocessors by requiring that all memory operations appear to execute in program order and atomically [53]. On the other hand, relaxed consistency models have been shown to significantly outperform sequential consistency for single-issue, statically scheduled processors with blocking reads.

However, current microprocessors aggressively exploit instruction-level parallelism using methods such as multiple issue, dynamic scheduling and non-blocking reads. This allows both relaxed and SC implementations to be more aggressive and as recently estab-

Network Parameters	
Router speed	250 MHz
Channel width	32 bits
Channel speed	500 MHz
Number of channels	1
Flit size	8 bytes
Non-data message size	16 bytes
Router's internal bus width	64 bits
Arbitration delay	1 router cycle

Table 2.3: Network's main parameters and their selected values

lished by Hill [40], "*the future performance gap between the aggressively relaxed models and sequential consistency will not be sufficient to justify exposing the complexity of the aggressively relaxed models to the authors of low-level software*".

In our particular case, we have configured RSIM to simulate sequential consistency following the guidelines given by Hill.

Table 2.2 illustrates the base values for the most important parameters of the memory hierarchy.

2.2.3. Interconnection Network

For remote communication, RSIM currently supports a two-dimensional mesh network. In particular, RSIM models a wormhole-routed network with contention at the various switches.

Most modern networks are deadlock-free as long as the modules on the network continue to accept transactions. Within the network, this may require restrictions on permissible routes or other special precautions, see [25] for details. However, the network transactions used by request-response protocols (which are intrinsic to a shared address space), could still cause deadlock situations and RSIM provides two separate networks for requests and responses to avoid them.

The flit delay per network hop, the width of the network, the buffer size at each switch, and the length of each packet's control header are user-configurable parameters. In our simulations, we use a state-of-the-art network whose parameters are given in Table 2.3.

ILP Processor	
Processor Speed	1 GHz
Max. fetch/retire rate	4
Active List	64
Functional Units	2 integer arithmetic
	2 floating point
	2 address generation
Branch Predictor	2-bit history predictor
Counters in the Branch Predictor Buffer	512
Shadow Mappers	8
Memory queue size	32 entries
Memory Hierarchy Parameters	
Cache line size	64 bytes
L1 cache (on-chip, WT)	Direct mapped, 32KB
L1 request ports	2
L1 hit time	2 cycles
L2 cache (on-chip, WB)	4-way associative, 512KB
L2 request ports	1
L2 hit time	15 cycles, pipelined
Number of MSHRs	8 per cache
Memory access time	70 cycles (70 ns)
Memory interleaving	4-way
Cache-coherence protocol	MESI
Consistency model	Sequential consistency
Directory Cycle	10 cycles
First coherence message creation time	4 directory cycles
Next coherence messages creation time	2 directory cycles
Bus Speed	1 GHz
Bus width	8 bytes
Network Parameters	
Router speed	250 MHz
Channel width	32 bits
Channel speed	500 MHz
Number of channels	1
Flit size	8 bytes
Non-data message size	16 bytes
Router's internal bus width	64 bits
Arbitration delay	1 router cycle

Table 2.4: Simulation parameters

2.2.4. Summary of the Simulation Parameters

Table 2.4 summarizes the values given to the parameters in our simulations. These values are representative of current multiprocessors.

2.3. Benchmarks

Here, we describe the ten benchmark programs used to evaluate the optimizations proposed in this thesis. The selected benchmarks are presented in Table 2.5 and cover a variety of computation and communication patterns. BARNES-HUT, CHOLESKY, FFT, OCEAN, RADIX, WATER-SP and WATER-NSQ are from the SPLASH-2 benchmark suite [90]. EM3D is a shared memory implementation of the Split-C benchmark [22]. MP3D application is

drawn from the SPLASH suite [82]. Finally, UNSTRUCTURED is a computational fluid dynamics application [67].

All experimental results reported in this thesis correspond to the parallel phase of these applications. Data placement in our programs is either done explicitly by the programmer or by RSIM which uses a first-touch policy on a cache-line granularity. Thus, initial data-placement is quite effective in terms of reducing traffic in the system. Problem sizes have been chosen commensurate to the maximum number of processors used in each application.

Benchmark	**Input Size**
BARNES-HUT	8192 bodies, 4 time steps
CHOLESKY	tk15.O
EM3D	38400 nodes, degree 2, 15% remote and 25 time steps
FFT	256K complex doubles
MP3D	48000 nodes, 20 time steps
OCEAN	258x258 ocean
RADIX	2M keys, 1024 radix
UNSTRUCTURED	Mesh.2K, 5 time steps
WATER-NSQ	512 molecules, 4 time steps
WATER-SP	512 molecules, 4 time steps

Table 2.5: Benchmarks and input sizes used in this work

2.3.1. BARNES-HUT

BARNES-HUT application simulates the interaction of a system of bodies (galaxies or particles, for example) in three dimensions over a number of time steps, using the Barnes-Hut hierarchical *N*-body method. Each body is modelled as a point mass and exerts forces on all other bodies in the system. To speed up the interbody force calculations, groups of bodies that are sufficiently far away are abstracted as point masses. In order to facilitate this clustering, physical space is divided recursively, forming an octree. The tree representation of space has to be traversed once for each body and rebuilt after each time step to account for the movement of bodies.

The main data structure in BARNES-HUT is the tree itself, which is implemented as an array of bodies and an array of space cells that are linked together. Bodies are assigned to processors at the beginning of each time step in a partitioning phase. Each processor calculates the forces exerted on its own subset of bodies. The bodies are then moved

under the influence of those forces. Finally, the tree is regenerated for the next time step. There are several barriers for separating different phases of the computation and successive time steps. Some phases require exclusive access to tree cells and a set of distributed locks is used for this purpose. The communication patterns are dependent on the particle distribution and are quite irregular. No attempt is made at intelligent distribution of body data in main memory, since this is difficult at page granularity and not very important to performance.

2.3.2. CHOLESKY

The blocked sparse CHOLESKY factorization kernel performs a Cholesky factorization of a sparse matrix. It uses a dynamically scheduled version of the supernodal fan-out method. The matrix is divided into supernodes, each of which is a set of columns with identical nonzero structure in the factor. Supernodes are further divided into conveniently sized chunks of columns called *panels*. A panel receives updates from other panels to its left in the matrix. Once it has received all of its updates, it is placed on the task queue. A free processor pulls the panel off the queue and performs all the modifications done by this panel. These modifications in turn yield other ready panels that are placed on the task queue.

The principal data structure of CHOLESKY is the structure used to hold the matrix itself. Since the matrix is sparse, it is stored in a compressed manner. The primary operation that repeatedly needs to be performed, is the addition of a multiple of one column of the matrix to another column. Contention occurs for the task queue and for column modifications. Both of these cases are protected by locks.

2.3.3. EM3D

EM3D models the propagation of electromagnetic waves through objects in three dimensions. The problem is framed as a computation on a bipartite graph with directed edges from E nodes, representing electric fields, to H nodes, representing magnetic fields, and *vice versa*. At each step in the computation, new E values are first computed from the weighted sum of neighboring H nodes, and then new H values are computed from the weighted sum of neighboring E nodes. Edges and their weights are determined statically.

The initialization phase of EM3D builds the graph and does some precomputation to improve the performance of the main loop. To build the graph, each processor allocates a

set of E nodes and a set of H nodes. Edges are randomly generated using a user-specified percentage that determines how many edges point to remote graph nodes. The sharing patterns found in this application are static and repetitive.

2.3.4. FFT

The FFT kernel is a complex 1-D version of the radix-$\sqrt{n}$ six-step FFT algorithm, which is optimized to minimize interprocessor communication. The data set consists of the n complex data points to be transformed, and another n complex data points referred to as the roots of unity. Both sets of data are organized as $\sqrt{n} \times \sqrt{n}$ matrices partitioned so that every processor is assigned a contiguous set of rows which are allocated in its local memory. Synchronization in this application is accomplished by using barriers.

2.3.5. MP3D

MP3D is a Monte Carlo simulation of rarefied fluid flow. This benchmark simulates the hypersonic flow of particles at extremely low densities.

Succinctly, MP3D simulates the trajectories of particles through an active space and adjusts the velocities of the particles based on collisions with the boundaries (such as the wind tunnel walls) and other particles. After the system reaches steady-state, statistical analysis of the trajectory data produces an estimated flow field for the studied configuration. MP3D finds collision partners efficiently by representing the active space as an array of three-dimensional unit-sized cells. Only particles present in the same cell at the same time are eligible for collision consideration. If the application finds an eligible pair, it uses probabilistic test to decide whether a collision actually occurs.

MP3D allocates work to each process through a static assignment of the simulated particles. Each simulated step consists of a move phase and a collide phase for each particle that the process owns. The move phase computes the particle's new position based both on its current position and velocity, and its interaction with boundaries. The collision phase determines if the particle just moved collides with another particle, and if so, adjusts the velocities of both particles. Data sharing occurs during collisions and through accesses to the unit-sized space cells. During a collision, a process may have to update the position and velocity of a particle owned by another process. Also, each space cell maintains a count of the particle population currently present in that cell. Therefore, each time a process moves a particle, it may have to update the population count of some space cells if that

particle passes from one cell to another. These data accesses to particles and space cells may lead to race conditions that optional locks eliminate at some performance cost. In this thesis, MP3D is compiled with these locks. The presence of locks typically slow down the application execution but eliminates the data races and allows repeatability of the results.

2.3.6. OCEAN

The OCEAN application studies large-scale ocean movements based on eddy and boundary currents.

The algorithm simulates a cuboidal basin using discretized circulation model that takes into account wind stress from atmospheric effects and the friction with ocean floor and walls. The algorithm performs the simulation for many time steps until the eddies and mean ocean flow attain a mutual balance. The work performed every time step essentially involves setting up and solving a set of spatial partial differential equations. For this purpose, the algorithm discretizes the continuous functions by second-order finite-differencing, sets up the resulting difference equations on two-dimensional fixed-size grids representing horizontal cross-sections of the ocean basin, and solves these equations using a red-back Gauss-Seidel multigrid equation solver. Each task performs the computational steps on the section of the grids that it owns, regularly communicating with other processes.

2.3.7. RADIX

The RADIX program sorts a series of integers, called *keys*, using the popular radix sorting method. The algorithm is iterative, performing one iteration for each radix *r* digit of the keys. In each iteration, a processor passes over its assigned keys and generates a local histogram. The local histograms are then accumulated into a global histogram. Finally, each processor uses the global histogram to permute its keys into a new array for the next iteration. This permutation step requires all-to-all communication. The permutation is inherently a sender-determined one, so keys are communicated through writes rather than reads. Synchronization in this application is accomplished by using barriers.

2.3.8. UNSTRUCTURED

UNSTRUCTURED is a computational fluid dynamics application that uses an unstructured mesh to model a physical structure, such as an airplane wing or body. The mesh is

represented by nodes, edges that connect two nodes, and faces that connect three or four nodes. The mesh is static, so its connectivity does not change. The mesh is partitioned spatially among different processors using a recursive coordinate bisection partitioner. The computation contains a series of loops that iterate over nodes, edges and faces. Most communication occurs along the edges and faces of the mesh.

2.3.9. WATER-NSQ

WATER-NSQ performs an *N*-body molecular dynamics simulation of the forces and potentials in a system of water molecules. It is used to predict some of the physical properties of water in the liquid state.

Molecules are statically split among the processors and the main data structure in *water-nsq* is a large array of records that is used to store the state of each molecule. At each time step, the processors calculate the interaction of the atoms within each molecule and the interaction of the molecules with one another. For each molecule, the owning processor calculates the interactions with only half of the molecules ahead of it in the array. Since the forces between the molecules are symmetric, each pair-wise interaction between molecules is thus considered only once. The state associated with the molecules is then updated. Although some portions of the molecule state are modified at each interaction, others are only changed between time steps. There are also several variables holding global properties that are updated continuously.

2.3.10. WATER-SP

This application solves the same problem as WATER-NSQ, but uses a more efficient algorithm. It imposes a uniform 3-D grid of cells on the problem domain, and uses an O(n) algorithm which is more efficient than WATER-NSQ for large numbers of molecules. The advantage of the grid of cells is that processors which own a cell only need look at neighboring cells to find molecules that may be within the cutoff radius of molecules in the box it owns. The movement of molecules into and out of cells causes cell lists to be updated, resulting in communication.

2.3.11. Summary of Applications Used in Our Experiments

Table 2.6 summarizes the main characteristics of the applications used in our evaluations. For each application, it shows the problem size that has been selected, the maximum number of processors that can be used, the size of the resulting working set as well as the dominant sharing patterns that are found. As in [47], three main sharing patterns have been considered: migratory, producer-consumer and widely-sharing data.

Benchmark	Problem Size	Max Processors	Memory lines	Dominant Sharing Patterns
BARNES-HUT	8192 bodies	64	22.32 K	Wide sharing
CHOLESKY	tk15.O	64	249.74 K	Migratory
EM3D	38400 nodes	64	120.55 K	Producer-consumer
FFT	256K complex doubles	64	200.87 K	Producer-consumer
MP3D	48000 nodes	64	30.99 K	Migratory
OCEAN	258x258 ocean	32	248.48 K	Producer-consumer
RADIX	2M keys	64	276.30 K	Producer-consumer
UNSTRUCTURED	Mesh.2K	32	13.54 K	Producer-consumer and migratory
WATER-NSQ	512 molecules	64	71.81 K	Migratory
WATER-SP	512 molecules	64	5.85 K	Wide sharing

Table 2.6: Summary of the problem sizes, maximum number of processors that can be used, total number of memory lines that are referenced and dominant sharing patterns for the benchmarks used in this thesis

BARNES-HUT is an example of a program with little widely-shared data. The top of the octree is widely shared. However, it is difficulty to determine statically how many levels of the octree are widely shared [47]. Another application showing widely-shared data is WATER-SP.

On the other hand, migratory sharing constitutes the main sharing pattern found in WATER-NSQ. Also, the two arrays used in MP3D (the particle array and the space array) constitute migratory data, and the handling of the task queue used in CHOLESKY involves migratory sharing [47].

The dominant sharing patterns found in OCEAN and RADIX are producer-consumer [44, 47]. FFT is an intensive bandwidth application that also exhibits a producer-consumer sharing pattern. Finally, the nodes in the graph used in EM3D have a produce-consumer relationship [20].

UNSTRUCTURED is different from the rest of the applications because it has different dominant patterns for the same data structures in different phases of the application. The

same data structures oscillate between migratory and producer-consumer sharing patterns [66].

Chapter 3

Multi-layer Clustering and Two-level Directories

3.1. Introduction

This chapter deals with the directory memory overhead problem, the most important component of the hardware overhead that the use of directories implies and, therefore, one of the factors that limits the scalability of directory-based shared memory multiprocessors. There are two main alternatives for storing directory information [23]: *flat, memory-based* directory schemes and *flat, cache-based* directory schemes. In flat, memory-based directory schemes the home node maintains the identity of all the sharers and the state, for every one of its memory lines. On the contrary, flat, cache-based directory protocols (also known as chained directory protocols), such as the IEEE Standard Scalable Coherent Interface (SCI) [34], rely on distributing the sharing code among the nodes of the system. For every one of its memory lines, the home node contains only a pointer to the first sharer in the list plus a few state bits. The remaining nodes caching the line are joined together in a distributed, doubly linked list, using additional pointers that are associated with each cache line in a node (which are known as forward and backward pointers). The locations of the copies are therefore determined by traversing this list via network transactions.

The most important advantage of flat, cache-based directory protocols is their ability to significantly reduce directory memory overhead. In these protocols, every line in main memory only has a single head pointer. The number of forward and backward pointers is proportional to the number of cache lines in the machine, which is much smaller than the number of memory lines. Although several commercial machines were implemented using this kind of protocols, such as the Sequent NUMA-Q [59] and Convex Exemplar [21] multiprocessors, the tendency seems to have changed these days and from the SGI Origin

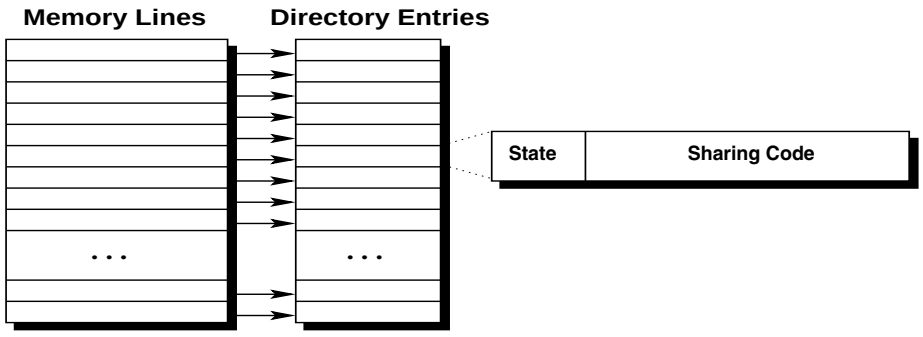


Figure 3.1: Directory organization for the family of coherence protocols assumed in this thesis

2000 [54] onwards, most newer designs use memory-based directory protocols, such as Piranha [12], the AlphaServer GS320 [28] or the Cenju-4 [42]. The decreased popularity of cache-based directory protocols is a consequence of some important drawbacks they introduce: the increased latency of coherence transactions as well as occupancy of cache controllers, and complex protocol implementations [23].

Therefore, our proposals are focussed on directory-based cache coherence protocols belonging to the first group. These protocols use the directory organization presented in Figure 3.1. For each memory line in main memory, a *directory entry* maintains all the necessary sharing information for the line. This sharing information comprises two main components[1]: the *state bits* used to codify one of the three possible states the directory can assign to the line (*Uncached*, *Shared* and *Private*), and the *sharing code*, that holds the list of current sharers. As a consequence of its simplicity, this directory organization has been extensively used in both the commercial and the research arena. Apart from those previously mentioned, examples of shared memory multiprocessors that have been constructed based on such a directory organization are the Stanford FLASH and DASH multiprocessors [50, 56], the MIT Alewife [7] and the HAL-S1 [89], among others.

As discussed in Chapter 1, most of the bits of each directory entry are devoted for codifying the sharing code. Therefore, its election directly affects the memory required for the directory and, consequently, the *directory memory overhead*. Directory memory overhead is typically measured as sharing code size divided by memory line size. The designer would like to keep the directory memory overhead as low as possible and would like it to scale very slowly with machine size [38].

Full-map sharing code (also known as *bit-vector* or Dir_N) constitutes the most popular

[1] Apart from other implementation-dependent bits.

way of keeping track of sharers in flat, memory-based directory schemes. The full-map sharing code [15] uses a presence bit vector to identify the exact sharers of a certain memory line. A presence bit is set if the associated node currently contains a copy of the line, and cleared otherwise. For an N-node multiprocessor, this sharing code requires N bits to keep a memory line coherent. Although it is efficient for small-scale machines, its scalability is very limited. For example, for a simple full-map sharing code and a 128-byte line size, the directory memory overhead for a system with 256 nodes is 25%, but when the node count reaches 1024, this overhead becomes 100%.

Compressed sharing codes have been proposed to reduce the width of directory entries in large-scale configurations. We refer to *compressed directory* as a directory structure in which a compressed sharing code is used. Compressed directories actually store the full directory information in a compressed way to use a fewer number of bits, introducing a loss of precision compared to *exact* ones, such as a full-map directory. This means that when this information is reconstructed, some of these nodes are real sharers and must receive the coherence message. However, some other nodes are not sharers actually, thus *unnecessary* coherence messages will be sent to them. Conversely, a full-map directory never sends unnecessary coherence messages and shows the best performance results.

As a consequence of the loss of precision, compressed sharing codes increase the execution time of the applications for two main reasons:

1. The overhead introduced by unnecessary coherence messages. These messages must be created and sent by the directory, and this requires additional clock cycles. Also, they must reach the destination nodes, consuming some of the resources of the system and increasing the contention on the network. At the destination nodes, some cycles are wasted processing these messages and there is an increased contention at the caches that do not actually have a copy of the line. Moreover, when an unnecessary coherence message arrives at its destination node, a *NACK* message must be returned to the home node (which requires processing time). Finally, the directory completes the coherence transaction when all replies (*ACK* from the actual sharers or *NACK* for the rest) are received. Thus, replies due to unnecessary messages may delay the completion of the coherence transaction.

2. More messages are needed to detect some race conditions. For example, if a read exclusive request arrives at a home node from the node holding the single copy of such line, a directory with precise information detects this particular situation and assumes

that there is a previous write-back message coming from this sharer that has been *surpassed* by the read exclusive one (race condition)[2]. It must wait for the write-back message to come in order to reply to the read exclusive message. However, when compressed sharing code schemes are used, there are situations where the identifier of the single node holding the line cannot be precisely codified. In this case, the read exclusive request is not delayed and coherence messages are sent to all the possible sharers (although there is a single sharer). When all these nodes have replied with a *NACK* message, the directory realizes the race condition, much later than in the previous case. A complete description of the extensions introduced in RSIM to allow the use of compressed directories is included in Appendix A.

An orthogonal way to diminish directory memory overhead is to reduce the total number of directory entries. That is, instead of having a directory entry for every memory line, one can construct a directory structure with less entries than the total number of memory lines and, for example, organize it as a cache [33, 73]. The main drawback of directory caches concerns how replacements are managed. Each time a directory entry is evicted, the directory must invalidate all shared copies of the associated memory line. Note that, these *unnecessary invalidations* may drastically increase the number of cache misses.

In this chapter we present *two-level directories*, a solution to the directory memory overhead problem which combines the advantages of both compressed sharing codes and directory caches and avoids their respective drawbacks.

The rest of the chapter is organized as follows. The related work is given in Section 3.2. In Section 3.3 we present several new compressed sharing codes based on the *multi-layer clustering* concept. Then, in Section 3.3.4, we analyze their impact on the final performance. Although compressed directories can significantly reduce directory memory overhead, they have negative consequences on the performance. In this way, Section 3.4 presents a *two-level directory architecture* especially conceived to reduce directory memory overhead without degrading performance. We study the performance of some two-level directory organizations in Section 3.4.3. Finally, Section 3.5 concludes the chapter.

[2]This is possible since the protocol implemented in RSIM does not assume point-to-point network order.

3.2. Related Work

Directory-based cache coherence protocols are accepted as the common technique in large-scale shared memory multiprocessors because they are more scalable than snooping protocols. Although directory protocols have been extensively studied in the past, directory memory overhead remains one of the major concerns on scalability.

As discussed previously, there are two main alternatives for storing directory information: *flat, memory-based* directory schemes and *flat, cache-based* directory schemes [23].

Flat, cache-based directory schemes rely on distributing the sharing code between the nodes that are currently sharing the memory line. Although some optimizations to the initial proposal have been studied (for example, in [17] and [70]) all these schemes introduce significant overhead, increasing drastically the latency of coherence transactions. We have not considered these organizations because they represent a different approach from the implementation point of view. For a comparison between flat, memory-based directory protocols and flat, cache-based ones refer to [38].

In flat, memory-based directory schemes the home node maintains the identity of all the sharers as well as the state, for each one of its memory lines. Memory overhead in these protocols is usually managed from two orthogonal points of view: reducing directory *width* and reducing directory *height*.

Some authors propose to reduce the width of directory entries by having a limited number of pointers per entry to keep track of sharers [8, 16, 81]. The differences between them are mainly found in the way they handle overflow situations, that is to say, when the number of copies of the line exceeds the number of available pointers [23]. As an example, $\mathrm{Dir}_i\mathrm{B}$ sharing code [8] provides i pointers to codify up to i potential sharers. It was inspired by experimental data suggesting that, in many cache invalidation patterns, the number of sharers is very low and a fewer number of pointers may be sufficient for the majority of cases [32]. When the number of available pointers i is exceeded, a broadcast bit in the directory entry is set. On a subsequent write operation, invalidation messages will be sent to all the nodes in the system, regardless of whether or not they are caching the line. Two interesting instances of this sharing code are $\mathrm{Dir}_1\mathrm{B}$ and $\mathrm{Dir}_0\mathrm{B}$. Whereas the former needs $1 + \log_2 N$ bits, the latter does not use any bit and always sends *N-1* coherence messages (invalidations or cache-to-cache transfer requests) when the home node cannot directly satisfy a certain L2 cache miss (i.e., on a coherence event), for a N-node system.

More recently, the segment directory has been proposed as an alternative to the limited

pointer schemes [19]. The segment directory is a hybrid of the full-map and limited pointers schemes. Each entry of a segment directory consists of two components: a segment vector and a segment pointer. The segment vector is a K-bit segment of a full-map vector whereas the segment pointer is a $log_2(N/K)$-bit field keeping the position of the segment vector within the full-map vector, aligned in K-bit boundary. Using directory's bits in this way results in a reduction of the number of directory overflows suffered by limited pointer schemes.

Others proposals reduce directory width by using *compressed sharing* codes. Unlike the well-known full-map sharing code, compressed sharing codes require a lower number of bits and achieve better scalability by storing an *in-excess* representation of the nodes that hold a line. This kind of sharing codes are also known as *multicast* protocols [65] and *limited broadcast* protocols [8].

Up till now, several compressed sharing code schemes have been proposed in the literature with a variety of sizes. Some of the most used compressed sharing codes are *coarse vector* [33], which is currently employed in the SGI Origin 2000 multiprocessor, *tristate* [8] and *gray-tristate* [65].

Unlike full-map sharing code, in coarse vector each bit of the sharing code stands for a group of K processors. The bit is set if any of the processors in the group (or some of them) cached the memory line. Thus, for a N-node system, the size of the sharing code is N/K bits. Assuming K=4, the total size of the coarse vector sharing code for a 16-node system is 4 bits. Bit 0 represents nodes 0, 1, 2 and 3, whereas bit 1 refers to nodes 4, 5, 6 and 7. Bits 2 and 3 stands for nodes from 8 to 15.

Tristate [8], also called the superset scheme by Gupta *et al.* [33], stores a word of d digits where each digit takes on one of three values: 0, 1 and *both*. If each digit in the word is either 0 or 1, then the word is the pointer to exactly one sharer. If any digit is coded *both*, then the word denotes sharers whose identifier may either be 0 or 1 in that digit, but match the rest of the word. If i digits are coded *both*, then 2^i sharers are codified. In this way, it is possible to construct a superset of current sharers. Each digit can be coded in 2 bits, thus requiring $2\log_2 N$ bits for a N-node system. Gray-tristate improves tristate in some cases by using Gray code to number the nodes.

Instead of decreasing directory width, other schemes try to reduce directory memory overhead by reducing directory height, that is the total number of directory entries available. This can be achieved either by combining several directory entries in a single entry (*directory entry combining*) [80] or by organizing the directory as a cache (*sparse direc-*

tory) [33, 73]. The first approach tends to increase the number of coherence messages per coherence event as well as the number of cache misses in those cases in which several memory lines share a directory entry. On the other hand, the use of sparse directories also increases the number of cache misses as a result of *unnecessary invalidations* that are sent each time a directory entry is evicted.

Finally, Everest [69] is an architecture for high performance cache coherence and message passing in partitionable distributed shared memory systems that use commodity SMPs as building blocks. In order to maintain cache coherence between shared caches included into every SMP, Everest uses a new directory design called Complete and Concise Remote (CCR) directory. In this design each directory maintains a *shadow* of the tags array of each remote shared cache. In this way, each directory consists of $N-1$ shadows for a N-node system, which prevents CCR directories from being a solution for scalable systems.

3.3. Multi-layer Clustering Concept

This section presents the first proposal of this thesis: several new compressed sharing code organizations based on the *multi-layer clustering* approach [1]. The goal of this approach is to improve the scalability of multiprocessors by reducing the size of the sharing code.

In this approach, nodes are recursively grouped into clusters of equal size until all nodes are grouped into a single cluster. Compression is achieved by specifying the smallest cluster containing all the sharers (instead of indicating *all* the sharers). Compression can be increased even more by indicating only the level of the cluster in the hierarchy. In this case, it is assumed that the cluster is the one containing the home node for the memory line. This approach is valid for any network topology.

Although clusters can be formed by grouping any integer number of clusters in the immediately lower layer of the hierarchy, we analyze the case of using a value equal to two. That is to say, each cluster contains two clusters from the immediately lower level. By doing so, we simplify binary representation and obtain better granularity to specify the set of sharers.

This recursive grouping into layer clusters leads to a logical binary tree with the nodes located at the leaves.

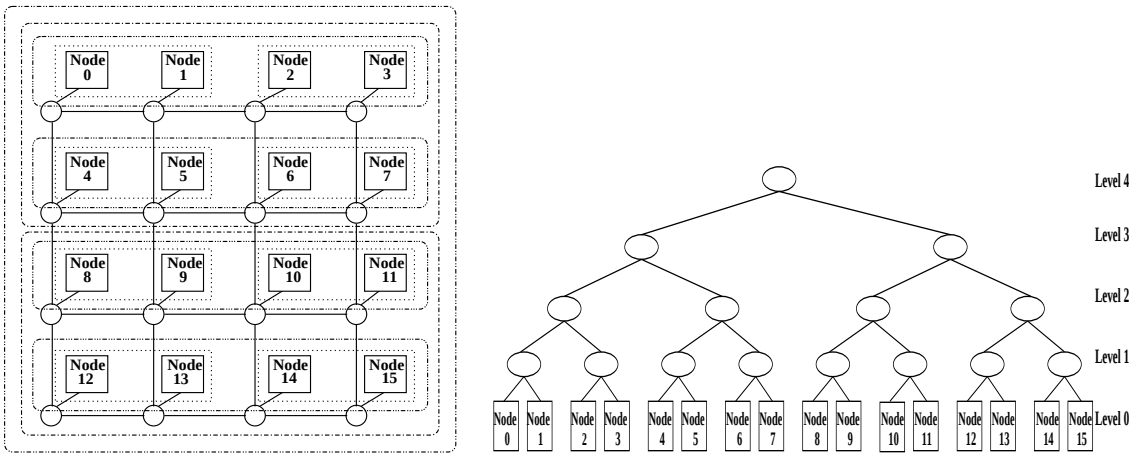
As an application of this approach, we propose three new compressed sharing codes. The new sharing codes can be shown graphically by considering the distinction between

the *logical* and the *physical* organizations. For example, we have a 16-node system with a mesh as the interconnection network, as shown in Figure 3.2(a), and we can imagine the same system as a binary tree (multi-layer system) with the nodes located at the leaves of this tree, as shown in Figure 3.2(b). Note that this tree only represents the grouping of nodes, not the interconnection between them. In this representation, each subtree is a cluster. Clusters are also shown in Figure 3.2(a) by using dotted lines. It can be observed that the binary tree is composed of 5 layers or levels ($\log_2 N + 1$, where N is a power of 2).

From this, the following three new sharing codes are derived: *Binary tree*, *Binary tree with symmetric nodes*, *Binary tree with subtrees*.

3.3.1. Binary Tree (*BT*)

Since nodes are located at the leaves of a tree, the set of nodes (sharers) holding a copy of a particular memory line can be expressed as the minimal subtree that includes the home node and all the sharers. This minimal subtree is codified using the level of its root (which can be expressed using just $\lceil \log_2 (\log_2 N + 1) \rceil$ bits). Intuitively, the set of sharers is obtained from the home node identifier by changing the value of some of its least significant bits to *don't care*. The number of modified bits is equal to the level of the above mentioned subtree. It constitutes a very compact sharing code (observe that, for a 128-node system, only 3 bits per directory entry are needed), but its precision may be low,



(a) *Physical* system (b) *Logical* system

Figure 3.2: Multi-layer clustering approach example

especially when few sharers, distant in the tree, are found (as is the normal case [23]). For example, consider a 16-node system such as the one shown in Figure 3.2(a), and assume that nodes 1, 4 and 5 hold a copy of a certain memory line whose home node is 0. In this case, node 0 would store 3 as the tree level value, which is the one covering all sharers (see Figure 3.2(b)).

3.3.2. Binary Tree with Symmetric Nodes (*BT-SN*)

We also introduce the concept of symmetric nodes of a particular home node. Assuming that 3 additional symmetric nodes are assigned to each home node, they are codified by different combinations of the two most-significant bits of the home node identifier (note that one of these combinations represents the home node itself). In other words, symmetric nodes only differ from corresponding home node in the two most significant bits. For instance, if 0 were the home node, its corresponding symmetric nodes would be 4, 8 and 12. Now, the process of choosing the minimal subtree that includes all the sharers is repeated for the symmetric nodes. Then, the minimum of these subtrees is chosen to represent the sharers. The intuitive idea is the same as before but, in this case, the two most significant bits of the home identifier are changed to the symmetric node used. Therefore, the size of the sharing code of a directory entry is the same as before plus the number of bits needed to codify the symmetric nodes (for 3 sym-nodes, 2 bits). In the previous example, nodes 4, 8 and 12 are the symmetric nodes of node 0. The tree level could now be computed from node 0 or from any of its symmetric nodes. In this way, the one which encodes the smallest number of nodes and includes nodes 1, 4 and 5 is selected. In this particular example, the tree level 3 must be used to cover all sharers, computed from node 0 or node 4.

3.3.3. Binary Tree with Subtrees (*BT-SuT*)

This scheme represents our most elaborated proposal. It solves the common case of a single sharer by directly encoding the identifier of that sharer. Thus, the sharing code size is, at least, $\log_2 N$ bits. When several nodes are caching the same memory line, an alternative representation is chosen. Instead of using a single subtree to include all sharers, two subtrees are employed. One of them is computed from the home node. For the other one, a symmetric node is employed. Using both subtrees, the whole set of sharers must be covered while minimizing the number of included nodes. Now, each directory entry has two fields of up to $\lceil \log_2 (\log_2 N) \rceil$ bits to codify these subtrees (depending on the size of

the subtree) and an additional field to represent the symmetric node used. An additional bit is needed to indicate the representation used (single sharer or subtrees). Note that, in order to optimize the number of bits required for this representation, we take into account the maximum size of the subtrees, which depends on the number of symmetric nodes used. Again, we assume 3 additional symmetric nodes for each home node. In the previous example, symmetric nodes do not change (i.e., nodes 4, 8 and 12). Node 0 should notice that the sharing code value implying fewer nodes is obtained by selecting node 4 as a symmetric node. Then, it encodes its tree level as 1 (covering node 1) and the tree level for the symmetric node as 1 (covering nodes 4 and 5).

In the example, Table 3.1 summarizes the nodes that would receive a coherence message with our proposed sharing codes. In addition, the same information is shown for full-map, Dir_0B, Dir_1B, coarse vector and gray-tristate sharing codes. The third column indicates the overhead with respect to full-map (computed as the relationship between the number of nodes encoded by the particular sharing code and the same number for full-map). As observed, the final number of sharers exceeds the total number of pointers for both Dir_0B and Dir_1B, resulting in the broadcast bit being set.

Scheme	**Nodes covered**	**Overhead**
Full-map	1,4,5	1
Dir_0B and Dir_1B	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15	5.34
Coarse vector	0,1,2,3,4,5,6,7	2.67
Gray-tristate	0,1,2,3,4,5,6,7	2.67
Binary tree	0,1,2,3,4,5,6,7	2.67
Binary tree with *SN*	0,1,2,3,4,5,6,7	2.67
Binary tree with subtrees	0,1,4,5	1.34

Table 3.1: Behavior of the evaluated sharing codes for the proposed example

Memory overhead of a sharing code scheme is computed as the amount of storage it requires divided by the storage used for the memory lines themselves. Table 3.2 shows the number of bits required for each sharing code (assuming 4 symmetric nodes), for both a 64-node multiprocessor and their general expression.

Figure 3.3 shows the percentage of memory overhead for each sharing code scheme as a function of the number of processors for a memory line of 128 bytes. As we mentioned above, full-map sharing code is characterized by its limited scalability. For instance, for

Sharing code	Size (in bits)	
	General	**N = 64**
Full-map	N	64
Dir_0B	0	0
Dir_1B	$1+\log_2 N$	7
Coarse vector	$\left\lceil \frac{N}{K} \right\rceil$	16
Gray-tristate	$2\log_2 N$	12
Binary tree	$\lceil \log_2 (\log_2 N+1) \rceil$	3
Binary tree with *SN*	$\lceil \log_2 (\log_2 N+1) \rceil + 2$	5
Binary tree with subtrees	$\max\{(1+\log_2 N), (1+2+2\lceil \log_2 (\log_2 N) \rceil)\}$	9

Table 3.2: Number of bits required by each one of the sharing codes

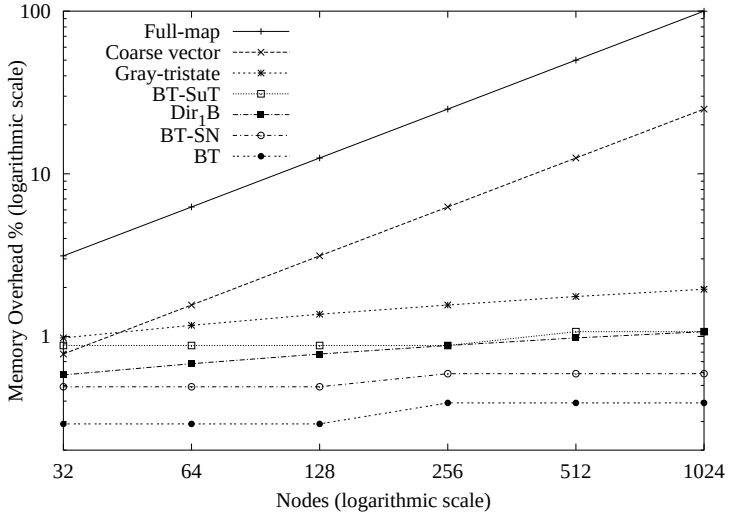


Figure 3.3: Memory overhead as a function of the number of nodes

a 1024-node configuration, 128 bytes would be required to keep a memory line coherent (100% of memory overhead). Coarse vector slightly reduces memory overhead, but it does not solve the scalability problem, since its sharing code is actually a linear function of *N*. The rest of the other schemes present a much better scalability since their sharing code size is not a linear function of the number of nodes but a logarithmic function. It is important to note that, for our most aggressive proposals (*binary tree (BT)* and *binary tree with symmetric nodes (BT-SN)*), memory overhead remains almost constant as the number of nodes increases. Finally, all three schemes proposed in this thesis achieve a much lower memory overhead than those proposed previously, such as gray-tristate.

3.3.4. Performance Evaluation

In Section 3.3, three novel compressed sharing codes have been presented: binary tree (*BT*), binary tree with symmetric nodes (*BT-SN*) and binary tree with subtrees (*BT-SuT*). In general, these schemes have been shown to obtain good scalability and the best results in terms of memory overhead. However, performance results in terms of number of unnecessary coherence messages and execution time are needed to evaluate the feasibility of such schemes. This section performs a performance evaluation of these sharing codes as well as other ones previously proposed (in particular, coarse vector, gray-tristate, Dir_1B and Dir_0B). For this, applications are executed using the maximum number of processors available, i.e. 64 processors for all applications except OCEAN and UNSTRUCTURED, which could be simulated using up to 32 processors. It was found that the more processors are used, the larger the impact the use of compressed sharing codes has on the application's performance.

Full-map sharing code provides the minimum execution time, since unnecessary coherence messages are completely eliminated. Table 3.3 gives the execution time (in processor cycles) for the applications evaluated when full-map sharing code is used (column two) as well as the total number of coherence events (column three), the number of coherence events per cycle (column four) and the average number of messages sent per coherence event (column five).

Application	Cycles $\times 10^6$	Coherence events $\times 10^3$	Events ($\times 10^3$) per Cycle ($\times 10^6$)	Messages per event
BARNES-HUT	33.99	204.02	6.00	2.15
CHOLESKY	42.85	213.98	4.99	1.11
EM3D	7.34	441.22	60.11	1.47
FFT	15.19	712.48	46.90	1.00
MP3D	71.22	1040.08	14.60	1.04
OCEAN	80.88	976.35	12.07	1.08
RADIX	12.72	337.46	26.53	1.04
UNSTRUCTURED	225.72	10532.46	46.66	1.10
WATER-NSQ	32.96	314.04	9.53	1.41
WATER-SP	29.24	55.92	1.91	7.42

Table 3.3: Execution times, number of coherence events, number of events per cycle and messages per event for the applications evaluated, when full-map sharing code is used

Column four of Table 3.3 provides an insight into the use of the directory information made by the applications. This depends on the L2 miss rates found in each case, and

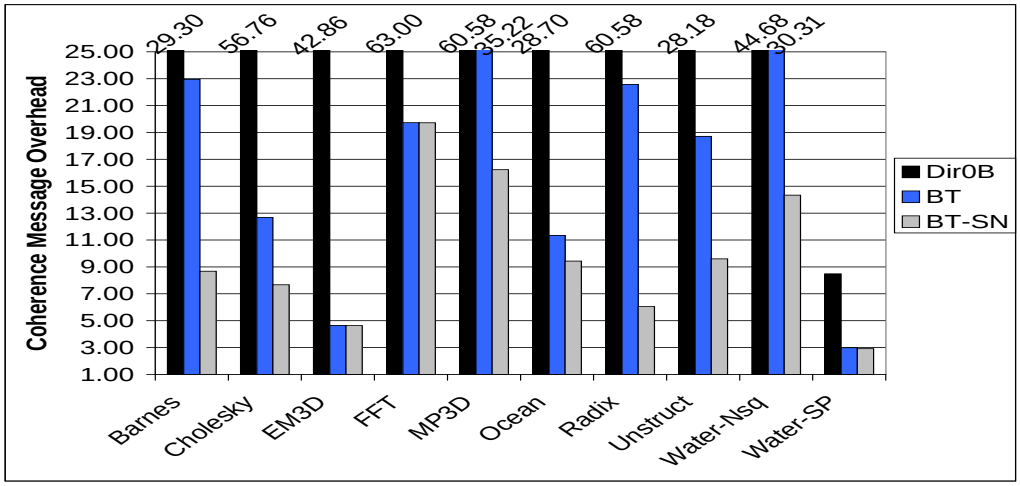


Figure 3.4: Normalized number of messages per coherence event for *BT-SN*, *BT* and Dir_0B sharing codes

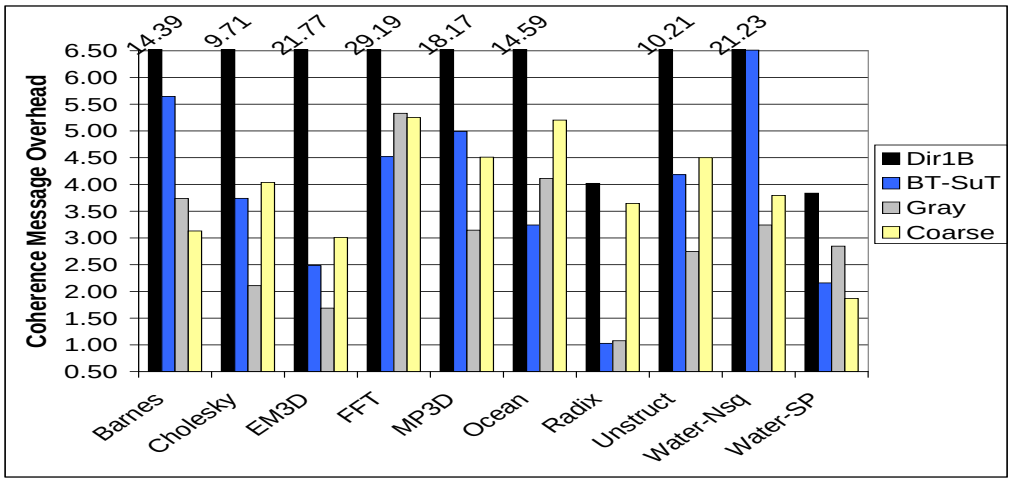


Figure 3.5: Normalized number of messages per coherence event for *coarse vector*, *gray-tristate*, *BT-SuT*, and Dir_1B sharing codes

we can find that whereas some applications frequently access directory information (for example, EM3D, FFT or UNSTRUCTURED) others, such as CHOLESKY and WATER-SP, make a lower utilization of this resource. This usage constitutes one of the parameters that influence the overhead on the execution time introduced by the use of compressed directories. Also, column five gives an approximate measure of the fraction of unnecessary coherence messages that will be sent using compressed directories.

Figures 3.4 and 3.5 show the average number of coherence messages sent per coherence

event for the sharing codes evaluated in this thesis. This number has been normalized with respect to the baseline configuration (using the full-map directory) and presented in Table 3.3 (column 5). Figure 3.5 shows the results obtained for those sharing codes requiring a greater number of bits and, therefore, expected to be more accurate. Whereas, Figure 3.4 presents the results gained when the less demanding sharing codes are employed.

As derived from Figure 3.4, when a small sharing code is employed, slight increases in the length of the sharing code translates into significant reductions on the number of unnecessary coherence messages in most cases. Completely removing the sharing code (Dir_0B scheme) significantly increases the number of messages that are sent on a coherence event (up to 63 times more messages are sent for FFT). Using a small sharing code as *BT* reduces this count, although much more coherence messages are still sent (35 times more for MP3D, 30 for WATER-NSQ and 23 for BARNES and RADIX). Adding symmetric nodes to BT (which supposes just 2 additional bits to the final size) obtains the best results, reducing the count to less than 15 times more messages than a full-map sharing code for all applications but FFT and MP3D.

On the other hand, for the most "memory consuming" sharing codes, there is no unique sharing code which can obtain the best results in all the cases. Whereas Dir_1B exhibits the poorest performance for all the applications, coarse vector obtains the best results for BARNES and WATER-SP, gray-tristate for CHOLESKY, EM3D, MP3D, UNSTRUCTURED and WATER-NSQ, and *BT-SuT* for FFT, OCEAN and RADIX. Now, having a greater number of bits for codifying the sharing code does not necessarily imply obtaining better results. For example, *BT-SuT* requires approximately half the number of bits used by gray-tristate and coarse vector.

Comparing the different sharing codes in terms of the number of messages that are sent per coherence event gives an idea about their accuracy. However, the most important metric is the impact on the application execution times. Figures 3.6 and 3.7 plot the execution times obtained for the evaluated sharing codes. These times have also been normalized with respect to the execution time obtained for the baseline configuration (which uses the full-map directory), so these graphs actually show the overhead introduced by the appearance of unnecessary coherence messages.

Although in most cases there is a correlation between the increment in the number of unnecessary coherence messages observed in Figures 3.4 and 3.5 and the performance degradation shown in Figures 3.6 and 3.7, there are some situations in which this is not true. In particular, for OCEAN and WATER-SP, the degradation observed in terms of ex-

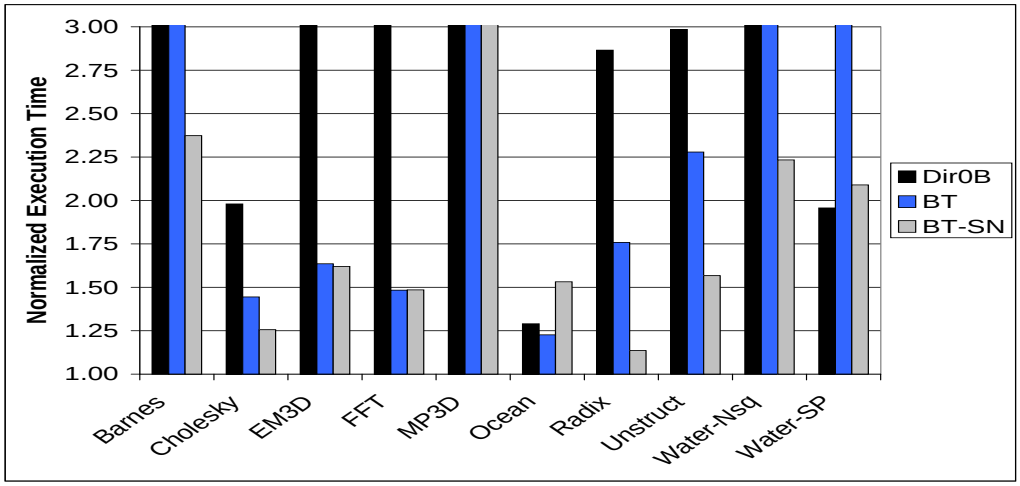


Figure 3.6: Normalized execution times for *BT-SN*, *BT* and *Dir$_0$B* sharing codes

ecution time when using Dir$_0$B is lower than that obtained when using *BT-SN*. However, Dir$_0$B was shown to send much more messages than *BT-SN* on average. These apparently incoherent results are a consequence of the excessive number of coherence messages that must be sent on every coherence event, most of which are indeed unnecessary coherence messages. This excessive number of coherence messages frequently exhausts the buffers in the system (network and directory buffers) and causes some of the requests that can not be served by the corresponding directory at a certain time, so they have to be retried. This indeterministic behavior is commonly found in systems highly loaded.

As expected, application performance is significantly degraded when Dir$_0$B, *BT* or *BT-SN* sharing codes are used (see Figure 3.6). Application execution times are increased to unacceptable levels when the Dir$_0$B scheme is applied. For most applications, slowdowns of more than 3 are suffered. This means that eliminating the sharing code does not constitute an effective solution to the scalability problem. As far as our schemes are concerned, *BT* obtains the worst results for all applications except the aforementioned (OCEAN and WATER-SP). We can see how the addition of such a small sharing code as *BT* significantly reduces the penalty suffered by Dir$_0$B for some applications as CHOLESKY, EM3D, FFT, RADIX and UNSTRUCTURED. For these applications, the average number of messages sent on a coherence event is much lower when *BT* sharing code is employed. On the other hand, for BARNES, MP3D and WATER-NSQ, *BT* sharing code was unable to reduce unnecessary coherence messages very much, and slowdowns of more than 3 were still suffered.

Adding symmetric nodes to *BT* (*BT-SN*) reduces the degradation observed for *BT* for

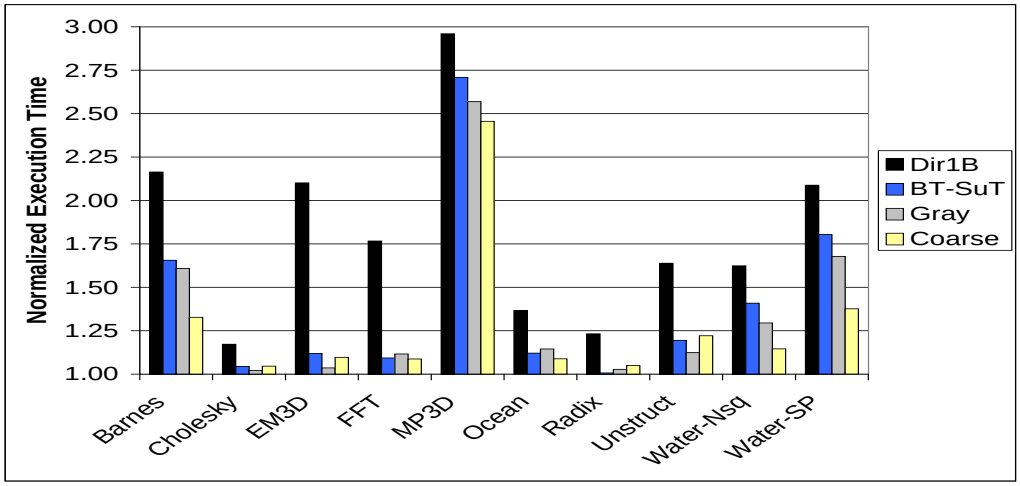


Figure 3.7: Normalized execution times for *coarse vector*, *gray-tristate*, *BT-SuT*, and Dir_1B sharing codes

some applications such as BARNES, CHOLESKY, RADIX, UNSTRUCTURED and WATER-NSQ. For other applications such as EM3D and FFT, both *BT-SN* and *BT* send approximately the same number of messages on every coherence event (see Figure 3.4) and similar degradations are observed for both sharing codes. Finally, MP3D performance is significantly degraded even when *BT-SN* is used (a slowdown of more than 3) and more elaborated sharing codes are needed.

Regarding the binary tree with subtrees (*BT-SuT*), Figure 3.7 shows how this sharing code significantly improves the results obtained by *BT-SN* for all the applications. The overhead introduced by *BT-SuT* scheme is very small for CHOLESKY (1.04), EM3D (1.12), FFT (1.09), OCEAN (1.12), RADIX (1.01), UNSTRUCTURED (1.19) and WATER-NSQ (1.39). However, the degradation introduced by this sharing code is still important for some other applications such as BARNES (1.66), WATER-SP (1.80) and, especially, for MP3D (2.71).

Comparing the directory schemes proposed in this work with previously compressed schemes (Dir_1B, coarse vector and gray-tristate), we can observe that *BT-SuT* outperforms Dir_1B for all the applications. Both sharing codes have approximately the same length, however *BT-SuT* makes more effective use of the bits. For CHOLESKY, EM3D, FFT, OCEAN, RADIX and UNSTRUCTURED, *BT-SuT*, coarse vector and gray-tristate obtain comparable numbers. For BARNES, MP3D, WATER-NSQ and WATER-SP, coarse vector obtains better results than *BT-SuT* and gray-tristate, although the differences are not very important. *BT-SuT* and gray-tristate sharing codes reach very similar performance numbers for

these applications. Remember that *BT-SuT* uses approximately half the storage required by gray-tristate and does not suffer the scalability problems found in coarse vector. Thus, we can conclude that *BT-SuT* achieves the best trade-off between memory overhead and performance degradation.

Finally, it is important to note that whereas the use of elaborated compressed directories (such as, *BT-SuT* or gray-tristate) can significantly reduce memory overhead with a small penalty on the final performance in most cases, there are some applications for which the use of a compressed directory significantly degrades the final performance (especially, for MP3D application). Therefore, the use of a compressed directory would not be a satisfactory solution for all cases and clever directory organizations are needed to significantly reducing directory memory overhead without damaging the final performance.

3.4. Two-level Directories: Reducing Memory Overhead and Keeping Performance

Although the use of compressed directories can significantly reduce the directory memory overhead, the performance degradation that they introduce limits its applicability. As discussed previously, this degradation is caused by the appearance of unnecessary coherence messages as a consequence of the loss of information that the compression of the sharing code entails.

As already mentioned, in addition to reducing directory entry *width*, an orthogonal way to diminish directory memory overhead is to reduce the total number of directory entries (the directory *height*) and organize it as a cache. The observation that motivates the utilization of fewer directory entries is the locality exhibited by memory references, that is, only a small fraction of the memory lines are used at a given time. Such a directory organization, which was called *sparse directory*, presents an important drawback: the *unnecessary invalidations* that are sent on replacements may drastically increase the number of cache misses.

In this thesis, we propose a solution to the scalability problem combining the benefits of both previous solutions. Two-level directories [1] (multi-level directories, in general) combine the properties of compressed sharing codes and sparse directories in order to increase the scalability of cc-NUMA multiprocessors without affecting performance. The idea is to have compressed directory information for all memory lines and, in addition, precise

directory information for only the subset of memory lines currently being referenced.

3.4.1. Two-Level Directory Architecture

In a two-level directory organization we distinguish two clearly decoupled structures:

1. *First-level directory (or uncompressed structure)*: Consists of a small set of directory entries, each one containing a precise sharing code (for instance, full-map or a limited set of pointers). In our particular case, we use full-map.

2. *Second-level directory (or compressed structure)*: In this level, a directory entry is assigned to each memory line. We use the compressed sharing codes proposed in this thesis (*BT*, *BT-SN* and *BT-SuT*) since their very low memory overhead makes them much more suitable for this level than any other previous scheme, thus achieving better scalability.

While the compressed structure has an entry for each memory line assigned to a particular node, the uncompressed structure has just a few entries, and is only used by a small subset of memory lines. Thus, for a certain memory line, *in-excess* information is always available in the second-level directory, but precise sharing code will be occasionally placed in the first-level directory depending on the temporal locality exhibited by this line. Note that in this organization if the hit rate of the first-level directory is high, the final performance of the system is close to the one obtained with a full-map directory. This hit ratio depends on several factors: size of uncompressed structure, replacement policy and temporal locality exhibited by the application.

Figure 3.8 shows the architecture of the proposed two-level directory. Both directory levels are located near main memory. Originally, state bits are only contained in the compressed structure. Tag information must also be stored in the uncompressed structure in order to determine whether there is a hit or not. Both directory levels are accessed in parallel and directory information is obtained from the directory cache, if it is present, or from the compressed second level otherwise. In the latter case, the compressed sharing code is previously converted into its full-map representation.

3.4.2. Implementation Issues

In sparse directories when an entry is replaced, invalidation messages are sent to all the nodes encoded in the evicted entry [23]. This affects the cache miss rate (and therefore

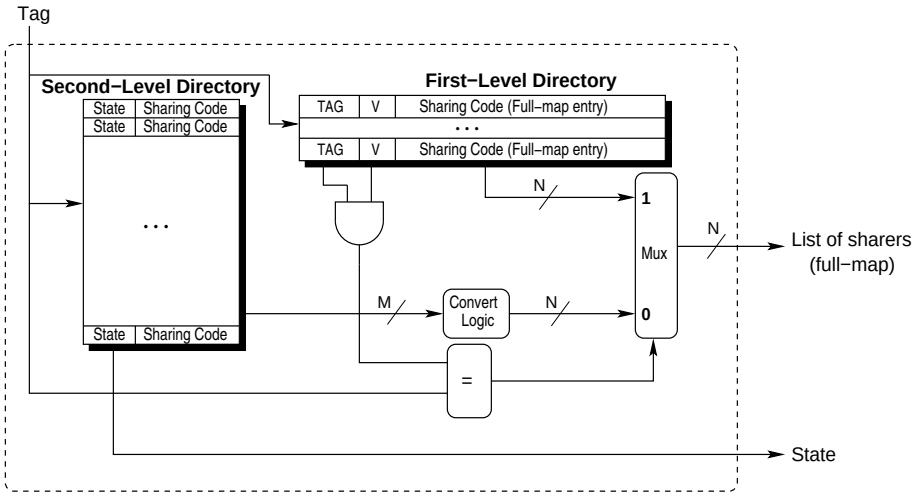


Figure 3.8: Two-level directory organization

the final performance) of processors having the remote copy of the line, since they receive the invalidation not because of a remote write, but because of a replacement in the remote directory cache. We refer to these invalidations as *premature invalidations*. These misses would not occur if a directory with correct information *per memory line* were used.

Our two-level directory organization will never send premature invalidations since correct information per memory line is always placed in the main (compressed) directory. For this organization, a miss in the first-level directory causes the second-level to supply the sharing information. In this case, the sharing code provides in-excess information and, maybe, unnecessary coherence messages will be sent to nodes that do not actually have a copy of the line. These messages will be replied with a negative acknowledgment (*NACK*). As opposed to premature invalidations, unnecessary coherence messages will never increase the cache miss rate with respect to a full-map directory implementation.

We assume that the organization of the first-level directory is fully associative, with a LRU replacement policy. Practical implementations can be set-associative, thus achieving similar performance at lower cost [63]. The entries of this structure use a full-map sharing code. The management of this uncompressed structure is carried out as follows:

a. When a request for a certain line arrives at the home node, an entry in the first-level directory could be allocated if the line is in *Uncached* state, or if an exclusive request was received. Note that, once an exclusive request has been completed, only one processor has a valid copy of the line. If the entry were allocated at some other time, only

in-excess information would be available, which would not exploit the features of this precise first level.

b. Since this uncompressed structure is quite small, capacity misses can degrade performance. In order to reduce such misses, an entry in the first-level directory is freed when a write-back message for a memory line in exclusive state is received. This means that the line is no longer cached in any of the system nodes, so its corresponding entry is available for other lines.

c. Replacements in the first-level directory are not allowed for the entries associated with those memory lines with pending coherence transactions.

Currently, entries in the first-level structure are not allocated as long as there is a single node holding a copy of the line and its identifier can be precisely encoded with the sharing code of the second-level directory (for example, when the binary-tree with subtrees is used as the sharing code of the second level).

3.4.3. Performance Evaluation

As expected, performance degradation occurs when compressed sharing codes are used. Such degradation depends on both the compressed sharing code used and the sharing patterns of the applications. This degradation could be negligible (when *BT-SuT* is used, the slowdown for RADIX is less than 1%) or significant (using *BT-SuT*, a slowdown of 2.71 is suffered for MP3D). In order to recover as much as possible from the lost performance, we organize the directory as a multi-level architecture. In this section, we evaluate the two-level directory organization previously proposed.

Figures 3.9 to 3.12 show the normalized execution times (with respect to a full-map directory) obtained with a two-level directory architecture, using several sharing codes for the second-level directory. We evaluate two different sizes for the full-map (FM) first-level directory: 512 and 1024 entries[3]. We have chosen these values according to the L2 cache size, 512*KB*. Having a first-level directory with 512 and 1024 full-map entries results in total sizes (excluding tags) of 4KB and 8KB, respectively, which constitutes less than 0.8% and 1.6% of the L2 cache size, respectively.

[3]Since the main memory is four-way interleaved, each memory module has a directory cache of 128 and 256 entries, respectively.

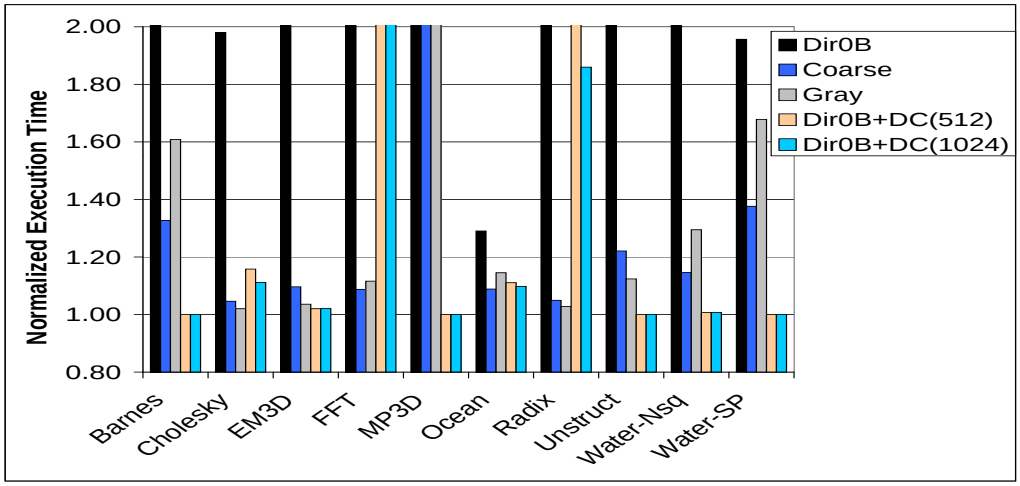


Figure 3.9: Normalized execution times for Dir_0B and first-level directory(*FM*)

First of all, we evaluate a directory architecture consisting of just a directory cache. For this, we set Dir_0B as the sharing code for the second-level directory. Figure 3.9 shows the normalized execution times obtained with a directory architecture combining both a directory cache of 512 and 1024 entries and a Dir_0B second-level directory (*Dir_0B+DC(512)* and *Dir_0B+DC(1024)*, respectively). For comparison purposes, results for Dir_0B, coarse vector and gray-tristate are also included. We can observe that, once a directory cache is used, the performance overhead introduced by a lack of sharing code almost disappears for BARNES, EM3D, MP3D, UNSTRUCTURED, WATER-NSQ and WATER-SP, especially when 1024 entries are used for the first-level directory. Due to the locality exhibited by these applications, directory references are concentrated on a small number of memory lines whose corresponding directory entries are frequently found in the first-level directory. On the contrary, for the rest of the applications having only a directory cache is not enough, since it still introduces important performance penalties.

Figure 3.10 presents the results obtained when the *BT* scheme is utilized for the second-level directory. As can be observed, the significant degradation introduced by such an aggressive compressed sharing code is almost hidden by the first-level directory, although 1024 entries are needed. These results are very promising, since the scalability of multiprocessors can be significantly enhanced using such a scalable compressed directory while performance is kept almost intact due to the presence of the first-level directory. Nevertheless, FFT, OCEAN, and RADIX still present some performance degradation, which may indicate that *BT* could be too aggressive for the second level.

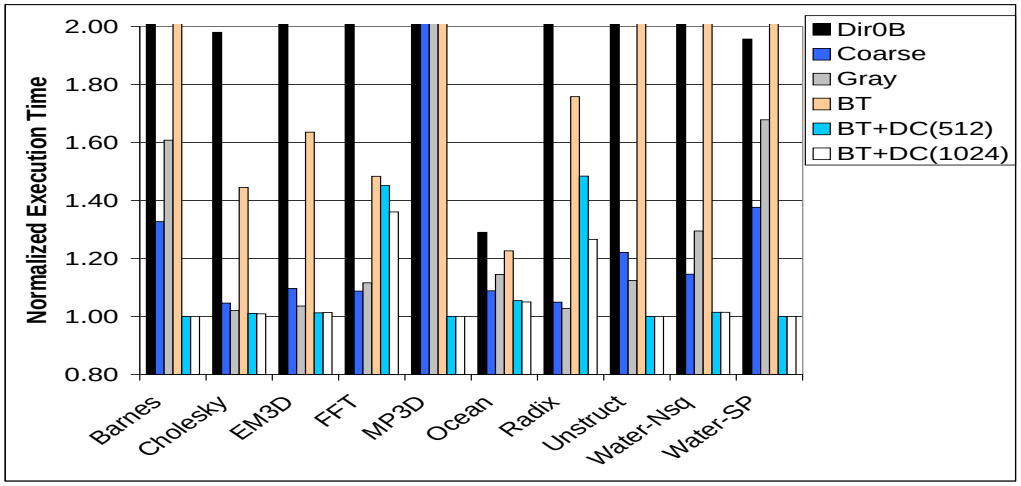


Figure 3.10: Normalized execution times for *BT* and first-level directory(*FM*)

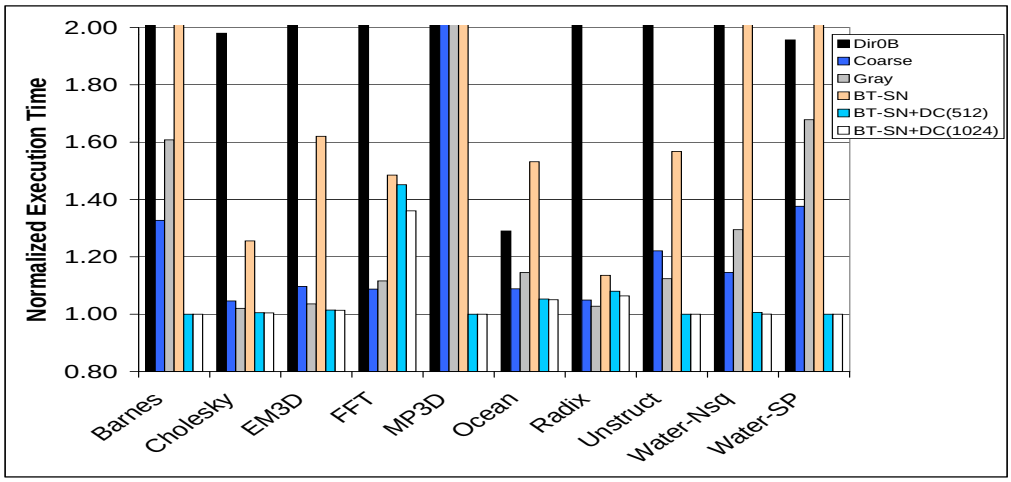


Figure 3.11: Normalized execution times for *BT-SN* and first-level directory(*FM*)

Figure 3.11 shows the performance of the two-level directory when the *BT-SN* sharing code is considered for the compressed structure. As shown in Figure 3.11, changing to *BT-SN* sharing code for the second-level directory has a minor impact on the final performance of FFT and OCEAN and the degradation numbers previously observed are also obtained. However, using BT-SN as the sharing code for the second-level directory in RADIX, significantly reduces the degradation previously reported, although 1024 entries in the first level are needed.

Finally, Figure 3.12 depicts the same results when *BT-SuT* is considered for the second-level directory. Observe that using just 512 entries in the first level (which constitutes less

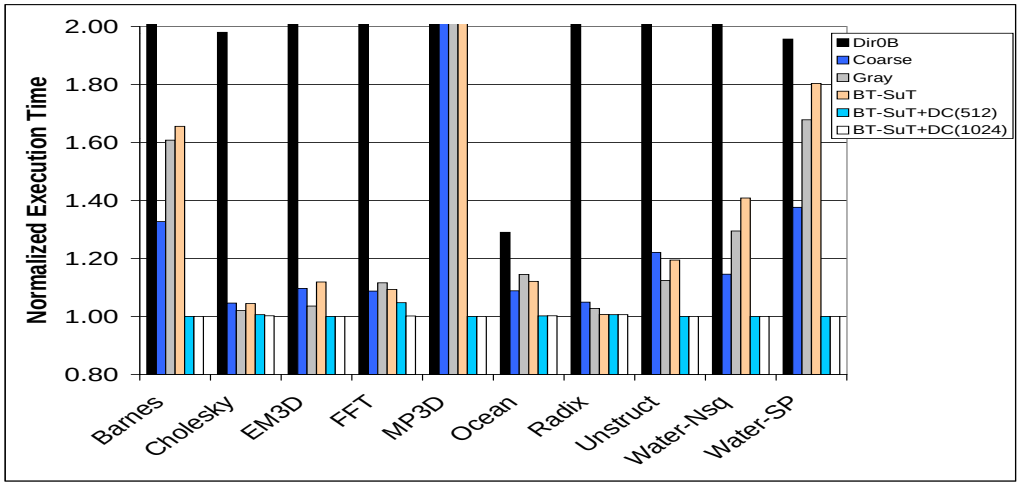


Figure 3.12: Normalized execution times for *BT-SuT* and first-level directory(*FM*)

than 0.8% of the L2 cache size) practically eliminates the penalty introduced by *BT-SuT* compressed sharing code, and the performance of a full-map directory is reached.

The results reported in this section clearly show that a two-level directory architecture composed of a very small full-map first-level directory and a compressed second-level directory constitutes an effective solution to the problem of directory scalability, causing negligible performance degradation. Our study compares our proposals to directory protocols that have very similar implementations. Specifically, we examined implementations that differ primarily in how the sharing code is encoded. We chose to exclude protocols that use different techniques, such as software-handled directories or distributed directories, because setting the plethora of implementation assumptions needed for these alternatives would have compromised the generality of our study.

3.5. Conclusions

The major objective of this chapter has been to overcome the scalability limitations that directory memory overhead imposes on current shared memory multiprocessors.

First, the multi-layer clustering concept is introduced and from it, three new compressed sharing codes are derived. Binary tree, binary tree with symmetric nodes and binary tree with subtrees are proposed as new compressed sharing codes with less memory requirements than existing ones. Compressed sharing codes reduce the directory entry *width* associated with a memory line, by having an *in-excess* representation of the nodes

holding a copy of this line. *Unnecessary* coherence messages degrading the performance of directory protocols appear as a result of this inaccurate way of keeping track of the sharers. A comparison between our three proposals and full-map sharing code is carried out in order to evaluate such a degradation. Also, a comparison with three of the most relevant existing compressed sharing codes, *coarse vector*, Dir_1B and *gray-tristate*, is presented. Results show that compressed directories slowdown the applications performance due to the presence of unnecessary coherence messages. Despite this degradation, our proposed scheme *BT-SuT* achieves a better trade-off between performance penalty and memory overhead ($\max\{(1+\log_2 N),(1+2+2\lceil\log_2(\log_2 N)\rceil)\}$ bits per entry) than previously proposed compressed sharing codes.

In order to alleviate the performance penalty introduced by compressed sharing codes, a novel directory architecture has also been proposed. *Two-level directory* architectures combine a very small uncompressed first-level structure (full-map directory) with a second-level compressed structure. Results for this directory organization show that a two-level directory combining both a small first-level directory and a *BT-SuT* second-level directory, can achieve the performance figures obtained by a system which uses a big and non-scalable full-map directory.

Therefore, we have shown that the directory architecture proposed in this chapter drastically reduces directory memory overhead while achieving similar performance. In this way, directory scalability is significantly improved when the directory is organized as a two-level structure.

Another remarkable property of the two-level directory architecture proposed in this chapter (in general, multi-level directories) is the possibility of implementing the small first-level directory using fast cache memories. In this way, directory accesses would be accelerated by avoiding the access to the slower main memory when directory information is found in the first level, consequently reducing the time needed to satisfy L2 misses.

Next chapter makes use of this property, exploring a new node architecture aimed at reducing the long L2 miss latencies found in cc-NUMA designs.

Chapter 4

A Novel Architecture to Reduce the Latency of L2 Misses

4.1. Introduction

The second factor limiting the scalability of cc-NUMA multiprocessors is the long L2 miss latencies suffered in these designs. Long miss latencies of directory protocols are caused by the inefficiencies that the distributed nature of the protocols and the underlying scalable network imply. One of such inefficiencies is the indirection introduced by the access to the directory. As pointed out in Chapter 1, the most important component of such indirection is the number of cycles that the home directory must employ to process the L2 miss (that is, the *directory component* of the L2 miss latency). This includes the time needed to access main memory in order to obtain directory information.

Unfortunately, a well-known industry trend is that microprocessor speed is increasing much faster than memory speed [36]. Both speed-growth curves are exponential, but they diverge. In this way, the increased distance to memory (the *memory wall* problem) raises even more the necessity of storing directory information out of main memory, in more efficient structures.

We can classify L2 misses according to the actions performed by directories in order to resolve them. This way, we can distinguish roughly two kind of L2 misses: those for which the corresponding home directory has to provide the memory line and those for which the memory line is not sent by the home directory. For those belonging to the latter category, the access to main memory in order to obtain directory information has a serious impact that significantly increases their latency when compared to symmetric multiproces-

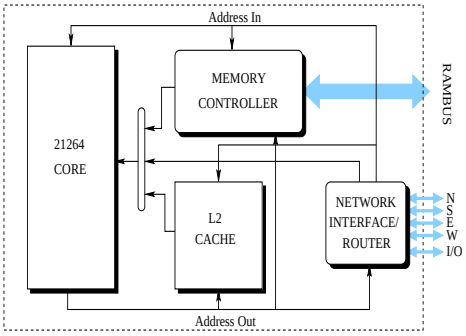


Figure 4.1: Alpha 21364 EV7 diagram

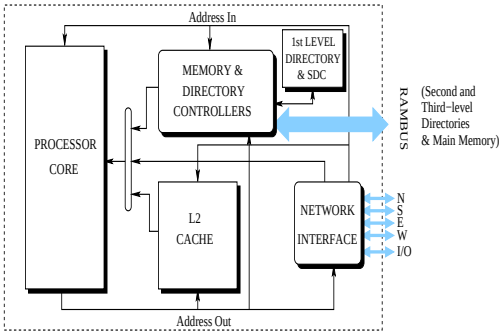


Figure 4.2: Proposed chip organization

sors (SMPs). Unfortunately, these L2 misses appear frequently and many efforts are being devoted to extend the scale at which snoopy coherence protocols can be applied. The Sun Enterprise E10000 [18] constitutes an example of a recent SMP which can accommodate up to 64 processors.

On the other hand, advances in VLSI have enabled a computing model where many processors will soon be integrated into a single die with little impact on system cost. Current technology improvements allow designers to integrate some key components of the system inside the processor chip. For example, the Compaq Alpha 21364 EV7 [35] includes on-chip memory controller, coherence hardware, and network interface and router (see Figure 4.1).

As we have demonstrated in Chapter 3, the directory could be organized as a two-level structure (a multi-level structure, in general) combining both a small first-level directory (which stores precise information for the most recently accessed memory lines) and a compressed second level (with compressed information for all the memory lines). This structure can significantly reduce directory memory overhead whilst keeping performance. In this chapter, we take advantage of the opportunities provided by current integration scale presenting a novel node organization especially designed to reduce the latency of L2 misses and the memory overhead of using directories that characterize cc-NUMA multiprocessors and limits their scalability [2, 4].

We extend the two-level directory architecture presented in Chapter 3 by adding another directory level which is included into the processor chip (along with the coherence controller). This results in a three-level directory architecture. The new directory level becomes the first-level directory, whereas the small full-map directory cache and the compressed directory are now the second and third-level directories, respectively. In addition, our proposal adds a small shared data cache to each one of the nodes forming the multi-

processor. The shared data cache is also included into the processor die and contains those memory lines for which the home node holds a valid copy in main memory, and that are expected to be requested in the near future. Finally, the second and third-level directories are placed outside the processor.

The on-chip integration of the small first-level directory and the shared data cache enhances performance. Now, those cache misses that find their corresponding directory entry and memory line (when needed) in the first-level directory and shared data cache, respectively, can be directly served from the processor chip, significantly reducing the directory component of the L2 miss latency (that is, the time needed by the home directory to satisfy L2 misses).

Additionally, and as previously demonstrated in Chapter 3, organizing the directory as a multilevel-level structure significantly reduces the memory overhead of using directories. In this case, this is achieved by having two directory levels out of the processor chip and using compressed data structures. The third-level directory is a complete directory structure (one entry per memory line) that uses a compressed sharing code to drastically reduce memory requirements, and the second-level directory is a small directory cache that tries to minimize the negative effects of having an imprecise third level as well as to quickly provide directory information when it is not present in the first level.

This chapter is organized as follows. Section 4.2 presents some related work. A taxonomy of the L2 misses found in cc-NUMA multiprocessors is presented in Section 4.3. The new node architecture is proposed and justified in Section 4.4. Finally, Section 4.5 shows a detailed performance evaluation of our novel proposal, and Section 4.6 concludes the chapter.

4.2. Related Work

In this chapter we study a novel node organization which takes advantage of: (1) on-chip integration as the means to reduce the usually long L2 miss latencies and (2) multi-level directories to significantly decrease the memory overhead of using directories.

Recent technology improvements have enlarged the number of components that can be included into a single chip. Some designers use the increased transistor counts and chip densities to exploit system-on-a-chip techniques. These techniques allow for integration of all system functions including compute processor, caches, communications processor, interconnection networks and coherence hardware onto a single die. Some previously

proposed designs exploiting system-on-a-chip techniques are the Compaq Alpha 21364 EV7 [35] and the IBM BlueGene/L supercomputer [86]. Other designs go further and use semiconductor technology trends to implement a complete multiprocessor into a single chip (multiprocessor-on-a-chip), for example the Compaq Piranha CMP [12] or the IBM Power4 [85]. Whereas the IBM BlueGene/L is a message-passing machine and cannot be compared with our proposals, the other two proposals are focused on the shared memory multiprocessing marketplace. In such designs, directory information is stored in main memory which puts the cycles needed to access main memory into the critical path of L2 cache misses.

Caching directory information was originally proposed in [33] and [73] as a means to reduce the memory overhead entailed by directories. More recently, directory caches have also been used to reduce directory access times [50, 63, 68]. The Everest architecture proposed in [69] uses directory caches to reduce directory access time. In addition, remote data caches (RDCs) have also been used in several designs (as [56], [59] and [72]) to accelerate access to remote data. A RDC is used to hold those lines that are fetched to the node from remote memory and acts as backup for the processor caches.

Some hardware optimizations, proposed to shorten the time processors loose because of cache misses and invalidations were evaluated in [84]. In [45], the remote memory access latency is reduced by placing caches in the crossbar switches of the interconnect to capture and store shared data as they flow from the memory module to the requesting processor. Subsequently, in [46] the same idea is applied to reduce the latency of cache-to-cache transfer misses. In this case, small directory caches are implemented in the crossbar switches of the interconnect medium to capture and store ownership information as the data flows from the memory module to the requesting processor. In both cases, special network topologies are needed to keep coherent the information stored in these switch caches.

Other proposals have focused on using snooping protocols with unordered networks. As an example, in [61], a technique that allows SMPs to utilize unordered networks is proposed, obtaining important improvements in terms of execution time when comparing with a cc-NUMA multiprocessor.

Multiprocessor-on-a-chip is emerging as an active research topic nowadays. The basic idea is to boost the scalability limits creating large-scale multiprocessors using chips integrating several processors as basic construction blocks. Examples of such architectures can be found in [12], [37] and [85]. Finally, Torrellas *et al.* [87] explore how a cache-coherent DSM machine built around Processor-In-Memory chips might be cost-effectively

organized.

4.3. Taxonomy of L2 Cache Misses

In the uniprocessor context, L2 cache misses are typically categorized into the *three Cs*: compulsory, capacity and conflict misses [39]. *Compulsory misses*, also known as *cold misses*, refer to those cache misses caused by the first access to a particular memory line by a processor. *Capacity misses* occur when all the lines that are referenced by a processor during the execution of a program do not fit in the cache (even with full associativity), so some lines are replaced and later accessed again. *Conflict misses* occur in caches with less than full associativity when the collection of lines referenced by a program that maps to a single cache set does not fit in the set. These misses would not have occurred in a fully associative cache. Cache-coherent multiprocessors introduce a fourth category of misses: *coherence misses*. These are the result of inter-process communication when memory lines are shared among multiple caches and are subsequently classified into *true* and *false* sharing misses [23].

In this section, we propose a novel taxonomy of the L2 cache misses found in cc-NUMA multiprocessors in terms of the actions performed by the directory in order to satisfy them [2]. Since the proposals presented in this thesis are focused on optimizing directory-based shared memory multiprocessor systems, we find our classification very suitable for our purposes and will be extensively used from now on.

Each time an access to a certain memory line from a particular processor (i.e., a load or a store) misses in the L2 cache, a request for the line is addressed to the corresponding *home* directory. The home directory accesses directory information in order to obtain both the state of the line and a list of those nodes (if any) that hold a copy of the memory line. Depending on both the state of the memory line and the type of access causing the miss, the directory determines the actions needed to satisfy the miss.

Directories receive misses that are caused by load and store instructions (load and store misses) and see memory lines as being in one of the following three states (MESI states are used for L2 caches) :

- *Uncached*: The memory line is not cached by any node and the home directory has a valid copy of the line.

- *Shared*: Several L2 caches are holding read-only copies of the memory line, and the home directory has a valid copy of the line.
- *Private*: There is a single L2 cache having a read-write copy of the memory line. Now, the copy stored by the home directory is not guaranteed to be valid and the copy hold by the owner node should be obtained.

We classify L2 cache misses into four categories in terms of the actions that directories perform to satisfy them: *$-to-$* misses, *Inv* misses, *Mem* misses and *Inv+Mem* misses.

4.3.1. *$-to-$* Misses

Cache-to-cache transfer misses occur when the requested line is in the *Private* state, that is, there is a single processor caching the memory line in the *Modified* or in the *Exclusive* states. In this case, the home directory forwards the miss to the current owner of the line. Then, the owner sends a reply with the line directly to the requestor and a revision message to the home directory. These misses are also known as *3-Hop misses*.

4.3.2. *Mem* Misses

Memory misses appear for write accesses when there is no processor caching the requested line (the line is in an *Uncached* state), and for read accesses when the requested line is cached by zero or more than one processor (the state of the line is *Uncached* and *Shared*, respectively). The home directory has a valid copy of the memory line and satisfies the miss by accessing the main memory in order to directly provide the line.

4.3.3. *Inv* Misses

Invalidation misses, also known as *upgrade misses*, take place when a write access for a memory line comes to the directory, there are several nodes holding a copy of the line (the line is in the *Shared* state) and one of them is the processor issuing the access (that is, this processor sent an *upgrade* for the line). In this situation, the directory sends invalidation messages to all sharers of the memory line except the requesting processor. Once all the invalidation acknowledgment messages have been received, the directory provides ownership of the line to the requestor.

4.3.4. *Inv+Mem* Misses

Invalidation and access to memory misses are caused by a write access for which there are several nodes caching the line but none of them is the one issuing the access. Now, the directory must first invalidate all copies the of the line. Once it has obtained all the invalidation acknowledgments, it sends a reply with the memory line to the requesting processor. The difference between *Inv* and *Inv+Mem* misses is that for the first category, the directory does not provide a copy of the memory line since the requestor already has a valid copy .

	Directory States			
	Private	**Shared**		**Uncached**
Load Miss	*$-to-$*	*Mem*		*Mem*
Store Miss	*$-to-$*	*Inv* (Upgrade)	*Inv+Mem*	*Mem*

Table 4.1: Directory actions performed to satisfy load and store misses

Table 4.1 summarizes the actions that the directory performs to satisfy load and store misses. A load miss is either satisfied by a cache-to-cache transfer or an access to memory. On the other hand, any of the four actions could be used for store misses. Observe that *Mem* misses are a consequence of cold, capacity, conflict or coherence misses, whereas *$-to-$*, *Inv* and *Inv+Mem* misses fall into the coherence misses category. Whereas capacity and conflict misses could be reduced by increasing the size of second level caches and their associativity, this is not true for coherence misses which are the result of inter-process communication. In this thesis, our efforts will be focused on coherence misses.

Finally, for the applications used in this thesis, Table 4.2 shows the percentage of L2 cache misses falling in each one of the four categories. These results have been obtained using the base configuration and running each one of the applications with the maximum number of nodes available in each case.

As observed for most applications, a great number of L2 cache misses belong to the first category. Cache-to-cache transfer misses (*$-to-$* misses) represent more than 30% of the total miss rate for all the applications but BARNES, CHOLESKY and WATER-SP. This is not new and 3-hop misses have been previously identified as constituting an important

Benchmark	$-to-$ miss	Inv miss	Mem miss	Inv+Mem miss
BARNES-HUT	26.86%	16.93%	54.96%	1.25%
CHOLESKY	15.63%	5.19%	78.30%	0.88%
EM3D	34.07%	34.06%	31.86%	0.01%
FFT	54.13%	37.20%	0.73%	7.94%
MP3D	57.64%	12.78%	27.44%	2.14%
OCEAN	33.58%	28.01%	36.43%	1.98%
RADIX	40.82%	1.72%	57.13%	0.33%
UNSTRUCTURED	63.30%	28.12%	8.47%	0.11%
WATER-NSQ	38.87%	25.79%	34.90%	0.44%
WATER-SP	13.70%	3.77%	80.97%	1.56%

Table 4.2: Percentage of *$-to-$*, *Inv*, *Mem* and *Inv+Mem* misses found in the applications used in this thesis

fraction of the L2 misses, not only scientific applications but also in the commercial area [3, 11, 28, 46, 61, 92]. Also, *Inv* and *Mem* misses represent an important fraction of the L2 cache miss rate for most applications. On the other hand, the percentage of *Inv+Mem* misses is kept very small in all cases.

In addition to the percentage of L2 misses falling into each one of the categories presented above, another important question is where the cycles needed to satisfy a certain miss type are spent. In particular, we considered two main components of the average L2 miss latency: *network latency* and *directory latency*. Network latency comprises the cycles spent on the interconnection network. For *$-to-$* misses it includes: first, network cycles to reach the corresponding home directory; second, the cycles the miss travels from the home directory to the current owner node, and third, network cycles to come back to the requesting node. For *Inv*, *Mem* and *Inv+Mem* misses we consider that network latency is the time needed to, first, reach the corresponding home directory and, next, come back to the requesting node. Note that, for an *Inv* miss (for example), the cycles the invalidation messages are travelling over the interconnect are not counted as network latency for the miss, since it is currently waiting at the directory.

On the other hand, directory latency represents the number of cycles a certain miss spends at the corresponding home directory. This includes the cycles the miss waits in directory buffers, the cycles needed to access directory information (which is usually stored in main memory), and also those needed to create and send coherence messages as well as to process the corresponding responses.

For each one of the former categories, Figures 4.3 to 4.6 present the normalized average

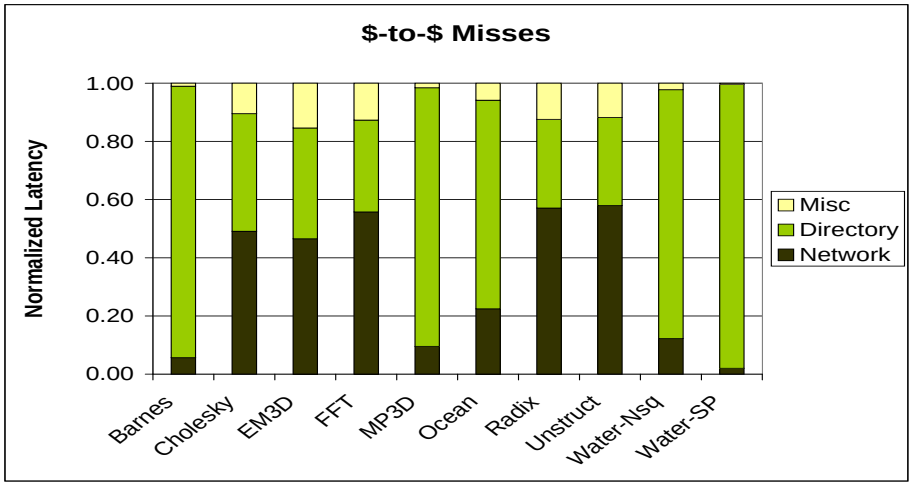


Figure 4.3: Average *$-to-$* miss latency

Figure 4.4: Average *Mem* miss latency

miss latency obtained when running the applications used in this thesis on the base system. Average miss latency is split into network latency, directory latency and miscellaneous latency (the latter includes the cycles employed on buses, cache accesses...). As can be observed, the most important fraction of the average miss latency is caused by the directory. This is true for all applications in the *Inv* and *Inv+Mem* cases, whereas only four applications found network latency to exceed directory one in the *Mem* and *$-to-$* categories. For *$-to-$* and *Mem* misses, directory latency is caused by the access to main memory to obtain the corresponding directory entry (for *Mem* misses, memory line lookup occurs in parallel with the access to the directory information, as in [54]). For *Inv* and *Inv+Mem* misses,

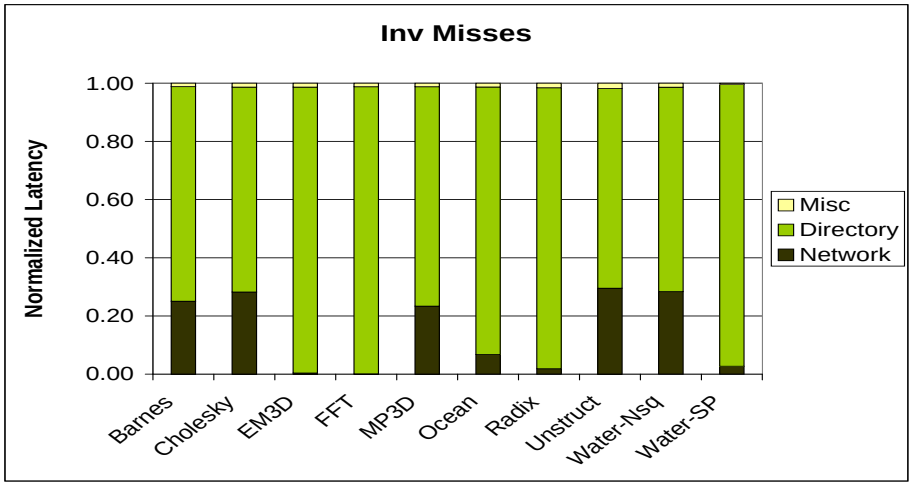


Figure 4.5: Average *Inv* miss latency

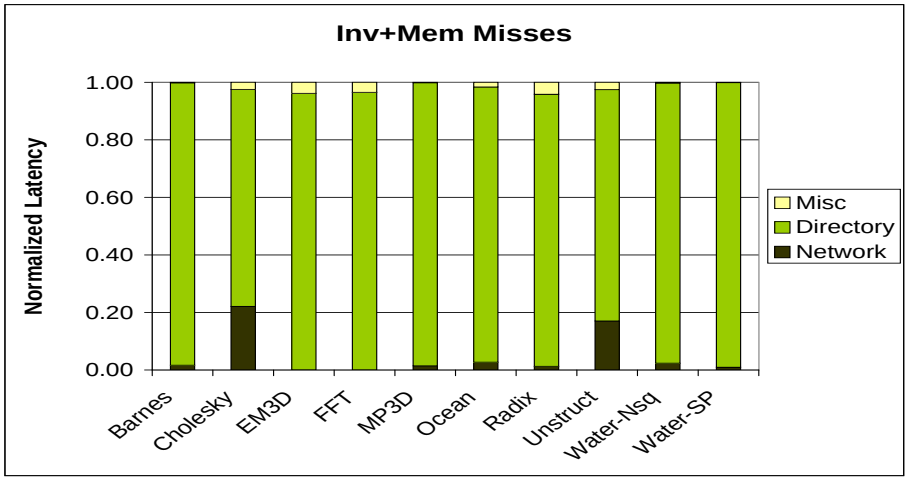


Figure 4.6: Average *Inv+Mem* miss latency

directory latency comprises the cycles needed to obtain directory information and to create and send invalidation messages, as well as to receive the corresponding acknowledgments (again, for a *Inv+Mem* miss the directory entry and the requested line are obtained at the same time).

4.4. A Novel Architecture to Reduce L2 Miss Latency

The last section presented a taxonomy of L2 misses in terms of the actions carried out by directories to satisfy them, and identified directory latency to constitute the most

important component of the L2 miss latency for the majority of the applications.

Assuming that nodes similar to that presented in Figure 4.1 are used to form a cc-NUMA multiprocessor (as described in [35]), in this section we propose a novel node organization especially designed to reduce the latency of each one of the previous categories by significantly decreasing the component of the latency caused by the access to the directory. Additionally, our proposal minimizes the memory overhead caused by directory information.

4.4.1. Node Architecture

The proposed node organization adds several elements to the basic node architecture shown in Figure 4.1. In particular, the directory is organized as a *three-level* structure and a shared cache for those memory lines that are frequently accessed by several nodes is included.

4.4.1.1. Three-Level Directory Architecture

Our proposal extends the two-level architecture presented in Chapter 2 adding a new directory level which is integrated into the processor chip. Two-level directories were originally proposed for reducing directory memory overhead whilst retaining performance. However, we can take advantage of their uncoupled nature and use them to also reduce L2 miss latency. It is important to note that coherence hardware, memory controller and network router are already included inside the processor die.

The three-level directory architecture included in our proposal consists of:

1. *First-level directory*: This directory level is located inside the processor chip, close to the directory controller. It is managed as a cache and uses a small set of entries, each one containing a precise sharing code consisting of three pointers (of $\log_2 N$ bits each one, for a N-node system). Note that, as shown in [32], a small number of pointers generally is sufficient to keep track of the nodes caching a memory line. Besides, from the graphs presented in [23], one can conclude that for more than 90% of the cases, memory lines are shared among up to 3 sharers for most applications. Therefore, having 3 pointers per line is a good compromise between cost and performance. On the other hand, choosing a sharing code linearly dependent on the number of processors (such as full-map) may make it infeasible to be incorporated

on the processor chip, compromising both scalability and performance. In short, it is preferable to invest the transistor budget dedicated to this directory in increasing the number of entries (which increases the hit rate) rather than directory width.

2. *Second-level Directory*: It is located outside the processor chip and also has a small number of entries. In this case, a non-scalable but precise full-map sharing code is employed. The second-level directory can be seen as a *victim cache* of the first level since it contains those entries that have been evicted from the first-level directory or do not fit there due to the number of sharers becoming larger than the available number of pointers (three, in our case). It is important to note that, if a directory entry for a certain memory line is present at the first level, it cannot be present at the second, and vice versa.

3. *Third-level Directory*: This level constitutes the complete directory structure (i.e., an entry per memory line) and is located near main memory (it could be included in main memory). Each entry in this level uses a compressed sharing code, in particular, the binary tree with subtrees (*BT-SuT*) sharing code (see Section 3.3.3 for details), which has space complexity O($log_2(N)$), for a N-node system. This sharing code provides performance numbers comparable to the ones obtained with gray-tristate sharing code but using approximately half the number of bits. Sharing information in this level is always updated when changes in the first or second-level directories are performed.

Accesses to the third-level directory imply main memory latency. The first-level directory has the latency of a fast on-chip cache whereas the second-level one provides data at the same speed as an off-chip cache. This way, *$-to-$*, *Inv* and *Inv+Mem* misses would be significantly accelerated if their corresponding directory entry were found at the first or second-level directories. In this case, the directory controller could immediately forward the miss to the owner of the line, in the case of an *$-to-$* miss, or send invalidation messages to the sharers of the line, in case of *Inv* and *Inv+Mem* misses. Note that for *Inv+Mem* misses, the access to main memory would not be in the critical path of the miss since it would occur in parallel with the creation and sending of the invalidation messages. Due to the locality exhibited by memory references, we expect the first and second-level directories to satisfy most of the requests, even remote accesses to a home node. Thus, this will bring important reductions in the component of the miss latency owed to the directory.

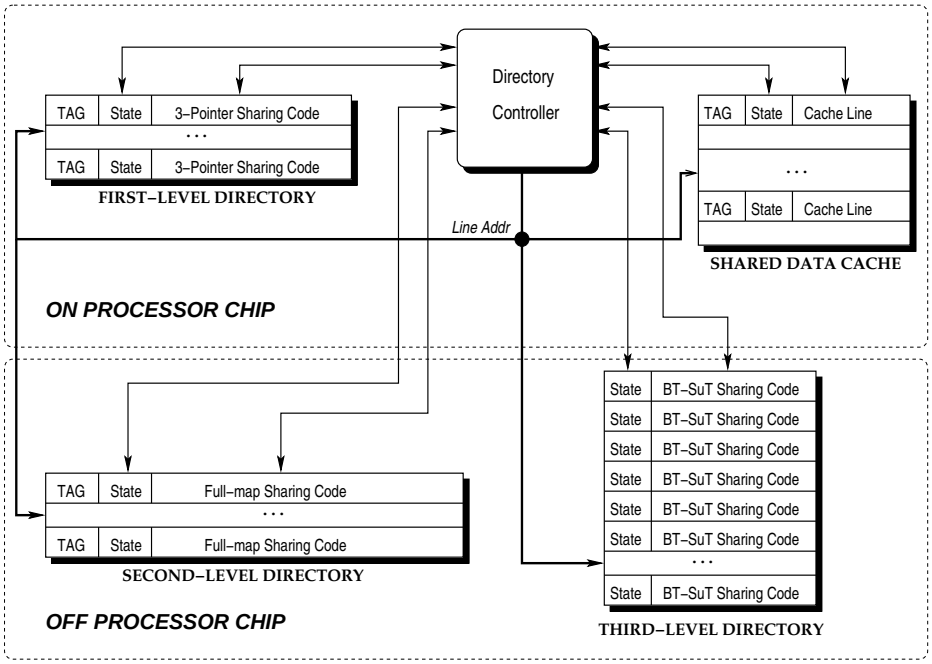


Figure 4.7: Proposed node architecture

4.4.1.2. Shared Data Cache (SDC)

Contrary to *$-to-$* misses, *Inv* misses and *Inv+Mem* misses which can significantly benefit from finding directory information in the first or second-level directories, *Mem* misses can not take full advantage of having the corresponding directory entry in these directory levels. For these misses, the directory controller must directly provide the memory line. Therefore, main memory has to be accessed, being in the critical path of the miss.

In order to also accelerate *Mem* misses, our design includes a small cache inside the processor chip, the *shared data cache (SDC)*. This cache is used by the directory controller of each home node to store a copy of those memory lines for which a valid copy is present in main memory (that is, they are in the *Shared* or *Uncached* states) and that are expected to be accessed in a near future. As coherence hardware, memory controller and network router are already included inside the processor chip, *Mem* misses can take significant advantage of finding their corresponding memory line in the shared data cache, since the access to the slower main memory would be avoided. State bits are also included in each entry of this shared data cache. The reason for this will be explained in next section.

Note that, unlike the remote data caches (RDCs) used in some systems to cache lines that are fetched to the node from remote memories, the shared data cache proposed in our

design holds some of the memory lines that having been assigned to the local node (i.e., it is the home node for that lines), are expected to be accessed in a near future, and valid copies of them are stored in main memory. Thus, this structure is used by the directory controller of the home node to reduce the latency of accessing main memory (when needed) by quickly providing data from the processor itself to the requesting node.

Figure 4.7 summarizes the node architecture proposed in this thesis. This node organization is divided into two different parts (on-chip and off-chip parts) according to their location in the processor chip. The on-chip part includes the first-level directory and the shared data cache. The off-chip structure comprises the second and third-level directories. Tag information is stored in the first and second-level directories as well as the shared data cache in order to determine whether there is a hit. The next section describes how the proposed node architecture operates to satisfy L2 misses.

4.4.2. Directory Controller Operation

Each time an L2 miss for a certain memory line reaches the directory controller, the address of the line associated with the request is sent to each one of the directory levels as well as the shared data cache and main memory. One of the following situations will take place:

1. **The directory entry is found in the first-level directory**. In this case, the directory controller obtains precise sharing information, ending the access to the second and third levels. Additionally, for *$-to-$* and *Inv* misses the accesses to the shared data cache and main memory are also cancelled since the home directory must not provide the memory line associated with the miss. For *Mem* and *Inv+Mem* misses, the access to main memory is ended if the memory line is found in the shared data cache. In all the cases, the directory controller properly updates sharing information in the first and third-level directories. After a *Mem* miss, it is possible that the final number of sharers will be greater than three. In this case, the number of pointers used in the first level is insufficient and, then, the entry in this level is freed and moved to the second level.

2. **The directory entry is found in the second-level directory**. Again, precise sharing information is obtained. The directory controller ends the access to the third-level directory whereas the access to the shared data cache has already been completed (due to the on-chip integration of the shared data cache, its latency is assumed to

be lower than or equal to the one incurred by the second-level directory). Then, the access to main memory is cancelled if a *$-to-$* or a *Inv* miss is found or, for *Mem* and *Inv+Mem* misses, if the memory line was obtained from the shared data cache. Sharing information in the second and third levels is also updated and if the final number of sharers is lower than four, the directory entry is moved from the second to the first-level directory.

3. **The directory entry is found in the third-level directory**. The third-level directory has a directory entry for each memory line, therefore it can provide sharing information for those memory lines for which an entry in the first and second-level directories has not been found. In this case, the directory controller obtains imprecise but correct sharing information which, once the miss is processed, is properly updated.

On an L2 miss for which an entry in the first-level directory was not found, an entry in this directory level is allocated only when *precise* sharing information can be guaranteed. This occurs in one of the following four cases:

1. The miss requested a private copy of the line. Once the miss has been completed, only the node that issued it caches a copy of the line.

2. The line was in the *Uncached* state. In this case, no sharing code was needed and the state of the line was obtained from the shared data cache (if the memory line was present in this structure) or from the third-level directory. Again, once the miss has been served, the line is only held by the requesting node.

3. An entry for the line was found in the second-level directory and the final number of sharers is not greater than the number of pointers used in the sharing code of the first-level directory (three, in this design).

4. The line was in the *Private* state and the directory entry was provided by the third level. In this case, the sharing code used by the third-level directory (that is, *BT-SuT*) is wide enough to precisely codify the identity of the single sharer and thus, precise sharing information is also available.

An entry in the shared data cache is allocated in one of these three situations:

Directory Receives		Directory information found in: First-level directory (DC1)	Second-level directory (DC2)	Third-level directory (MEM)	Shared data cache (SDC)
L2 Miss	Miss Type: *$-to-$*	Update entry	Move entry to DC1	Allocate an entry in DC1	Not allowed
	Mem	Update entry (Sharers≤3) or move entry to DC2 Insert in SDC (if needed)	Update entry (Sharers>3) or move entry to DC1 Insert in SDC (if needed)	Allocate an entry in DC1 (if line state = *Uncached*)	Allocate an entry in DC1 and extract line from SDC
	Inv	Update entry and extract line from SDC (if found)	Move entry to DC1 and extract line from SDC (if found)	Allocate an entry in DC1	Not allowed
	Inv+Mem				
Write-back		Free DC1 entry and insert line in SDC	Free DC2 entry and insert line in SDC	Insert line in SDC	Not allowed
$-to-$ Response		Insert line in SDC (if load miss)	Insert line in SDC (if load miss)	Nothing	Not allowed

Table 4.3: Directory controller operation

1. On a *Mem* miss for which an entry is present in the first or second-level directories. In this case, there are several sharers for the line since otherwise, a directory entry in one of these levels would have not been used. Note also that the miss had to be caused by a load instruction (otherwise, an *Inv* or *Inv+Mem* miss would have been obtained).

2. On a write-back message from the owner node of the line. This message is caused by a replacement from the single L2 cache holding the memory line and always includes a copy of the line[1]. The line is inserted into the shared data cache (and also into main memory if it was modified). Since no sharing code is necessary, the state bits of the associated entry in the shared data cache are used to codify the state of the line (*Uncached*, in this case). In this way an entry in the first or second-level directories will not have been wasted.

3. On receiving the response to a *$-to-$* miss indicating that a cache-to-cache transfer was performed, when the miss was caused by a load instruction. In this case, the state of the line was changed from *Private* to *Shared*, and the acknowledgment message contains a copy of the line[1], which is included at this moment into the shared data cache (and also into main memory if it was modified).

An entry in the first and second-level directories is freed each time a write-back message for a memory line in the *Private* state is received. This means that this line is no longer cached in any of the system nodes, so its corresponding directory entry is made

[1]The original coherence protocol did not include a copy of the line neither in a write-back message nor in a cache-to-cache transfer response when the copy contained in main memory was valid.

available for other lines. An entry in the shared data cache is freed in two cases: first, after a miss that obtains an exclusive copy of the memory line is processed. This takes place when either the miss was caused by a store instruction or the memory line was in the *Uncached* state (remember we are assuming a four-state MESI coherence protocol). And, second, when its associated directory entry is evicted from the second-level directory.

Replacements from the second-level directory are discarded, since correct information is present in the third-level directory. The same occurs for those memory lines that are evicted from the shared data cache, since main memory contains a valid copy of them. Finally, replacements in the first and second-level directories are not allowed for the entries associated with those memory lines with pending coherence transactions. Table 4.3 summarizes the most important issues described in this section.

4.4.3. Implementation Issues

In this work we assume that the organization of the first and second-level directory caches as well as of the shared data cache is fully associative, with a LRU replacement policy[2]. Each line in the first and second-level directories contains a single directory entry. The state bits field used in every entry of the shared data cache are implemented using the already included state bits of each entry. Now, three states are possible for a line in the shared data cache: not present, present in the *Uncached* state or present in the *Shared* state.

We assume that the directory controller provides write buffers to update the three directory levels, the shared data cache and main memory. Thus, writes to each one of the directory levels as well as to main memory and to the shared data cache are assumed to occur immediately.

Finally, the first and second-level directories are implemented as directory caches, using the usual technologies for on-chip and off-chip processor caches, respectively. The small sharing code used for the third-level directory would avoid the need of external storage for this directory level since, as in [12] and [72], it could be directly stored in main memory by computing ECC at a coarser granularity and utilizing the unused bits. This approach leads to lower cost by requiring fewer components and pins and provides a simpler system scaling.

[2] Again, practical implementations can be set-associative, achieving similar performance at lower cost.

4.5. Performance Evaluation

This section presents a performance evaluation for the novel architecture described previously. Through extensive simulation runs, we compare three system configurations, using the maximum number of processors available for each application (that is to say, 64 processors for all the applications except OCEAN and UNSTRUCTURED, for which a maximum of 32 processors could be simulated).

The compared systems are the *base* system and two configurations which use the node architecture proposed in this chapter. The first one (*UC* system), which shows the potential of our proposal, uses an unlimited number of entries in the first and second-level directories as well as in the shared data cache. The second one (*LC* system) limits the number of entries in these structures. For the three system configurations, the directory controller is assumed to be included inside the processor chip (that is, the value of the *Directory Cycle* parameter presented in Table 2.2 is now one cycle for the three configurations). The base system uses full-map as the sharing code for its single level directory, which obtains the best results since unnecessary coherence messages do not appear. The coherence protocol used in the *UC* and *LC* configurations has been extended with the modifications introduced in Appendix A to support the use of a compressed third-level directory (remember that this implies that more messages are needed to detect some race conditions) and also to include always the memory line in write-back messages and in cache-to-cache transfer responses (which increases the number of cycles needed by these messages to reach the corresponding home directory). On the contrary, the base system does not include these overheads.

In this section we present and analyze simulation results for the base and *UC* systems as well as for two instances of the *LC* configuration. The first one, *LC-1*, limits the number of entries used in the first and second-level directories and in the shared data cache to 512, 256, and 512, respectively. This results in total sizes of less than 2 and 3 KB for the first and second-level directories, respectively, and of 32 KB for the shared data cache. The other instance, *LC-2*, increases the number of entries of all components to 1024, resulting in total sizes of 3 KB, 10 KB and 64 KB for the first and second-level directory and shared data cache, respectively. In all cases, the sizes of these components represent a small percentage of the L2 size (12.5% in the worst case). The latencies assumed for the first and second-level directories and for the shared data cache are 1 cycle, 10 cycles, and 6 cycles, respectively.

4.5.1. Impact on L2 Miss Latencies

In Section 4.3 we classified L2 misses according to the actions performed by the directory to satisfy them into cache-to-cache transfer misses (*$-to-$*), access to memory misses (*Mem*), invalidation misses (*Inv*), and invalidation and access to memory misses (*Inv+Mem*). This section analyzes how the node architecture presented in this chapter impacts on the latency of each of these categories.

Tables 4.4 to 4.7 show the directory structures that are involved when solving each miss type for the *LC-1*, *LC-2* and *UC* configurations. The first and second columns (*DC1* and *DC2*) present the percentage of misses for which directory information is obtained from the first and second-level directories, respectively (and additionally for *Mem* misses, the corresponding memory line is found in the shared data cache (*SDC*)). Thus, these misses would benefit from the novel node architecture proposed in this chapter, contrary to the ones shown in the column *MEM*, for which main memory must be accessed. Note that all the accesses in the base configuration are to the main memory. As observed, for the majority of the applications, the most important fraction of the misses can be managed using information from the two first directory levels and the shared data cache, especially when the *LC-2* configuration is employed. However, some applications still require the third-level directory to provide sharing information for an important percentage of certain miss types. Note that obtaining the directory information from the third-level directory can entail two negative effects on performance: first, the miss must wait until main memory provides the corresponding directory entry and second, in some situations *unnecessary coherence messages* could appear as a consequence of the compressed nature of the third-level directory.

Figures 4.8 to 4.11 illustrate the normalized average latency for each miss type split into network latency, directory latency and miscellaneous latency (buses, cache accesses...), for the base, *UC*, *LC-1* and *LC-2* configurations. Normalized average latencies are computed dividing the average latencies for each one of the configurations by the average latencies for the base case. As can be seen from the results obtained for the *UC* configuration, the integration into the processor die of the first-level directory and the shared data cache (*SDC*) has the potential of significantly reducing the latency of L2 misses. This is a consequence of the important reduction in the component of the latency associated with the directory.

	LC-1 Configuration			LC-2 Configuration			UC Configuration		
Application	*DC1*	*DC2*	*MEM*	*DC1*	*DC2*	*MEM*	*DC1*	*DC2*	*MEM*
BARNES-HUT	100.00%	0.00%	0.00%	100.00%	0.00%	0.00%	100.00%	0.00%	0.00%
CHOLESKY	63.84%	3.87%	32.29%	78.90%	7.40%	13.70%	100.00%	0.00%	0.00%
EM3D	100.00%	0.00%	0.00%	100.00%	0.00%	0.00%	100.00%	0.00%	0.00%
FFT	11.11%	9.68%	79.21%	35.70%	46.20%	18.10%	100.00%	0.00%	0.00%
MP3D	99.72%	0.28%	0.00%	100.00%	0.00%	0.00%	100.00%	0.00%	0.00%
OCEAN	76.04%	2.85%	21.11%	81.67%	9.97%	8.36%	100.00%	0.00%	0.00%
RADIX	10.65%	3.03%	86.32%	17.86%	45.26%	36.88%	100.00%	0.00%	0.00%
UNSTRUCTURED	100.00%	0.00%	0.00%	100.00%	0.00%	0.00%	100.00%	0.00%	0.00%
WATER-NSQ	100.00%	0.00%	0.00%	100.00%	0.00%	0.00%	100.00%	0.00%	0.00%
WATER-SP	100.00%	0.00%	0.00%	100.00%	0.00%	0.00%	100.00%	0.00%	0.00%

Table 4.4: How *$-to-$* misses are satisfied

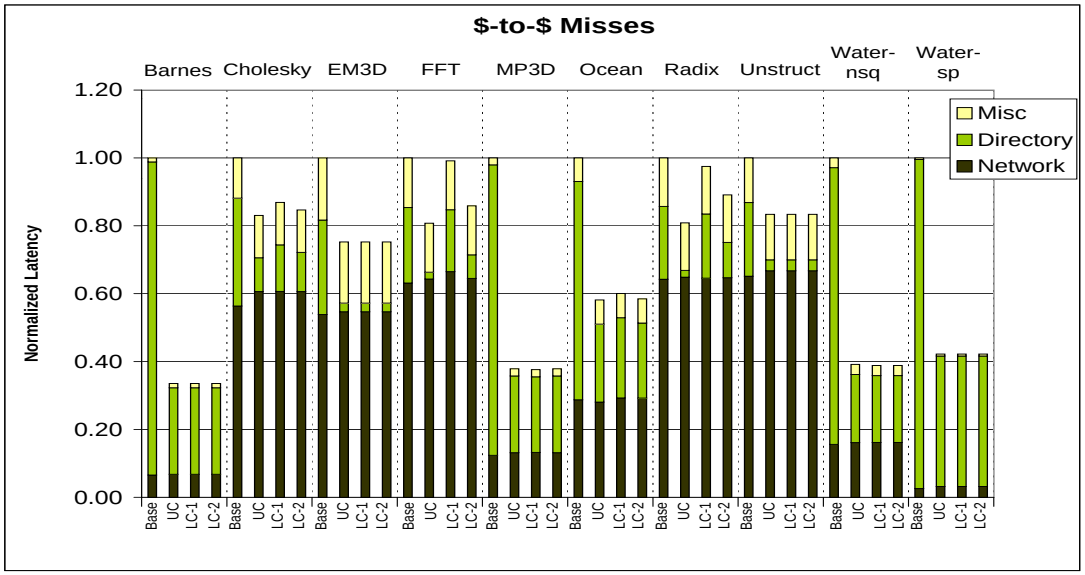


Figure 4.8: Average *$-to-$* miss latency

4.5.1.1. Impact on *$-to-$* Miss Latencies

As shown in Figure 4.8, the small number of entries used in the *LC-1* case for the directory caches as well as for the shared data cache suffices to virtually reach the latency reductions found in the *UC* case for BARNES (66%) , CHOLESKY (13%), EM3D (25%), MP3D (62%), OCEAN (40%), UNSTRUCTURED (17%), WATER-NSQ (62%) and WATER-SP (58%). As derived from Table 4.4, for FFT and RADIX, 80% or more of the misses requiring a cache-to-cache transfer need the third-level directory to provide the sharing information, which motivates the small benefit obtained by the *LC-1* configuration for these applications. The increased number of entries used in the *LC-2* case significantly reduces this percentage. Now, only 18.10% for FFT and 36.88% for RADIX of the *$-to-$* misses

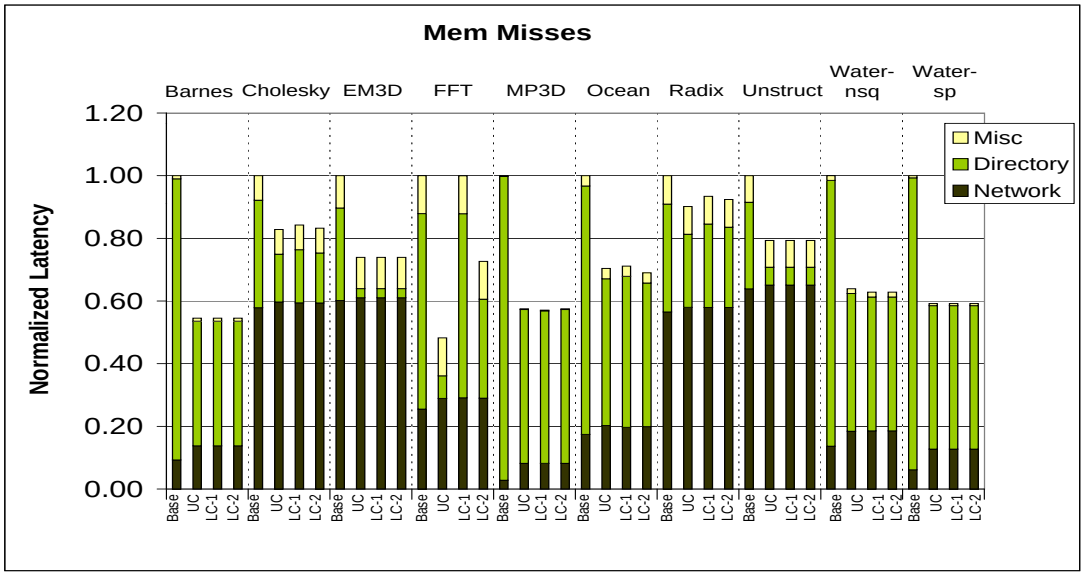


Figure 4.9: Average *Mem* miss latency

must wait for main memory latency to obtain the sharing information and reductions of 14% for FFT and 11% for RADIX in average latency are observed. Fortunately, unnecessary coherence messages cannot appear for *$-to-$* misses, even when the third-level directory is accessed. This is because the compressed sharing code used by this directory level (that is, *BT-SuT*) is wide enough to exactly codify the identity of a single sharer, which is the case for these misses.

4.5.1.2. Impact on *Mem* Miss Latencies

Application	LC-1 Configuration			LC-2 Configuration			UC Configuration		
	DC1 +SDC	*DC2 +SDC*	*MEM*	*DC1 +SDC*	*DC2 +SDC*	*MEM*	*DC1 +SDC*	*DC2 +SDC*	*MEM*
BARNES-HUT	32.22%	67.38%	0.40%	32.22%	67.38%	0.40%	32.22%	67.38%	0.40%
CHOLESKY	15.87%	49.73%	34.40%	16.29%	52.89%	30.82%	16.34%	57.96%	25.70%
EM3D	96.80%	3.20%	0.00%	96.80%	3.20%	0.00%	96.80%	3.20%	0.00%
FFT	0.04%	6.39%	93.57%	0.04%	33.20%	66.76%	0.08%	99.92%	0.00%
MP3D	16.00%	83.93%	0.07%	15.68%	84.29%	0.03%	15.68%	84.29%	0.03%
OCEAN	16.50%	31.30%	52.20%	16.53%	47.49%	35.98%	19.21%	56.43%	24.36%
RADIX	2.04%	23.75%	74.21%	2.05%	27.63%	70.32%	3.26%	33.78%	62.96%
UNSTRUCTURED	14.44%	84.26%	1.30%	14.44%	84.26%	1.30%	14.44%	84.26%	1.30%
WATER-NSQ	9.44%	90.52%	0.04%	9.44%	90.52%	0.04%	9.59%	90.37%	0.04%
WATER-SP	10.39%	89.58%	0.03%	10.39%	89.58%	0.03%	10.39%	89.58%	0.03%

Table 4.5: How *Mem* misses are satisfied

Figure 4.9 shows the normalized average latencies for *Mem* misses. For these misses, the directory must provide the memory line directly from either the shared data cache

(*SDC*) or main memory. Therefore, unnecessary coherence messages can not appear. Latency reductions of 45% for BARNES, 16% CHOLESKY, 26% for EM3D, 43% for MP3D, 29% for OCEAN, 21% for UNSTRUCTURED, 37% for WATER-NSQ and 40% for WATER-SP are obtained for the *LC-1* configuration. The reason for the significant reductions observed in these applications, is that a large number of the *Mem* misses obtain the memory line from the shared data cache and the corresponding directory entry in one of the two first directory levels (or in the own shared data cache for lines in the *Uncached* state), saving the access to the slower main memory (see Table 4.5). Again, the former values are very similar to those provided by the *UC* configuration. For FFT, the *LC-1* configuration is unable to reduce the latency of *Mem* misses since more than 93% of them must obtain the data from main memory. When moving to the *LC-2* configuration, this percentage decreases and a reduction of 28% is obtained. However, this reduction still remains far from the potential found for the *UC* configuration (52%). Finally, small latency reductions are obtained for RADIX when the *UC* configuration is used (only a 10%). As shown in Table 4.5, a large percentage of the *Mem* misses suffered by this application (62.96%) are for lines that are accessed for the first time (cold misses), which prevents a shared data cache with an unlimited number of entries from being able to provide the memory line. The percentage of *Mem* misses satisfied by main memory is even higher for *LC-1* and *LC-2* configurations, and latency reductions of 7% and 8%, respectively, are observed.

4.5.1.3. Impact on *Inv* Miss Latencies

	LC-1 Configuration			LC-2 Configuration			UC Configuration		
Application	*DC1*	*DC2*	*MEM*	*DC1*	*DC2*	*MEM*	*DC1*	*DC2*	*MEM*
BARNES-HUT	64.85%	35.15%	0.00%	64.85%	35.15%	0.00%	64.85%	35.15%	0.00%
CHOLESKY	86.87%	5.89%	7.24%	88.04%	7.29%	4.67%	88.37%	11.63%	0.00%
EM3D	75.43%	24.57%	0.00%	75.43%	24.57%	0.00%	75.43%	24.57%	0.00%
FFT	22.55%	12.20%	65.25%	47.47%	52.08%	0.45%	100.00%	0.00%	0.00%
MP3D	98.86%	1.14%	0.00%	98.86%	1.14%	0.00%	98.86%	1.14%	0.00%
OCEAN	84.95%	4.44%	10.61%	89.40%	7.92%	2.68%	97.38%	2.62%	0.00%
RADIX	73.14%	0.11%	26.75%	73.18%	0.09%	26.73%	92.57%	7.43%	0.00%
UNSTRUCTURED	98.52%	1.48%	0.00%	98.52%	1.48%	0.00%	98.52%	1.48%	0.00%
WATER-NSQ	96.73%	3.27%	0.00%	96.73%	3.27%	0.00%	96.73%	3.27%	0.00%
WATER-SP	51.14%	48.86%	0.00%	51.14%	48.86%	0.00%	51.14%	48.86%	0.00%

Table 4.6: How *Inv* misses are satisfied

Figure 4.10 shows the normalized average latency for *Inv* misses. In this case, the directory must send invalidation messages and receive their corresponding acknowledgments. Therefore, the main component of the latency is caused by the directory. For *Inv* misses, *LC-1* and *UC* configurations obtain virtually identical latency reductions for

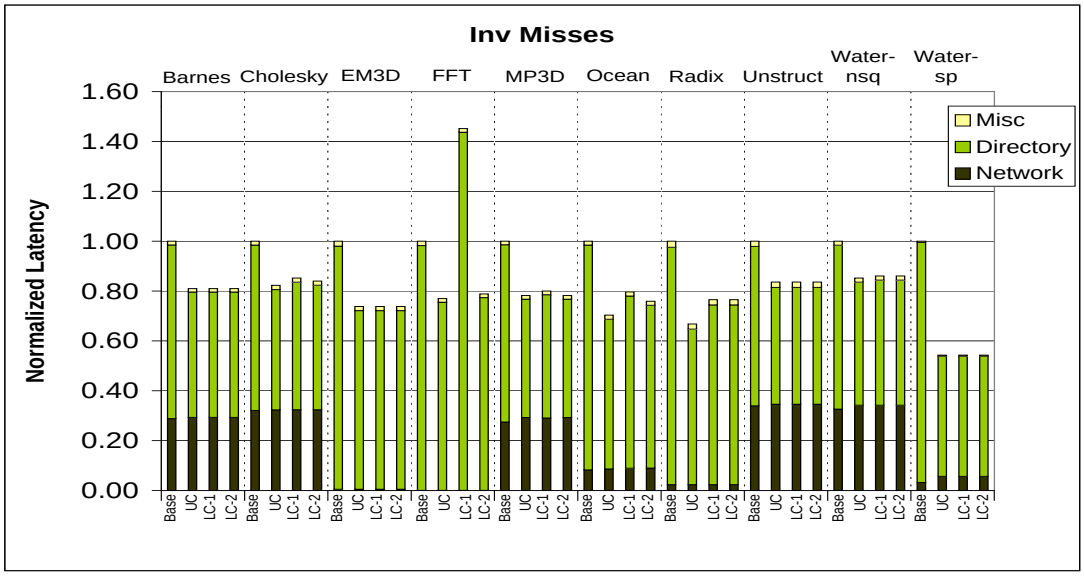


Figure 4.10: Average *Inv* miss latency

BARNES (20%), CHOLESKY (15%), EM3D (26%), MP3D (20%), UNSTRUCTURED (16%), WATER-NSQ (15%) and WATER-SP (45%). As observed in Table 4.6, 26% of the *Inv* misses in the *LC-1* and *LC-2* configurations must obtain sharing information from main memory for RADIX application. In this case, the compressed sharing code used in the third-level directory does not increase the number of invalidation messages observed for the base case, which explains the small difference between the reduction obtained for the *UC* configuration and the ones obtained for both the *LC-1* and the *LC-2* cases (only 10%). For FFT, again, an important fraction of the *Inv* misses do not find directory information in the first or second-level directories when the *LC-1* configuration is used (65.25%). In this case, however, the compression of the third-level directory implies a loss of precision that increases the number of invalidation messages sent per invalidation event when compared with the base system and consequently, the average time needed to satisfy these misses. This explains the significant latency increment (more than 40%) shown in Figure 4.10. This performance degradation practically disappears when the number of entries in the first and second-level directories is increased and the *LC-2* configuration can obtain the latency reduction of 23% observed for the *UC* case. Finally, for OCEAN, 11% of the *Inv* misses obtain directory information from the third level when the *LC-1* configuration is used. Moving to *LC-2* reduces this fraction to 3%, and latency reductions very close to those obtained with the *UC* configuration are seen (24% for *LC-2* and 30% for *UC*). In

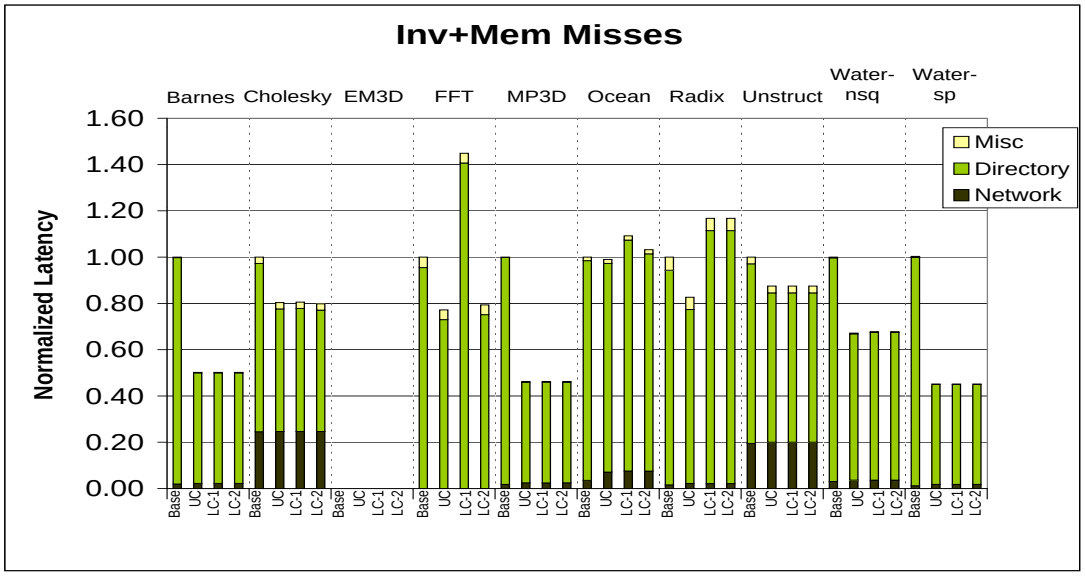


Figure 4.11: Average *Inv+Mem* miss latency

general, finding directory information in the first or second-level directory for an *Inv* miss reduces the number of cycles required to find the identity of current sharers of the memory line. However, invalidation messages must be sent to them and their associated responses received, before the directory can complete the miss and reply to the requesting node. In Chapter 6, we will present a technique that allows to overlap access to the directory and invalidation of the sharers, for *Inv* misses.

4.5.1.4. Impact on *Inv+Mem* Miss Latencies

	LC-1 Configuration			LC-2 Configuration			UC Configuration		
Application	*DC1*	*DC2*	*MEM*	*DC1*	*DC2*	*MEM*	*DC1*	*DC2*	*MEM*
BARNES-HUT	52.80%	47.20%	0.00%	52.80%	47.20%	0.00%	52.80%	47.20%	0.00%
CHOLESKY	95.50%	3.00%	1.50%	96.02%	3.98%	0.00%	97.13%	2.87%	0.00%
EM3D	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
FFT	14.77%	14.90%	70.33%	44.93%	54.79%	0.28%	100.00%	0.00%	0.00%
MP3D	52.29%	47.71%	0.00%	52.00%	48.00%	0.00%	52.00%	48.00%	0.00%
OCEAN	25.63%	20.50%	53.87%	39.70%	38.77%	21.53%	85.61%	14.39%	0.00%
RADIX	3.18%	0.00%	96.82%	3.18%	0.00%	96.82%	79.24%	20.76%	0.00%
UNSTRUCTURED	76.09%	23.91%	0.00%	76.09%	23.91%	0.00%	76.09%	23.91%	0.00%
WATER-NSQ	54.99%	45.01%	0.00%	54.99%	45.01%	0.00%	57.06%	42.94%	0.00%
WATER-SP	39.29%	60.71%	0.00%	39.29%	60.71%	0.00%	39.29%	60.71%	0.00%

Table 4.7: How *Inv+Mem* misses are satisfied

Figure 4.11 illustrates the normalized average latency for *Inv+Mem* misses. When the *LC-1* configuration is used, important latency reductions are found for *Inv+Mem* misses

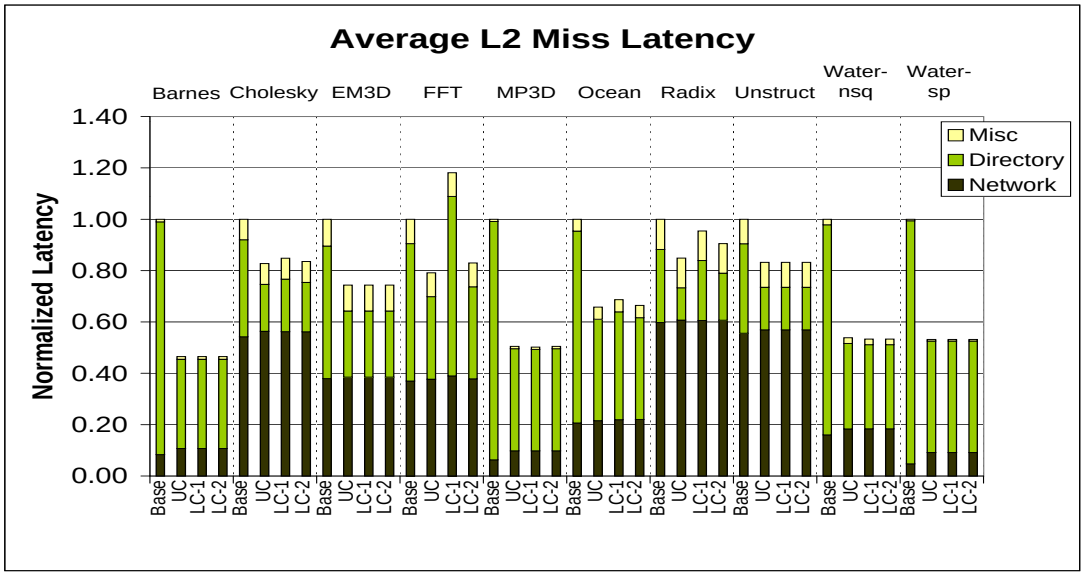


Figure 4.12: Average L2 miss latency

for BARNES (50%), CHOLESKY (20%), MP3D (54%), UNSTRUCTURED (13%), WATER-NSQ (32%) and WATER-SP (55%), which coincide with the ones obtained for the *UC* case. As shown in Table 4.7, this kind of miss was not found in EM3D application. As for the *Inv* misses, the use of the imprecise third-level directory increases the latency of *Inv+Mem* misses for some applications. In particular, when the *LC-1* configuration is used, latency degradations of more than 40% for FFT, 9% for OCEAN and 20% for RADIX are observed. These degradations are completely eliminated in FFT with the use of the *LC-2* configuration and the performance of the *UC* case is obtained. However, the situation does not improve much when moving to the *LC-2* configuration in OCEAN and RADIX, and 21.53% and 96.82%, respectively, of the *Inv+Mem* misses must still obtain directory information from main memory.

4.5.2. Impact on Execution Time

The ability of the proposed node architecture to reduce application execution times will depend on the reductions in the average latency for each miss type, the percentage of L2 misses belonging to each category and the weight these misses have on the application execution time.

For the applications used in our study, Table 4.2 presented in Section 4.3 showed the percentage of L2 misses belonging to each type. In this section, Figures 4.12 and 4.13

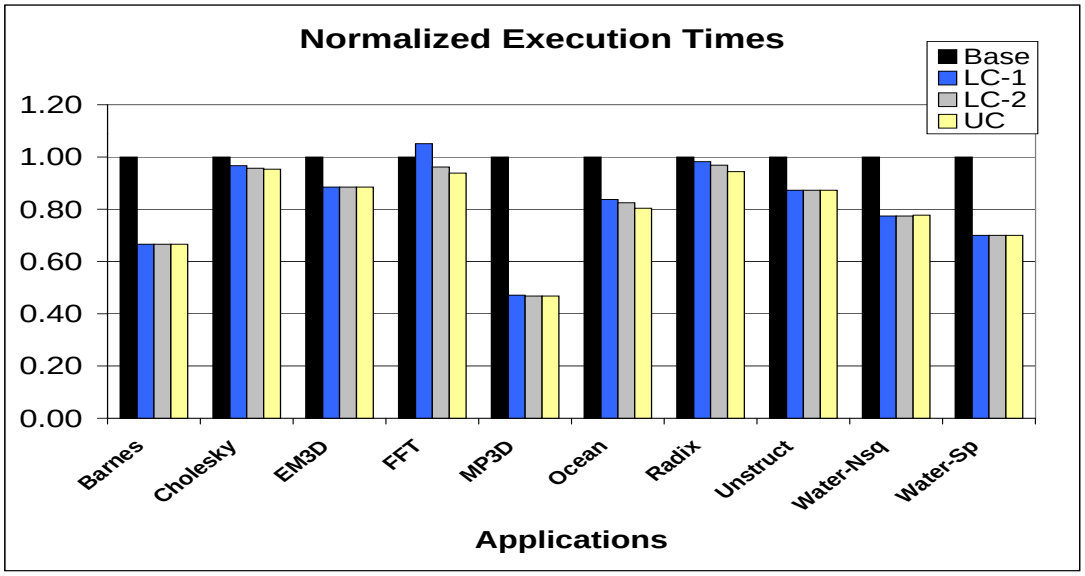


Figure 4.13: Normalized execution times

illustrate the normalized average L2 miss latency and execution time, respectively, for each one of the configurations considered in this thesis.

As shown in Figure 4.12, our proposal slightly increases the average latency due to the network in some applications (up to 4% for WATER-SP) when compared with the base configuration. This is caused by (1) the presence of unnecessary coherence messages and (2) the inclusion of the memory line in all the cache-to-cache transfer responses and in the write-back messages. However, important average L2 miss latency reductions are obtained for those applications that can take significant advantage of the on-chip integration of the first-level directory and the shared data cache. L2 misses have been significantly accelerated in BARNES, MP3D, OCEAN, WATER-NSQ and WATER-SP, which motivates the important reductions on the average L2 miss latency observed for these applications (56% for BARNES, 49% for MP3D, 35% for OCEAN, 46% for WATER-NSQ and 47% for WATER-SP). These reductions finally translate into significant improvements in terms of execution time (reductions of 34% for BARNES, 53% for MP3D, 20% for OCEAN, 22% for WATER-NSQ and 30% for WATER-SP). More modest reductions on L2 miss latencies were found for CHOLESKY, EM3D, UNSTRUCTURED and reductions of 17%, 25% and 18%, respectively, on average L2 miss latency are obtained, resulting in reductions of 5%, 11% and 13%, respectively, on execution time. For FFT application, *Inv* and *Inv+Mem* misses represent more than 45% of the total miss rate, as illustrated in Table 4.2. Hence, the performance

degradation observed for these misses when using the *LC-1* configuration translates into an increase on the average L2 miss latency of 18% and finally, on execution time of 5%. This degradation disappeared when moving to the *LC-2* configuration and reductions on average L2 miss latency and execution time close to the ones reached with the *UC* configuration are obtained (execution time reductions of 4% and 6% for the *LC-2* and *UC* cases, respectively). Finally, only *Inv* misses could be significantly accelerated for RADIX application (reduction of 25% for the *LC-1* and *LC-2* configurations). However, as shown in Table 4.2, only 1.72% of the misses belong to this category, which explains the low benefit observed for the *LC-2* and *UC* configurations (reductions of 3% and 5% on execution time, respectively).

4.6. Conclusions

Two-level directories (multi-level directories, in general) were presented in Chapter 3 as an effective technique to significantly reduce directory memory overhead whilst retaining the performance of a full-map directory. In this chapter, we extend the two-level directory architecture previously presented and apply it to also reduce the latency of L2 cache misses.

Our proposal is based on both current technology improvements and the uncoupled nature of multi-level directories. The increased integration scale allows designers to introduce some key components of the system inside the processor chip, such as the memory controller, coherence hardware and network interface/router. In this way, organizing the directory as a multi-level structure also makes the inclusion of a small first-level directory into the processor chip a feasible option.

Moreover, our proposal replaces the traditional single-level directory with a novel three-level directory architecture and adds a small shared data to each one of the nodes that form the multiprocessor. The first-level directory as well as the small shared data cache are integrated into the processor chip of every node, whereas the second and third-level directories are placed outside the processor. On-chip integration of the small first-level directory (which uses a limited number of pointers as sharing code) and the shared data cache ensures performance, since access to main memory is avoided when the directory controller finds the information in these structures. Whereas, the memory overhead entailed by directory information is significantly decreased by having two directory levels outside the processor chip. The third-level directory is a complete directory structure (one entry per

memory line) that uses our *BT-SuT* compressed sharing code to drastically reduce memory requirements, and the second-level directory is a small directory cache (using full-map sharing code) that tries to minimize the negative effects of having an imprecise third level as well as to quickly provide directory information when not present in the first level.

In order to better understand the reasons for performance improvement, a taxonomy of the L2 misses according to the actions performed by the directory to satisfy them, has been presented. Misses have been classified into cache-to-cache transfer misses (*$-to-$*), access to memory misses (*Mem*), invalidation misses (*Inv*) and invalidation and access to memory misses (*Inv+Mem*). A cc-NUMA multiprocessor using the proposed node architecture can achieve important reductions in the average miss latency of each one of these categories when compared with a traditional cc-NUMA multiprocessor in which each node includes the coherence controller into the processor chip. Latency reductions of up to 66% for *$-to-$* misses, 45% for *Mem* misses, 45% for *Inv* misses and 55% for *Inv+Mem* misses have been obtained. These reductions translate into important improvements in the application execution times (reductions of up to 53%).

The reported improvement in performance could make our architecture competitive for medium-scale systems (64-256 processors) at the same time that scalability to larger systems is guaranteed. In addition, the simplicity of our proposal and the fact that it could be easily introduced in commercial processors cuts down its cost, unlike the expensive sophisticated network designs required by moderate-scale SMPs.

Our proposal can also be combined with other protocol optimizations. For example, some directory protocols (e.g., that implemented by the Compaq AlphaServer GS320 [28]) assume that the network provides a total order between forwarded requests and invalidations, thereby eliminating the need to acknowledge invalidations. Such kind of protocols could be employed in a multiprocessor using on-chip directory integration to reduce the directory latency component of *Inv* and *Inv+Mem* misses even more.

The proposal presented in this chapter constitutes a general technique focused on reducing the latency of the four categories of the taxonomy. The idea behind the proposed architecture is to reduce the amount of time L2 cache misses spend at the directory by having directory information and the corresponding memory line (when needed) stored in small cache structures integrated into the processor die. In this way, the component of the L2 miss latency caused by the directory can be significantly reduced.

For *$-to-$* and *Inv* misses, the directory is accessed to find the identity of the nodes (or node) holding a copy of the memory line. Then, the appropriate coherence messages

(transfer requests and invalidations, respectively) are posted from the directory to them. If on suffering a *$-to-$* or an *Inv* miss, the missing node were able to know the identity of the nodes with a copy of the memory line, it could send the coherence messages directly, completely removing the access to the directory from the critical path of the miss. This would require to extend the original coherence protocol to deal with some emerging situations, as well as the development of effective prediction schemes and will be the objective of next two chapters.

Chapter 5

Owner Prediction for Accelerating Cache-to-Cache Transfer Misses

5.1. Introduction

As seen in Chapter 4, an important fraction of the L2 cache misses found in scientific parallel applications fall into the *$-to-$* category (that is, *cache-to-cache transfer misses*). For these misses, the home node has an obsolete copy of the memory line and the corresponding directory entry stores the identity of the node holding the single valid copy of the memory line (this node is said to be the *owner* node). Therefore, the role of the directory in these cases is to *redirect* the misses to the corresponding owner nodes (of course, the directory also serves as the serialization point for all accesses to a particular memory line and therefore, to ensure correctness).

Recently, several research results have reported that the percentage of cache-to-cache transfer misses (also known as 3-hop misses) found in commercial workloads is even higher (more than 60% in some cases) and cache-to-cache transfer misses are currently the subject of extensive study [3, 11, 28, 46, 61, 92].

An example of a scenario where a cache-to-cache transfer miss takes place is a producer-consumer sharing pattern. The producer holds a particular memory line in the *Modified* state. Then, the consumer issues a load for that memory line, which causes an L2 cache miss. The miss is sent to the corresponding home directory, which observes the line to be in the *Private* state and forwards the miss as a *transfer request* to the owner node (the producer, in this case). On receiving the transfer request from the directory, the producer sends a copy of the memory line to the requesting node (the consumer) as well as an

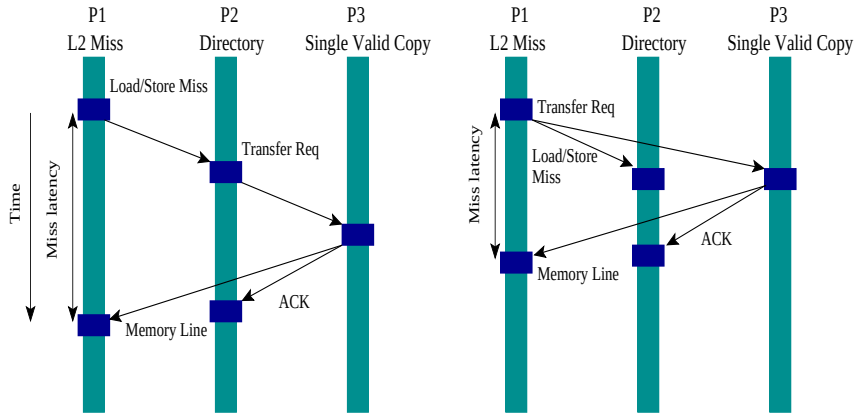


Figure 5.1: Coherence protocol operations for a 3-hop miss in a conventional cc-NUMA (left) and in a cc-NUMA including prediction (right)

acknowledgment message to the home directory (which also includes a valid copy of the memory line for load misses), and properly updates the state of its local copy of the line (to *Shared*). Finally, directory information is updated after the acknowledgment message from the owner has been received. This behavior is illustrated in Figure 5.1 (left).

One can immediately see how directory-based shared memory multiprocessors place the access to the directory information into the critical path of cache-to-cache transfer misses, which significantly penalizes them compared to SMP designs. For SMPs, cache-to-cache transfer misses are placed on the shared ordered network where all the cache controllers observe them. This way, the one that holds the valid copy of the line can immediately reply to the miss. If directory-based shared memory multiprocessors are meant to be used in the commercial marketplace, engineering decisions optimizing cache-to-cache transfer misses must be introduced. As pointed out in [92], these decisions may include faster directory checkup, no speculative read of memory in parallel with directory lookup (which will waste memory bandwidth anyway), faster interconnection network and cache-to-cache transfer support.

In this chapter we propose and evaluate *owner prediction* [3], a technique able to convert 3-hop misses into a *new type* of 2-hop misses. As shown in Figure 5.1 (right), if the requesting node had been able to "guess" where the single valid copy of the memory line resided, it would have directly sent the miss to the corresponding owner node, removing the access to the directory information from the critical path, as is done in a snooping-based design. In parallel, the miss could be sent to the corresponding home directory where the prediction would be checked. If the prediction had hit (which is the case presented in the

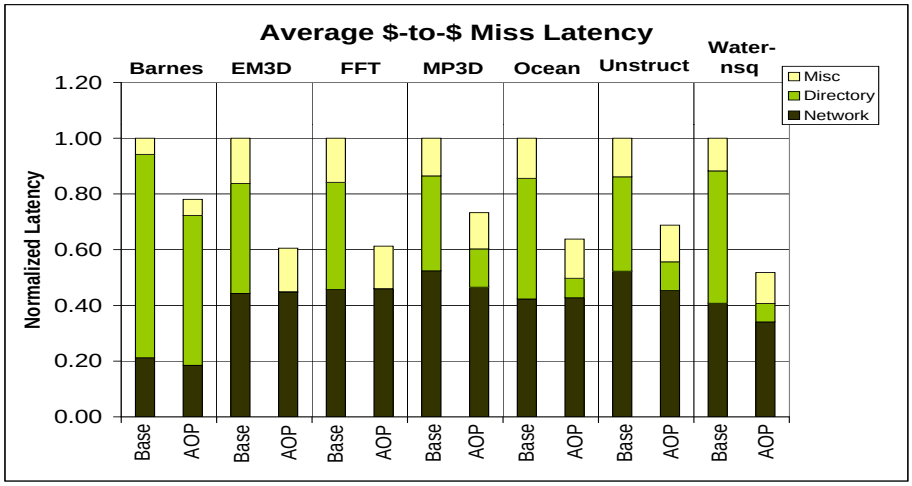


Figure 5.2: Normalized average *cache-to-cache transfer miss* latency

example) the directory would wait until the acknowledgment from the owner, and would then update directory information. Otherwise, appropriate corrective action would be taken (for example, to send the transfer request to the correct owner when the prediction was sent to a different node).

The benefits that our proposal has on cache-to-cache transfer misses are twofold. On the one hand, and most importantly, the directory component of the latency can potentially be eliminated, since now the access to the directory is not in the critical path of the miss. On the other hand, the network component of the latency is also reduced for those 3-hop misses in which the requesting processor, the home directory and the owner are in different nodes, since the cycles needed to reach the home directory are avoided. Figure 5.2 shows the reductions on average 3-hop miss latencies that are obtained when an *almost-oracle* predictor is used (details on how this predictor is implemented are given in Section 5.5). Results are shown for those applications for which more than 16% of the misses are 3-hop. As can be seen, owner prediction has the potential of significantly accelerating cache-to-cache transfer misses for all applications (reductions on average miss latency ranging from 22% for BARNES to 50% for WATER-NSQ). Although network latency is slightly reduced for some applications as MP3D, UNSTRUCTURED and WATER-NSQ, the reported benefits are mainly a consequence of a significant reduction on the directory component of the latency, which is even eliminated in some applications such as EM3D and FFT.

Owner prediction requires the development of two main elements. The first one is a prediction engine able to predict both whether an L2 cache miss is 3-hop or not and if so,

the owner of the line. We show how a predictor with a total size of less than 64 KB can deliver good prediction rates for most cases. The second one is a coherence protocol designed to support the use of prediction. Our proposal is based on the observation that 3-hop misses usually present repetitive behavior: they are caused by a small number of instructions and the set of nodes from which the missing instruction receives the corresponding memory line is small (a single node in some cases) and frequently the same. In this way, a well-tuned prediction engine could be successfully employed to capture this fact.

The rest of the chapter is organized as follows. Related work is given in Section 5.2. Section 5.3 shows the two-step prediction scheme that we propose. Extended coherence protocol is presented and justified in Section 5.4. Section 5.5 presents a detailed performance evaluation of our novel proposal. Finally, Section 5.6 concludes the chapter.

5.2. Related Work

Prediction has a long history in computer architecture and it has proved useful for improving microprocessor performance. For example, value prediction has recently been applied in the field of superscalar processors mainly to reduce the impact of data dependences [27, 30, 58, 79].

Prediction in the context of shared memory multiprocessors was first studied by Mukherjee and Hill, who showed that it is possible to use address-based[1] 2-level predictors at the directories and caches to track and predict coherence messages [66]. By accurately predicting and performing the necessary coherence operations speculatively in advance, all of the coherence overhead can be potentially eliminated, resulting in remote accesses that are as fast as local ones. The paper concentrates on showing that high prediction accuracies can be obtained as a result of predictable coherence message signatures that arise from stable sharing patterns of memory lines.

Subsequently, Lai and Falfasi modified these predictors to improve their accuracy and to reduce their size (that is, their implementation costs). Additionally, the authors presented the first design for a speculative coherent cc-NUMA using pattern-based predictors by executing coherence operations speculatively to hide the remote read latency [51]. In particular, the proposed scheme tries to predict when the producer of a particular memory line has finished writing to it. Then, the writable copy is speculatively invalidated and forwarded to the consumers.

[1] Address-based stands for predictors whose table is accessed using the effective memory address.

Finally, Kaxiras and Young [49] presented a taxonomy of all prediction schemes in a uniform space, and provided simulation results on the accuracy of a practical subset of them. Through this process, the authors derived prediction schemes that are more accurate than those previously proposed.

Alternatively, Kaxiras and Goodman [48] proposed and evaluated prediction-based optimizations of migratory sharing patterns (converting some load misses that are predicted to be followed by a store-write fault to coherent writes), wide sharing patterns (to be handled by scalable extensions to the SCI base protocol) and producer-consumer sharing patterns (pre-sending a newly created value to the predicted consumers).

Differently from these proposals, we do not attempt to predict coherence messages to, for example, pre-empt the cache-to-cache transfer altogether by speculatively transferring the data in advance. Here, we apply prediction for avoiding the indirection introduced by the access to the directory, and consequently, for accelerating the cache-to-cache transfer miss itself. In this way, we can take advantage of more knowledge by delaying the transfer to the point of the demand, which results in lower bandwidth consumption.

A key way in which many computer vendors optimize cache-to-cache transfer misses is by supporting *snooping* cache coherence. Systems using snooping –usually known as symmetric multiprocessors or SMPs– depend on broadcasting coherence transactions to all the processors and memory over a network with a completely ordered message delivery (traditionally a bus), which restricts the number of processors that can be incorporated into such designs. Some proposals trying to overcome the limitations imposed by the use of the bus have appeared.

In [61], Martin *et al.* propose a technique that allows SMPs to utilize unordered networks (with some modifications to support snooping). However, scalability to larger systems is still compromised since coherence transactions must be sent to all processors in the system which, in turn, must process them. In addition, Bandwidth Adaptive Snooping Hybrid (BASH) has been proposed to reduce the requirements that snooping protocols put on the interconnect [62]. BASH is a hybrid protocol that ranges from behaving like snooping (by broadcasting coherence transactions) when excess bandwidth is available to behaving like a directory protocol (by unicasting coherence transactions) when bandwidth is limited.

Bilir *et al.* [13] investigated a hybrid protocol that tried to achieve the performance of snooping protocols and the scalability of directory-based ones. The protocol is based on predicting which nodes must receive each coherence transaction. If the prediction hits, the

protocol approximates the snooping behavior (although the directory must be accessed in order to verify the prediction). Performance results in terms of execution time were not reported and the design was based on a network with a completely ordered message delivery, in particular an Isotach-like Fat Tree Network [77], which could restrict its scalability. Our work focuses on reducing the latency of 3-hop misses by means of predicting the node holding the single valid copy of the memory line. We can take advantage of any of the current and future high-performance point-to-point networks and it could be incorporated into cc-NUMA multiprocessors with minimal changes in the coherence protocol.

In [46], Iyer *et al.* proposed to reduce the latency of the load misses that are solved with a cache-to-cache transfer by placing small directory caches in the crossbar switches of the interconnect to capture and store ownership information as the data flows from the memory module to the requesting processor. However, the fact that special network topologies are needed to keep the information stored in these switch caches coherent represents its main drawback. Our proposal is not constrained to any network topology and it is equally applicable to reduce the latency of cache-to-cache transfer misses caused by load and store instructions.

The new Compaq AlphaServer GS320 [28] constitutes an example of cc-NUMA architecture specifically targeted at medium-scale multiprocessing (up to 64 processors). The hierarchical nature of its design and its limited scale make it feasible to use simple interconnects, such as a crossbar switch, to connect the handful of nodes, allowing a more efficient handling of cache-to-cache transfer misses than traditional directory-based multiprocessors by exploiting the extra ordering properties of the switch. On the contrary, our proposal does not require any interconnection network with special ordering.

Michael and Nanda [68] proposed the integration of directory caches inside the coherence controllers to minimize directory access time. In this way, one can use a small and fast directory cache holding sharing information only for those memory lines in the *Private* state. Those 3-hop misses for which an entry in the directory cache is found, could be accelerated by avoiding the access to the main directory, which is usually stored in main memory. Our proposal attempts to completely remove the access to the directory from the critical path of 3-hop misses.

5.3. Predictor Design for Cache-to-Cache Transfer Misses

The first component of our proposal is an effective prediction scheme that allows each node of a cc-NUMA multiprocessor to answer two key questions: first, *is an L2 miss for a certain memory line going to be served with a cache-to-cache transfer?*, and second, if so, *which node is likely to hold the copy of the line?*. Therefore, the prediction is performed in two steps.

Figure 5.3 illustrates the anatomy of the prediction scheme we propose and evaluate in this work. The proposed scheme consists of two prediction tables. The *first-step* predictor is mainly in charge of answering the first of the two previous questions. That is, it detects those L2 misses that are probably being satisfied with a cache-to-cache transfer. On the other hand, the purpose of the *second-step* predictor is to provide, for those misses which are predicted as 3-hop, a list of the nodes that are supposed to have the single valid copy of the memory line. Additionally, the *No Predict Table* (*NPT*) is required in order to save the addresses of those memory lines for which a miss caused by a store instruction cannot be predicted. This is done to ensure the correctness of the coherence protocol, as discussed in Section 5.4.

5.3.1. First-step Predictor

The first-step predictor is an example of an *instruction-based* predictor [48], that is, this level is indexed using the PC of the instruction causing the miss. We have observed that a few static load/store instructions are responsible for the majority of the 3-hop misses. As shown in Figure 5.3, the *Confidence $-to-$* field, which is implemented as a two-bit saturating counter, along with the information obtained from the *NPT* table are used to make the first prediction. Additionally, a pointer to a node is also included in this level (along with its confidence bits). We have observed that, in some cases, the 3-hop misses caused by a certain instruction almost always receive the memory line from the same node. In these situations, the first-step predictor could be used to make both predictions. Each entry in the first-step table needs ($log_2 N + 4$) bits, for a N-node system.

The first-step predictor is implemented as a non-tagged table and works as follows: initially, all entries in the prediction table have the two saturating counters (*Confidence $-to-$* and *Confidence Pointer 1* fields) initialized to 0. On each L2 cache miss, the predictor

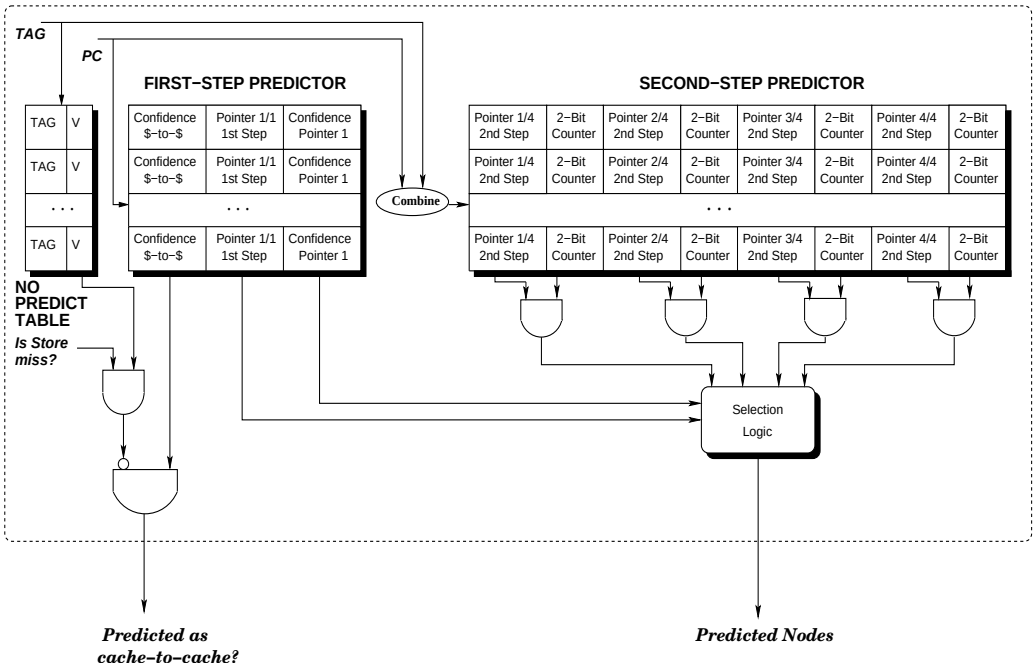


Figure 5.3: Anatomy of the two-step predictor

is probed and, if the *Confidence $-to-$* counter provides confidence (values of 2 or more) the miss is predicted as 3-hop miss. For store instructions, the address of the line must not be contained in the *NPT* table. Later, on the response, predictions are verified, incrementing the *Confidence $-to-$* counter if the miss was served with a cache-to-cache transfer or decrementing it otherwise. In case of a 3-hop miss, the *Confidence Pointer 1* counter is also updated, incrementing it when the value stored by the *Pointer 1/1 1st Step* field agrees with the owner of the line or decrementing it otherwise. When this counter reaches 0, the value of the field *Pointer 1/1 1st Step* is changed to the identifier of the sender of the line.

5.3.2. Second-step Predictor

Although the first-step predictor could be used for making both predictions, a more complex structure is needed in most of cases to correctly answer the second question. The predictor that provides the list of potential holders of the line (second-step predictor) is accessed using both PC and address information. We have observed that a single static instruction causes 3-hop misses for different memory lines, held by different owners. Therefore, the combination of the PC of the load/store with the effective address provides more accurate information as well as reduces interference. As shown in Figure 5.3, each

entry in the second-step predictor consists of four pointers, each one with its corresponding confidence bits (2-bit up-down saturating counters), to store the identity of up to four potential owners for a certain memory line. Thus, $(4 \times log_2 N + 8)$ bits per entry are needed in this case.

The prediction table is also implemented as a non-tagged table and works as follows: initially, all entries in the second-step table store value 0 into the 2-bit saturating counters. On each L2 cache miss predicted as 3-hop by the first-step predictor, the second-step one is accessed and *predicted transfer requests* are sent to the nodes indicated by those pointers whose *2-Bit Counter* fields provide confidence (values of 2 or 3). Additionally, a transfer request is sent to the node indicated by the *Pointer 1/1 1st Step* field whenever this node is not one of those already included (of course, if its confidence value is 2 or more). In this way, when all the *2-Bit Counter* fields in the corresponding entry of the second-step predictor store 0 or 1, the first-step predictor makes both predictions (when the *Confidence Pointer 1* field gives confidence) or the miss is not predicted and is sent to the home directory as would be done in the normal case. On the responses, predictions are verified and the second-step predictor is updated. If the identity of the owner is found in one of the four pointers, the corresponding saturating counter is incremented. Otherwise, its identifier is added using one of the pointers that are unused (if any) and setting the corresponding saturating counter to 1. However, if all the pointers are being used, all the *2-Bit Counter* fields are decremented and the pointer associated with one of the counters that becomes 0 (if any) is used.

5.3.3. Implementation Issues

An important design decision is the maximum number of nodes per prediction. Too few nodes per prediction would cause the second-step predictor to frequently miss for some memory lines (those that are written by several nodes without a defined pattern). However, an excessive number of nodes wastes network bandwidth and could introduce a significant overhead in the directories as well as in the cache controllers. In our case, we have found that a maximum of five nodes per prediction constitutes a good compromise. One node is obtained from the corresponding first-step predictor entry, while the rest are provided by the second-step predictor.

Detecting when a miss is going to be served with a cache-to-cache transfer as well as correctly predicting the node that holds the valid copy of the line, causes the miss to be

forwarded directly to the owner cache, avoiding the subsequent network hops towards the home node, directory controller occupancies and slow DRAM directory access. However, a wrong prediction (either by incorrectly predicting a miss as 3-hop miss or by failing the owner cache for a 3-hop miss) has negative consequences: additional network latency, in some cases, and increased contention in the network, in the directory and in remote L2 caches.

Finally, it is important to note that, contrary to the uniprocessor/serial-program context where predictors are updated and probed continuously with every dynamic instruction instance, we only update the prediction history and only probe the predictor to retrieve information in the case of an L2 miss. Thus, as in [48], we believe that the predictors neither constitute a potential bottleneck nor add cycles to the critical path, because their latency can be hidden from the critical path (since predictions are needed for events with significant latency anyway).

5.4. Coherence Protocol Supporting Prediction

Some modifications must be included into the coherence protocol to make use of the prediction scheme previously presented. Our starting point is the invalidation-based, four-state MESI coherence protocol implemented by RSIM. This coherence protocol is similar to that included in the SGI Origin 2000 [54]. Appendix A describes the coherence protocol found in RSIM as well as it enlightens the main differences between this protocol and the one implemented by the SGI Origin 2000. Two main premises guided our design decisions: first, to keep the resulting coherence protocol as close as possible to the original one, avoiding additional race conditions, and second, to assume sequential consistency [40]. As in [23], we use the following terminology for a given memory line:

- The *directory node* or *home node* is the node to whose main memory the block is allocated.
- The *exclusive node* or *owner node* is the node that holds the single valid copy of the line.
- The *requesting node* is the node containing the L2 cache that issues a miss for the line.

5.4.1. Extended Coherence Protocol Operation

When an L2 miss for a certain memory line occurs, the predictor implemented into the cache controller of the requesting node is accessed. If an entry is not found or the predictor is not confident, the miss is sent to the directory node, where it is satisfied as usual. Otherwise, the miss is predicted to be satisfied with a cache-to-cache transfer and the following actions are undertaken:

Requesting Node Operation. Sends transfer requests to the nodes (or node) predicted to have the valid copy of the line (exclusive node). Each message includes a bit identifying the transfer request as *predicted*. In order to verify the prediction, the miss is also sent to the corresponding home directory (this message includes a list of the nodes that were predicted).

Exclusive Node Operation. When a *predicted* transfer request for a certain memory line comes to the cache controller, the line is searched in the L2 cache. If the line is not in the *Exclusive* or the *Modified* state, a negative acknowledgment message (*NACK*) is sent to the directory node notifying that the *predicted* request cannot be satisfied by this node as well as the identity of the requesting node. Otherwise, as would occur in a regular (not predicted) cache-to-cache transfer miss, the exclusive node immediately sends a copy of the line to the requesting processor as well as an acknowledgment message (*ACK*) indicating this to the directory node (which includes a valid copy of the line for load misses) and properly updates the state of its local copy of the line. Note that for the second case, a *3-hop miss* would be converted into a new kind of *2-hop miss*.

Directory Node Operation. The home directory is responsible for checking the prediction when the message from the requesting node is received, as well as for collecting all the responses from the predicted nodes (*ACK* or *NACK* messages) of a certain prediction.

When a *predicted* request is received, the directory must perform two tasks. First, it checks whether both the miss is 3-hop or not and if so, whether the owner of the line was predicted or not. If the directory finds that the miss is 3-hop but the requesting node failed to include the owner of the line, it immediately sends an additional transfer request to the corresponding owner[2]. In this way, the degradation suffered when the correct owner is not predicted is minimal. Second, a buffer entry is allocated[3] and the number of expected

[2]This is only done for those 3-hop misses that were caused by load instructions. For those that are a consequence of store instructions, we have observed that sending the transfer request to the node indicated by the directory frequently misses, since other predicted misses are usually in progress and one of them hit.

[3]If a buffer entry is not available a retry message is returned to the sender.

responses (that is, the total number of transfer requests sent) is saved. This includes the additional transfer request sent by the directory node (when applicable). On every response, this number is decreased. Once all the responses have been received, one of the following situations can take place:

1. If only *NACK* responses have been received, three scenarios can be distinguished:

   1.1 The memory line is not in the *Private* state. This means that the first-step predictor failed when it identified the miss as 3-hop. In this case, the request is converted into *not predicted* and is processed as it would be in the normal case.

   1.2 The memory line is in the *Private* state but a transfer request was not sent to the current owner node. This takes place, for example, when the directory do not send the additional transfer order to the owner node despite the prediction failed to include it. As done before, the request is converted into *not predicted* and processed as usual.

   1.3 The memory line is in the *Private* state and the transfer order was sent to the exclusive node. The *NACK* from the owner indicates that either a *replacement* for the memory line was performed, or that another transfer order was previously received and processed. In this case, the directory node must wait for the replacement notice or *ACK* message from the owner. This race condition is already considered and solved by the original coherence protocol.

2. In the case of an *ACK* being received, there are two possible scenarios:

   2.1 If the *ACK* comes from the expected node that is, the one codified by the sharing code associated with the memory line, the state and the sharing code must be updated immediately. In those cases where the memory line has a pending access, the *predicted* request must be processed before that access, since the exclusive node has already served the miss. Note that the case in which the line has a pending access when the *ACK* is processed is equivalent to having a pending request for a memory line when a replacement message for the line is received from the single cache holding the line. This race condition is already considered in the original protocol, so additional changes are not needed to support this case.

   2.2 When the *ACK* comes from a different node to the one provided by the sharing code associated with the memory line, it means that something that precedes

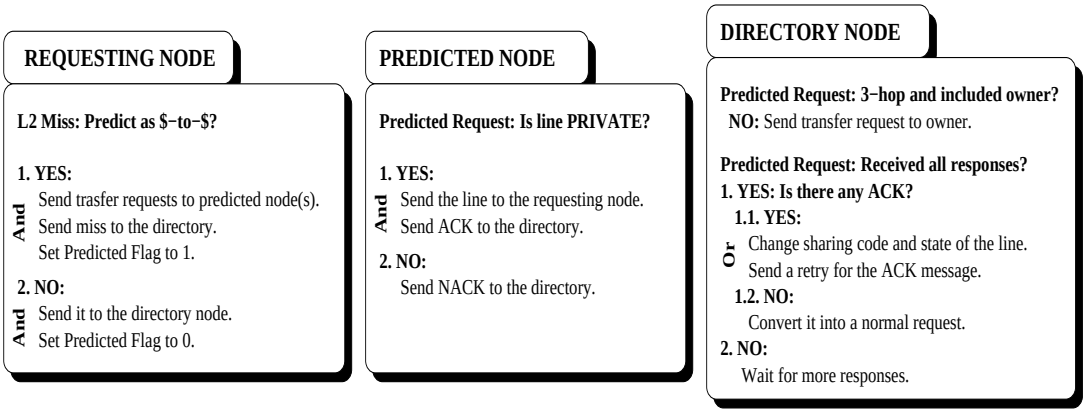


Figure 5.4: How prediction is included into the original coherence protocol

this *ACK* took place. Therefore, a message is sent to the source of the *ACK* informing that the *ACK* could not be observed at that time and that the message must be re-sent. The use of retries avoids deadlocks since it ensures that the message from the expected node can find an entry in the directory buffers.

It is important to note that only one *ACK* message can be received because for cache-to-cache transfer misses there is a single node caching the line. Another interesting point is that race conditions for lines in the *Private* state are now solved by the owner cache of the line (not by the directory).

Figure 5.4 summarizes the previous coherence protocol extended with prediction. As it can be observed, prediction could be included in an existing coherence protocol with minimal changes.

5.4.2. Dealing with Race Conditions: the *NPT* Table

There is an additional situation that must be considered in order to preserve the correctness of the coherence protocol. Note that our coherence protocol is based on the fact that at every moment the directory can find the precedence order between all the events related to a memory line in the *Private* state (*predicted* and *non-predicted* requests) as actually happens. This is possible since the directory always knows where the single valid copy of the line can be found and only a message from such a node for the line will be processed. However, when write-write sharing [23] takes place, prediction can cause such a precedence order to get lost. As an example, assume the next scenario: initially, node *A* has the single copy of a memory line *M*. Then, node *B* wants to write to the line *M*. Obviously an

L2 cache miss occurs, and node *B* predicts node *A* as having the single copy of the line. When node *A* sees the *predicted* request, it sends a copy of *M* to node *B*, invalidates its local copy and sends the *ACK* message to the home directory. Later on, the processor in node *A* attempts to write to the memory line *M* and again, a miss occurs. Assuming node *A* predicts node *B* as having the line in the *Private* state, the same event sequence takes place. Finally, another node, for example node *C*, has a write miss for line *M* and predicts node *A* as having the single copy of the line. The problem arises when the home directory sees the second *ACK* from node *A* before the very first one. In this situation, the directory assumes the line *M* to have been moved first from *A* to *C*, so that the desired ordering is lost. This is possible due to: *i)* we assume a point-to-point network without ordering properties, and, *ii)* even with such an ordered point-to-point network, the problem still occurs due to the use of retry messages. Finally, note that if the *ACK* from node *B* arrived before the first *ACK* from node *A*, a retry message would be returned to *B*.

To avoid the problem, each time a cache controller receives a *predicted* request that hits in the local L2 cache and was caused by a store instruction, it looks for a free entry in the No Predict Table described in the previous section. If the *NPT* table were full, the request could not be handled as *predicted* and the exclusive node would force a replacement for the line, after which it would send the *NACK* to the directory node. Otherwise, the tag is included into the *NPT* table of the exclusive node. If the next miss suffered by this node for the memory line is caused by a store instruction it will not be predicted, forcing the miss to go through the directory to ensure the precedence order. In any case, the miss frees its entry for the memory line in the *NPT* table. Observe also that the use of the *NPT* tables avoids that several predicted accesses prevent indefinitely a non-predict access from obtaining ownership of the line (that is, livelock situations).

5.4.3. Directory Requirements

An important issue that must be also considered is the features that a particular scheme requires from the directory. In particular, it is important to define the sharing codes that the directory could provide. The technique presented in this chapter could be used in conjunction with compressed and multi-level directories (as well as with exact sharing codes as *full-map*). There is just a requirement that the selected compressed sharing code must satisfy: it must be large enough to exactly codify the identity of the owner of a memory line. The modifications performed to the original coherence protocol are based

on the assumption that at every moment the directory knows the identity of the node from which it must receive the acknowledgment (*ACK*) for a particular 3-hop miss. This way, the directory architecture presented in the last chapter, which employs *BT-SuT* as the sharing code for the third-level directory, fulfills this requirement.

5.5. Performance Evaluation

In this section, we present a detailed evaluation of our proposal using extensive execution-driven simulations of a 16-node cc-NUMA multiprocessor[4]. From the set of applications available, we select those for which more than 16% of the L2 cache misses are 3-hop misses. Using these applications, we compare the base system (*Base*) with two configurations which use two instances of the prediction scheme previously presented. For the first of these configurations, referred to as *UL-2Step*, the first and second-step prediction tables as well as the *NPT* table have an unlimited number of entries. This predictor has a high cost, but it completely eliminates the aliasing effect caused by the non-tagged nature of the prediction tables.

The second configuration, called *L-2Step*, implements a "realistic" version of the former predictor. In this case, the first-step prediction table has a total of 2K entries (total size of 2 KB) whereas 16K entries are available in the second-step one (total size of 48 KB). The *NPT* table included in this configuration has a total of 128 entries (size of 512 bytes), which is enough since we have observed that in most cases, a small number of entries are needed for this table. The first-step table is directly indexed using the ten least significant bits of the PC of the instruction missing in the L2 cache. Access to the second-step table is carried out from the result of computing the XOR between bits from 5 to 18 of the missing address and bits from 2 to 15 of the PC. As in [29], we use XOR-based placement to optimize the use of the entries in the second-step table. Note that both prediction tables are non-tagged so aliasing can still occur. Finally, due to its small number of entries, the *NPT* is organized as a totally associative buffer structure.

Additionally, we also show the performance benefits that would be obtained if an almost-oracle predictor were available at each one of the nodes of the multiprocessor. This configuration, which is called the *AOP* system, gives an approximation of the potential of our technique. The *AOP* predictor used in this work accesses directory information on ev-

[4]In this case, we reduce the number of system nodes to achieve lower simulation times. We have found that similar results are obtained for larger configurations.

ery L2 cache miss to determine whether the memory line is in the *Private* state or not, and, if so, the identity of the owner of the line, and directly sends the miss to the corresponding node. We have modified RSIM to allow nodes suffering an L2 cache miss to directly access the corresponding directory entry without spending any cycle. However, this is an approximation of the oracle predictor behavior since mispredictions are still possible. For example, when two different nodes make a prediction at the same time for the same memory line, one of them will miss. However, these situations occur infrequently so that more than 90% of the predictions were correct for all the applications. Moreover, since the *AOP* predictor sends a single message per prediction, misprediction penalty is always kept very small.

In all cases, we assume that a full-map single-level directory is used in each one of the nodes of the multiprocessor. In this way we focus on the impact that our technique has on application performance, avoiding the effects of having a compressed directory level.

5.5.1. Predictor Accuracy

The main objective of the prediction-based technique presented in this chapter is to directly send those L2 cache misses that are going to be served with a cache-to-cache transfer to the owner of the line. Therefore, two predictions must be carried out: whether or not a certain cache miss is a 3-hop miss and if so, the identity of the node owning the line. The *two-step* prediction scheme proposed in Section 5.3 uses the history stored in the first-step table to detect 3-hop misses, whereas the location of the valid copy of the memory line is determined using both the first and second-step tables. This section analyzes the accuracy results that are obtained for the predictors included in both the *UL-2Step* and *L-2Step* configurations. First, we study the accuracy of the first and second-step predictors individually to finally show the fraction of the 3-hop misses that are successfully predicted.

Figure 5.5 illustrates the accuracy of the *first-step predictor*. Over the total number of L2 misses, it shows the percentage of misses that were correctly identified as 3-hop or non-3-hop (*Hit*), the percentage of misses that were seen as 3-hop misses but were not (*Miss True*) and the percentage of misses that were incorrectly predicted as non-3-hop misses (*Miss False)*. As observed, when the *UL-2Step* configuration is used, hit rates of almost 100% are obtained for all applications but BARNES and EM3D, which demonstrates the high accuracy of this predictor. The hit rates obtained for BARNES and EM3D are slightly

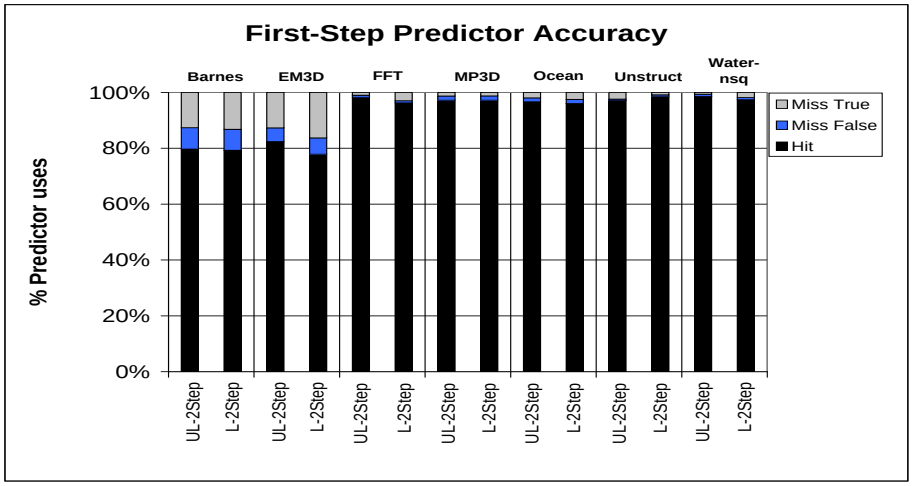


Figure 5.5: First-step predictor accuracy

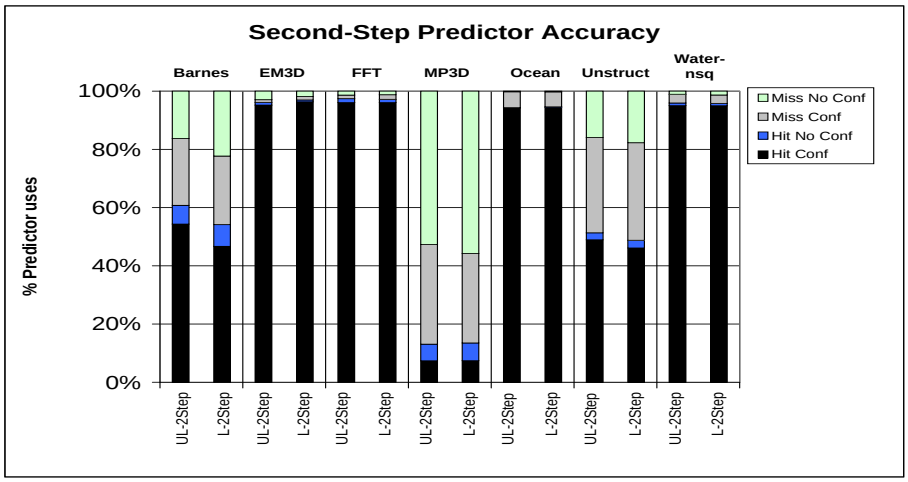


Figure 5.6: Second-step predictor accuracy

lower (80% in each case). As seen, for these applications most of the prediction misses fall into the *Miss True* category (16% approximately in each case), that is, they are non-3-hop misses that are probed as being 3-hop.

The accuracy of the *second-step predictor* is presented in Figure 5.6. In this case, over the number of accesses to this predictor, it shows the percentage of prediction hits (*Hit Conf*), prediction misses (*Miss Conf*), hits that were not predicted since the 2-bit counters used at this level did not give confidence to the prediction (*Hit No Conf*) and misses that were saved since the 2-bit counters did not allow the prediction (*Miss No Conf*). As can be seen, hit rates of more than 50% are obtained for all the applications but MP3D when the

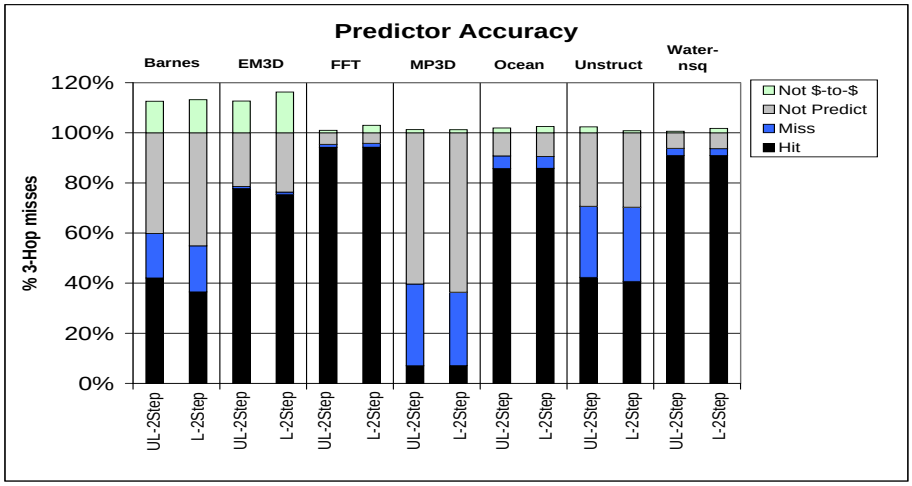


Figure 5.7: Percentage of 3-hop misses predicted

UL-2Step configuration is used. For this application, we have observed that the majority of cache-to-cache transfers occur for a small number of lines that are accessed by all the nodes, which prevents the second-step predictor from making any predictions.

Finally, over the total number of 3-hop misses, Figure 5.7 shows the percentage of those that have been correctly predicted (*Hit*), those for which the second-step predictor missed the correct owner (*Miss*), those that could not be predicted due to the first-step predictor or the second-step (or both), not assigning confidence to the prediction (*Not Predict*) and those that were probed as 3-hop misses but were not (*Not $-to-$*). The latter is shown starting from 100%, since it corresponds to misses that are not 3-hop misses, and these percentages are computed over the total number of misses. As can be seen, a high percentage of the 3-hop misses can be successfully predicted for EM3D, FFT, OCEAN and WATER-NSQ applications (more than 80%) when the *UL-2Step* configuration is used. The irregular behavior observed in MP3D prevents the second-step predictor from predicting more than 60% of the 3-hop misses, although the first-step predictor successfully identifies them as being served with a cache-to-cache transfer. For this application, the use of confidence bits reduces the number of mispredictions and then, saves certain misses from wasting bandwidth. Although in less extension, this is also the situation found in BARNES. Finally, for UNSTRUCTURED we have found that an important number of 3-hop misses (13% for the *UL-2Step* system) are not predicted due to the high number of store misses for which an entry in the *NPT* was found. The reason is the significant amount of false sharing observed in this application. Note also that for all the applications the number of 3-hop misses that

could not be predicted (*Not Predict*) exceeds those that were incorrectly predicted (*Miss*). Also, it is important to note that the percentage of misses incorrectly predicted as 3-hop (*Not $-to-$*) is very low, which demonstrates the high accuracy of the first-step predictor. Finally, observe that the realistic predictor included in the *L-2Step* configuration obtains almost the same accuracy results than those reported for the *UL-2Step* case.

5.5.2. Performance Analysis

In this section we analyze quantitatively the performance benefits of our proposal. First, we study how prediction affects the average latency of 3-hop misses, then the effect on the average latency of load and store misses is presented, and finally, reductions on execution times are also reported. In every case, we compare a *base* configuration, which does not use prediction, and the two configurations which include the two-step predictor (*UL-2Step* and *L-2Step*), and whose accuracy was presented in the last section. Additionally, the results obtained with the almost-oracle predictor (*AOP*), which gives us an approximation of the maximum benefit that could be obtained, are also presented in some cases.

Application	AOP	UL-2Step	L-2Step
BARNES-HUT	1.00	1.48	1.43
EM3D	1.00	1.09	1.18
FFT	1.00	1.00	1.24
MP3D	1.00	1.30	1.30
OCEAN	1.00	1.02	1.07
UNSTRUCTURED	1.00	2.42	2.30
WATER-NSQ	1.00	1.11	1.07

Table 5.1: Number of nodes included per prediction

Figure 5.8 presents the normalized average latency for 3-hop misses split into network latency, directory latency and miscellaneous latency (buses, cache accesses...), for the *base*, *AOP*, *UL-2Step* and *L-2Step* configurations. Network latency comprises the number of cycles the 3-hop miss spends on the interconnection network. Directory latency represents the cycles needed by the directory to satisfy the miss. Normalized average latencies are computed dividing the average latencies for each one of the configurations by the average latencies for the base case. Table 5.1 shows the average number of nodes included in each prediction.

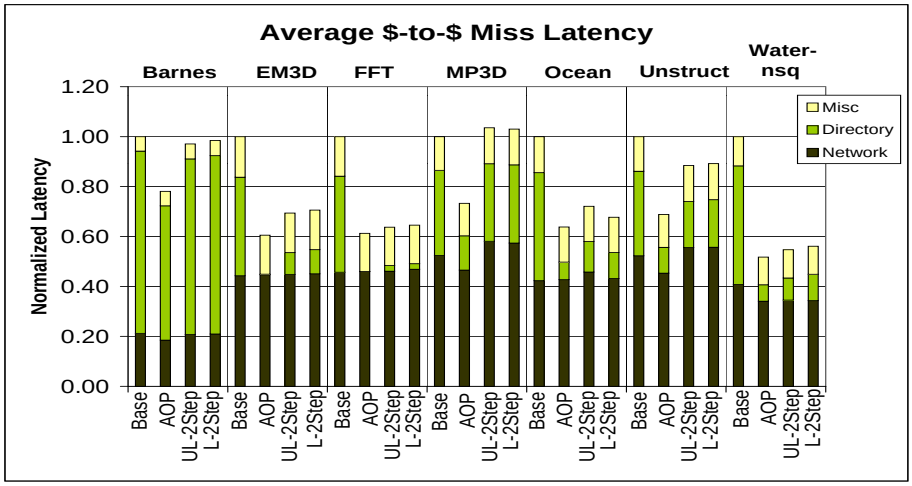


Figure 5.8: Normalized average 3-hop miss latency

As can be observed from the results obtained for the *AOP* configuration, prediction has the potential to significantly reduce the latency of 3-hop misses for all the applications (from 22% for BARNES to 48% for WATER-NSQ). In practice, the reductions that could be reached with a non-oracle predictor are lower, since not all the 3-hop misses can be correctly predicted and for some predictions, several transfer requests must be sent. However, these benefits are still very important for all the applications but BARNES and MP3D when the *UL-2Step* system is used (latency reductions ranging from 12% for UNSTRUCTURED to 45% for WATER-NSQ are obtained), and, more importantly, they could be obtained with the realistic implementation provided by the *L-2Level* configuration. Note, also, that these benefits are a consequence of a significant reduction on the directory component of the latency. As seen in the last section, the irregular patterns found in BARNES and MP3D prevents the two-step predictor from successfully predicting the majority of 3-hop misses. This translates into insignificant performance benefits for BARNES (latency reductions of 3%) and a small performance degradation for MP3D (latency degradation of 4%). Finally, the non-tagged nature of the *L-2Step* predictor makes some of its entries be shared between different predictions. In general, this slightly increases the average number of transfer requests sent per prediction as well as the average latency of 3-hop misses when compared to the *UL-2Step* scheme (see Table 5.1 and Figure 5.8). Only for OCEAN, the untagged nature of the predictor used by the *L-2Step* system has positive consequences on the average latency of 3-hop misses (reductions of 28% and 32% for the *UL-2Step* and *L-2Step* systems, respectively).

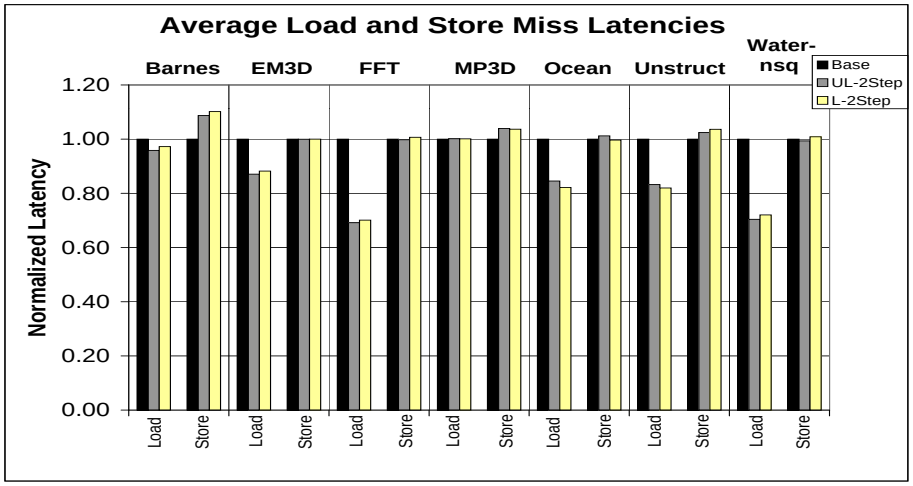


Figure 5.9: Normalized average load/store miss latency

Application	Load misses		Store misses		Total 3-hop
	3-hop misses	*Rest load misses*	*3-hop misses*	*Rest store misses*	
BARNES-HUT	28.18%	30.78%	12.79%	28.25%	40.98%
EM3D	26.92%	46.16%	0.00%	26.92%	26.92%
FFT	39.82%	18.73%	0.35%	41.10%	40.17%
MP3D	19.23%	5.52%	55.95%	19.30%	75.18%
OCEAN	15.46%	29.86%	0.44%	54.24%	15.90%
UNSTRUCTURED	29.77%	7.33%	33.13%	29.77%	62.90%
WATER-NSQ	34.34%	29.32%	3.00%	33.34%	37.34%

Table 5.2: Fraction of load and store misses that are 3-hop, and total percentage of 3-hop misses found in each application

The important benefits found for 3-hop misses also lead to reductions in the average latency of load and store instructions. As can be observed from Table 5.2, a significant fraction of the load misses is served with a cache-to-cache transfer for all applications which, as shown in Figure 5.9, translates into significant reductions on load miss latencies when the *UL-2Step* system is used for EM3D (13%), FFT (31%), OCEAN (15%), UNSTRUCTURED (17%) and WATER-NSQ (31%). Again, these reductions can be obtained with the *L-2Step* configuration. On the other hand, the benefits found for store misses are not so significant and even small slow-downs are observed in some cases. These results can be expected for EM3D, FFT, OCEAN and WATER-NSQ due to, as illustrated in Table 5.2, a very small percentage of the 3-hop misses was caused by store misses (0% in some cases). On the contrary, the fraction of store misses served with a cache-to-cache transfer is substantial for BARNES, MP3D and UNSTRUCTURED. However, we have found that most of

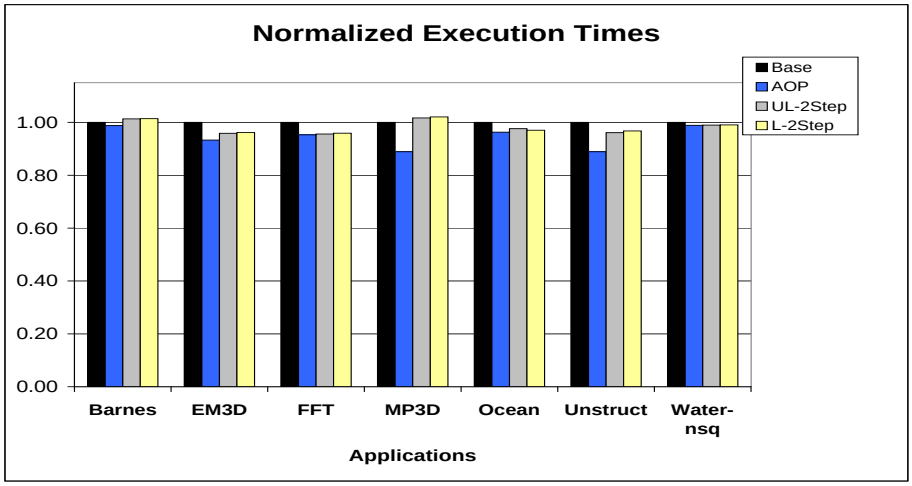


Figure 5.10: Normalized execution times

the mispredictions suffered in BARNES, MP3D and UNSTRUCTURED are for store misses, which is the reason of the degradations shown in Figure 5.9 for these applications. For UNSTRUCTURED, besides, an important fraction of the store misses found an entry in the *NPT* table and consequently, could not be predicted. Finally, while load misses can be significantly accelerated using the proposed prediction scheme, store misses cannot take much advantage of it, due to the fact that either cache-to-cache transfers are rarely used to serve store misses, finding the correct owner on a store miss is hard, or to write-write sharing precludes prediction from being used for most of the store misses. Therefore, two-step predictors could be simplified (eliminating the need of having the *NPT* tables) without significantly hurting the final performance by not predicting store misses.

The ultimate metric for application performance is the execution time. Figure 5.10 shows the execution times for the *AOP*, *UL-2Step* and *L-2Step* configurations normalized with respect to the base system. We find that the degradations observed for the store misses in BARNES and MP3D finally translate into small slowdowns on application execution times (1% for BARNES and 2% for MP3D). On the other hand, the reductions in terms of execution time obtained when both the *UL-2Step* and the *L-2Step* configurations are used are very close to those obtained in the *AOP* case for EM3D (4%), FFT (4%), OCEAN (3%) and WATER-NSQ (1%). Whereas reductions on execution time of 11% could be obtained for UNSTRUCTURED, in practice they are smaller when the *UL-2Step* and *L-2Step* configurations are used (4%).

5.6. Conclusions

Several recent studies have observed that cache-to-cache transfer misses constitute an important fraction of the total miss rate (more than 60% in some cases), so that optimizations to reduce the usually long latencies associated with these misses have become the subject of important research efforts. In this chapter, we propose the use of prediction to directly send 3-hop misses to the corresponding node where the single valid copy of the line resides. This would eliminate the significant number of cycles needed to access the directory information from the critical path of 3-hop misses.

The prediction-based technique proposed in this chapter consists of two main components. The first one is a *two-step prediction scheme* that includes: a first-step predictor, which is indexed using the PC of the instruction causing the miss and is used to detect whether a certain L2 cache miss is going to be served with a cache-to-cache transfer or not, and a second-step predictor, which is accessed using both the PC and the address of the misses, to provide a list of the potential owners for those misses identified as 3-hop. The second component is a four-state MESI coherence protocol, similar to that used in the SGI Origin 2000, properly extended (with minimal changes) to support the use of prediction.

In order to evaluate the effects of our proposal, we executed several shared memory applications on top of the RSIM detailed execution-driven simulator. First, we analyzed the prediction accuracy of the proposed two-step predictor. For this, we individually studied the behavior of each one of the predictors and found that, for the majority of the applications, the first-step predictor could obtain very high hit rates. Correctly predicting the owner node, however, seemed to entail more problems for some applications. Finally, we showed that for all applications except MP3D, more than 40% of the 3-hop misses could be correctly predicted. The irregular behavior observed for MP3D prevented this application from taking advantage of the proposed prediction scheme.

Then, we analyzed our proposal in terms of its impact on application performance. We found that our proposal can significantly reduce the latency of 3-hop misses (reductions on average latency ranging from 12% for UNSTRUCTURED to 45% for FFT). These benefits translated into important reductions on the average latency of load misses, but did not have a positive impact on store misses. Finally, application execution times were accelerated up to a maximum of 4% for several applications. Additionally, we showed that these results can be obtained using a predictor with a total size of less than 64 KB.

Additional optimizations for 3-hop misses could be derived from the results presented

in this chapter. For example, the high hit rates observed for the first-level predictor suggest that small predictors could be introduced in every node to avoid, when detecting a 3-hop miss, the speculative read of memory that, otherwise, would occur in parallel to the access to the directory (which wastes memory bandwidth).

Chapter 6

Sharers Prediction for Accelerating Upgrade Misses

6.1. Introduction

In Chapter 4 we presented a taxonomy of the L2 misses according to the actions performed by directories to satisfy them. We showed that for two of the four categories of the taxonomy, *Mem* and *Inv+Mem* misses, the directory must provide the memory line. For misses falling into the other two, *$-to-$* and *Inv* misses, the directory is accessed to find the identity of the nodes (or node) that must receive the corresponding coherence transactions (transfer requests and invalidations, respectively). In Chapter 5 we showed that prediction can be applied to accelerate cache-to-cache transfer misses by directly sending the corresponding transfer request from the requesting node to the current owner. In parallel, and in order to check the prediction, the miss was also sent to the directory. In this way, in case of prediction hits, accesses to the directory were removed from the critical path of the misses. Now, in this chapter, we focus on accelerating *Inv* misses (also known as upgrade misses) by increasing the parallelism between the actions performed to satisfy them.

Upgrade misses are caused by store instructions which find read-only copies of the lines in the local L2 caches. For this kind of misses, the L2 caches already have the valid data and only need exclusive ownerships. In this way, the directory must invalidate all copies of the memory lines except those held by the requesting processors. To do that, the directory must first determine the identity of the sharers by accessing directory information. Then, invalidation messages are created and sent. Only when the directory has received all the replies to the invalidation messages, the requesting node is given the ownership of the line.

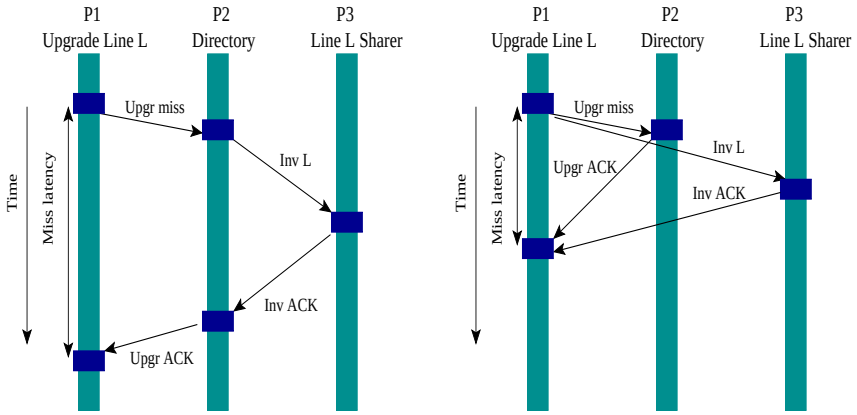


Figure 6.1: Coherence protocol operations for an upgrade miss in a conventional cc-NUMA (left) and in a cc-NUMA including prediction (right)

This scenario is illustrated in Figure 6.1, left. In Chapter 4, we proposed an architecture designed especially to reduce long L2 miss latencies that characterize cc-NUMA designs by means of integrating some key components into the processor die. In general, finding directory information in a low-latency on-chip directory cache reduces the number of cycles required to find the identity of the current sharers of the memory line. Unfortunately, in most cases, sending invalidation messages to the sharers and receiving the corresponding responses (which can not take advantage of finding directory information in the fast directory cache) consumes most of the cycles needed by upgrade misses.

In this chapter, we present and evaluate *sharers prediction* [5], a technique that reduces the latency of upgrade misses by invalidating the sharers in parallel with the directory lookup. As shown in Figure 6.1, right, upgrade misses could be significantly accelerated if instead of going through the directory, sharers were directly invalidated from the requesting node. This way, on an upgrade miss, the faulting node would predict the current sharers for the line and would send invalidation messages directly to them. In order to check the prediction, the upgrade miss is also sent to the directory. However, different to a conventional coherence protocol, now the directory is accessed in parallel with the invalidation process. If all the sharers are correctly predicted (as is the case presented in Figure 6.1), the directory immediately gives ownership of the line. On the contrary, if any of the actual sharers is not included in the list of predicted sharers, the directory invalidates it before replying to the requesting node. This node assumes the ownership of the line once it has received all the responses from the predicted nodes as well as the acknowledgment message from the directory. It is important to note that, correctly predicting the sharers of

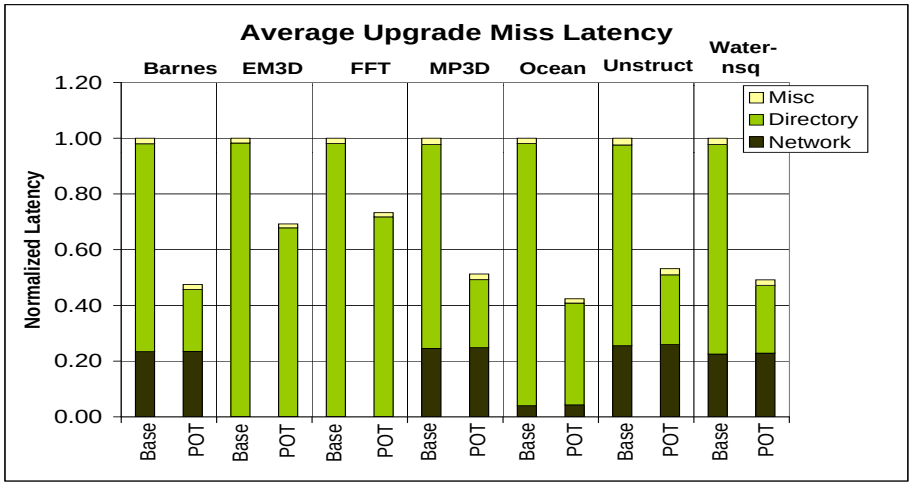


Figure 6.2: Normalized average *upgrade miss* latency

a certain memory line would cause access to the directory to be performed in parallel with the invalidation of the sharers, significantly reducing the number of cycles employed by directories to satisfy upgrade misses. On the other hand, when prediction fails, the miss will be satisfied as usual and its latency could be slightly affected as a result of the increased network and cache controller occupancies to create, send and process invalidation messages and their associated responses.

Our proposal is based on the observation that upgrade misses present a repetitive behavior: the nodes that are invalidated on an upgrade miss for a certain memory line will be invalidated again on future upgrade misses for the same line. Also, as previously pointed out [23, 32], the number of sharers that must be invalidated on an upgrade miss is very small (one, in some cases), which reveals that a more efficient management of this kind of misses could be effective. Therefore, a well-tuned prediction engine could be employed to capture the repetitive behavior of upgrade misses, significantly accelerating them by removing the access to the directory from the critical path and thus, improving the performance of cc-NUMA architectures. Figure 6.2 shows a comparison between the average upgrade miss latencies that are obtained for the base system and an approximation to the ones that, in the best case, are reached when the sharers prediction is used, that is, when all upgrade misses are correctly predicted (*POT*). Details on how this configuration is implemented are given in Section 6.5. Results are shown for the applications used in Chapter 5. As can be observed, sharers prediction has the potential to significantly reduce the latency of upgrade misses (reductions ranging from 27% for FFT to 58% for OCEAN), and

as can be seen, these benefits are a consequence of a significant reduction on the directory component of the latency.

Similarly to owner prediction, the technique proposed in this chapter entails the development of two elements: first, an effective prediction engine able to find out the identity of current sharers in cases of an upgrade miss and second, a coherence protocol designed to support the use of prediction. The proposed prediction scheme is an *address-based* predictor, accessed on an upgrade miss to obtain the identity of up to three potential sharers. We show how a predictor with a total size of less than 48 KB is able to obtain very high hit rates in most cases. Regarding the new coherence protocol, the main goal is to provide prediction by introducing the minimum number of changes into an already existing coherence protocol. In particular, our proposal extends the four-state MESI coherence protocol implemented by RSIM.

The rest of the chapter is organized as follows. Related work is given in Section 6.2. Section 6.3 shows the prediction scheme that we propose. Extended coherence protocol is presented and justified in Section 6.4. Section 6.5 shows a detailed performance evaluation of our proposal. Finally, Section 6.6 concludes the chapter.

6.2. Related Work

Snooping cache coherence protocols optimize upgrade misses (sharing misses, in general) by directly performing the invalidation of the sharers from the requesting node. Systems using snooping –usually known as symmetric multiprocessors or SMPs– broadcast upgrade misses to all the processors and memory over a network with a completely ordered message delivery (traditionally a bus). Upon receiving the *BusUpgr* transaction, each sharer invalidates its local copy of the memory line. However, as these designs rely on broadcasting all the coherence transactions on a bus, the number of processors that can be incorporated is restricted and some proposals trying to extend their scalability have appeared (as [13, 61, 62], which were commented on in the previous chapter).

Several works, as [24, 60, 93], have tried to reduce the latency of upgrade misses by using a multicast scheme to perform the invalidation of the sharers. Instead of sending individual invalidation messages to each one of the sharers, the directory could send a single multidestination invalidation message. Additionally, the number of acknowledgments is also reduced by combining some of them at the intermediate nodes on their way back to the home directory. However, as we will see, the number of sharers that must be invali-

dated on an upgrade miss is small [23, 32], precluding multicast schemes from obtaining any significant advantage.

Similarly to the case of the 3-hop misses, the hierarchical nature of the design of the Compaq AlphaServer GS320 [28] and its limited scale (it has an architecture specifically targeted at medium-scale multiprocessing) make it feasible to use simple interconnects, such as a crossbar switch, to connect the handful of nodes, allowing a more efficient handling of upgrade misses than traditional directory-based multiprocessors by exploiting the extra ordering properties of the switch to eliminate explicit invalidation acknowledgments. On the contrary, our proposal does not require any interconnection network with special ordering.

Another related technique to reduce coherence overhead resulting from the invalidation of shared lines was originally presented in [55]. Some heuristics are applied to allow each processor to detect and *self-invalidate* those shared lines that will probably not be accessed in the future. Subsequently, Lai and Falsafi [52] proposed Last-Touch Predictors (LTPs) to improve self-invalidation accuracy. Contrary to our prediction scheme that could be implemented using less than 48 KB, LTPs require much more memory to store their signatures (encoding of program traces used to detect last-touches to every line).

Alternatively, Kaxiras and Goodman [48] and Nilsson and Dahlgren [71] examined a hardware technique to detect and tag loads in the instruction stream that are followed by a store to the same address. The idea was to reduce the L2 miss rate by converting a load miss into a coherent write. Note that, this technique allows saving the load miss (which would be converted into the subsequent store miss) but it does not accelerate upgrade misses at all.

6.3. Predictor Design for Upgrade Misses

Similarly to owner prediction, the first component of the proposal presented here is an effective prediction scheme that allows each node of a cc-NUMA multiprocessor to *guess* the sharers of a certain memory line once an upgrade miss for the line is triggered.

Figure 6.3 illustrates the architecture of the prediction scheme that we propose and evaluate in this work. The scheme consists of a *prediction table* which, on an upgrade miss, provides a list of the nodes that are supposed to have a read-only copy of the memory line and an *Invalidated Lines Table* (*ILT*) used to save the addresses of some of the memory

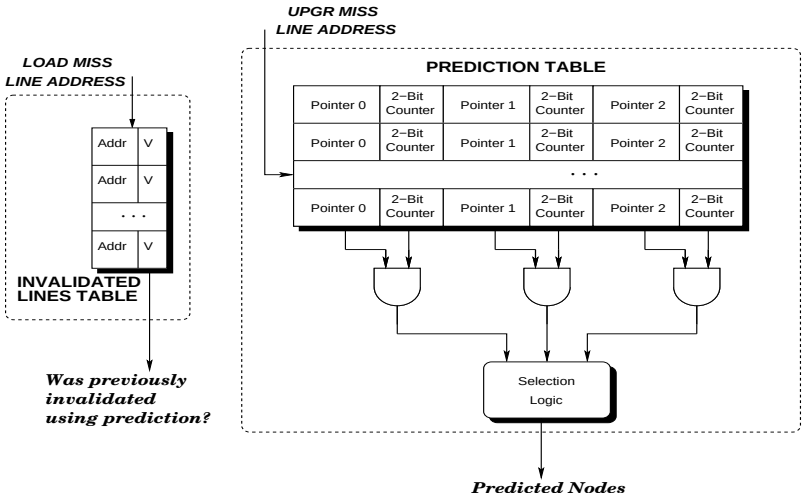


Figure 6.3: Anatomy of the proposed predictor

lines for which a *predicted* invalidation was received. The latter is done to ensure the correctness of the coherence protocol, as discussed in Section 6.4.

The prediction table is indexed using the effective memory address that caused the miss. Hence, this is an example of an *address-based* predictor. As shown in Figure 6.3, each entry in the prediction table consists of a set of pointers (three in our particular case), each one with its corresponding confidence bits (a 2-bit up-down saturating counter). These pointers are used to encode the identity of the sharers for a particular line (a subset of them when more than three sharers are found). In this way, each one of the entries needs $(3 \times log_2 N + 6)$ bits, for a N-node system.

An important design decision is the maximum number of pointers that each entry must provide. Having a small number of pointers could cause all the potential sharers for a line not to be included. On the other hand, having N 2-bit up-down saturating counters per entry, for a N-node system, would increase the width of the predictor and consequently, its final size. Earlier studies have shown that most of the time only a few caches have a copy of a line when it is written [23, 32]. We observed that having three pointers per line is enough to store the identity of all the potential sharers in most cases. Also, it is important to note that small benefits can be expected from applying prediction to lines that are widely shared. Thus, we think that providing three pointers per entry constitutes a good compromise.

The prediction table is implemented as a non-tagged table and works as follows: initially, all the entries store value 0 into the 2-bit saturating counters. On each upgrade miss,

the predictor is probed. The miss is predicted when at least one of the *2-Bit Counter* fields provides confidence (values of 2 or 3). In this case, invalidation messages are sent to the nodes indicated by those pointers whose *2-Bit Counter* fields store values of 2 or 3. On receiving the responses, predictions are verified. The reply message received from the directory for an upgrade miss contains the list of those nodes that actually had a copy of the line. Whereas, the predictor holds a list of three potential sharers. Confidence counters associated with the pointers of those nodes that appear in both lists are incremented. On the other hand, confidence bits of those pointers whose nodes are not present in the list of actual sharers are decremented. Finally, those sharers that do not have an associated pointer are added with a confidence value of 2 as long as free pointers are available (confidence values of 1 or less). An additional optimization is also included as a consequence of the observation made by Gupta and Weber in [32]. The authors found that migratory sharing causes an important percentage of the upgrade misses. Migratory sharing arises when shared data is modified in turn by several processors. One is made aware of this fact by updating the predictor each time the read-only copy of a certain memory line is not received from the corresponding home directory, but instead from the owner of the line through a cache-to-cache transfer. In this way, the owner of the line is considered a potential sharer.

The *ILT* structure is used to store the address of some of the lines for which a predicted invalidation was received and that have not been referenced since then. This information is originally stored into the L2 cache entry when the line is invalidated (by setting the corresponding bit). Only when this cache entry is subsequently assigned to a different memory line (and the corresponding bit is still set), this information is moved to the *ILT* table. Therefore, each entry in the *ILT* table consists of the address of a memory line and a presence bit indicating whether the entry is currently being used or not.

Again, we assume that the predictors neither constitute a potential bottleneck nor add cycles to the critical path, because their latency can be hidden from the critical path (since we only need their predictions on events with significant latency anyway). Note that, contrary to the uniprocessor/serial-program context where predictors are updated and probed continuously with every dynamic instruction instance, we only update the prediction history and only probe the predictor to retrieve information in the case of an upgrade miss.

6.4. Coherence Protocol Supporting Prediction

Additionally, sharers prediction requires to include some modifications into the coherence protocol. Again, our starting point is the invalidation-based, four-state MESI coherence protocol implemented by RSIM and also, the same two main premises guided our design decisions: first, to keep the resulting coherence protocol as close as possible to the original one, avoiding additional race conditions, and second, to assume sequential consistency [40]. As in [23], we use the following terminology for a given memory line:

- The *directory node* or *home node* is the node to whose main memory the block is allocated.
- The *sharing nodes* are those nodes that hold read-only copies of the line.
- The *requesting node* is the node containing the L2 cache that issues an upgrade miss for the line.

6.4.1. Extended Coherence Protocol Operation

When an upgrade miss for a certain memory line occurs, the predictor implemented into the cache controller of the requesting node is accessed. If an entry for the line is not found or if it is not confident, the miss is sent to the directory node, where it is satisfied as usual. Otherwise, the upgrade miss is *predicted* and the following actions are undertaken:
Requesting Node Operation (1). Sends invalidation messages to the nodes predicted to have read-only copies of the line (sharing nodes). Each message includes a bit identifying the invalidation as *predicted*. In order to verify the prediction, the miss is also sent to the corresponding home directory (a list of the predicted nodes is included into the message).
Directory Node Operation. When a *predicted* upgrade miss for a line in the *Shared* state is received, the directory node checks if all the current sharers were predicted and thus, invalidated. If any (or some) of the sharers were not predicted, the directory invalidates it (or them) as for a non-predicted upgrade miss. Once the non-predicted sharers have been invalidated[1] (if any), the directory updates the entry associated with the line (now the requesting node has the ownership of the line) and sends the reply message giving exclusive ownership of the line to the requesting node. If a *predicted* upgrade miss is

[1]The directory has received the corresponding replies to the invalidations from them.

received for a line that is not in the *Shared* state, something preceding the upgrade miss took place. In this case, the miss is converted into non-predicted and processed as usual.
Sharing Node Operation. When a *predicted* invalidation for a certain memory line comes to the L2 cache controller, the line is searched into the L2 cache. One of these situations could occur:

1. The line is not found in the L2 cache. In this case, a *NACK* message is returned to the requesting node informing that the prediction for this node missed. Additionally, if the predicted node has a pending load miss for the line, the invalidation is noted down in the *MSHR* (miss status holding register) associated with the miss (as would be done in the normal case).

2. The line is found to be in the *Exclusive* or *Modified* states. In this case, this node requested a read-write copy of the line and the directory satisfied this miss before the predicted one. Thus, a *NACK* must be returned to the requesting node.

3. The line is found to be in the *Shared* state. This is the case of a prediction hit. There are three possible situations that should be considered:

   3.1 The node does not have a pending upgrade for the line. In this case, the line is immediately invalidated and an *ACK* reply is sent to the requesting node. Besides, the line is marked as having been invalidated by a remote node (not by the directory), by setting a bit (the *Predicted bit*) into the corresponding L2 cache entry (this bit will be cleared in the next miss for that line). Later on, if this L2 cache entry were used to store a different memory line, an entry for the previously invalidated line would be allocated in the *Invalidated Lines Table* (*ILT*) whenever the *Predicted bit* is still set. If all the entries in the *ILT* table are being used at this moment, one of them would be used (a kind of replacement) and a load miss would be forced for the memory line associated with the removed entry. An entry for a certain memory line in the *ILT* is freed as soon as an L2 cache miss for that line appears.

   3.2 The node has a pending upgrade for the line but the reply from the directory has not arrived yet. In this case, the line is invalidated and an *ACK* reply returned.

   3.3 The node has a pending upgrade for the line and the reply from the directory has already arrived. In this case, this node is becoming the new owner of the

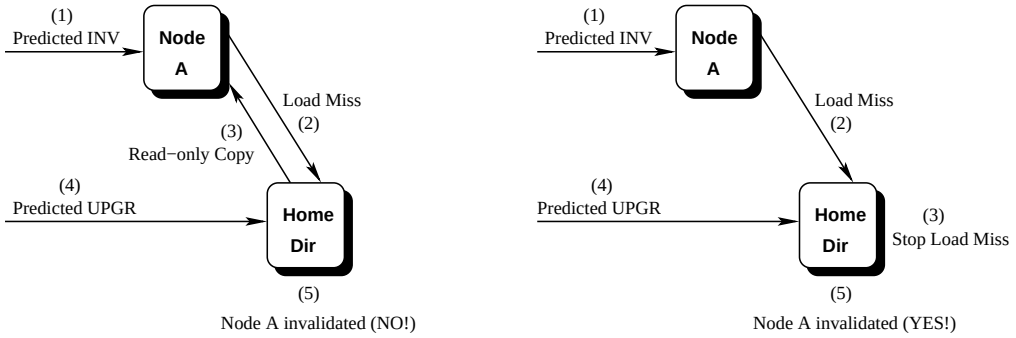


Figure 6.4: Case in which the first scenario occurs (left) and how it is solved (right)

line, so the line must not be invalidated and, in turn, a *NACK* message is sent to the requesting node.

Requesting Node Operation (2). Once the requesting node has received all the *ACK*/*NACK* replies from the predicted nodes as well as the reply from the directory, the miss *would seem* to have been completed. If the miss was not satisfied as an upgrade miss (because a different upgrade miss was first served), the requesting node can now assume exclusive ownership of the line, since it has already been guaranteed by the directory.

6.4.2. Dealing with Race Conditions: the *ILT* Table

On the other hand, in the case of an upgrade miss we cannot be completely sure that all the shared copies of the line have been invalidated at this point. There are two possible scenarios in which this may not actually be true.

The first scenario is illustrated in Figure 6.4 and arises when one of the *predicted* sharers (say node *A*) requests a read-only copy of the previously invalidated line (due to a load miss), and this request reaches the directory before the *predicted* upgrade miss causing the invalidation. The directory would satisfy the load miss as usual[2] and afterwards, on processing the predicted upgrade miss, it would assume that the last copy on node *A* has been invalidated by the requesting node. Fortunately, this situation would be easily avoided if node *A* notifies the directory that the load miss is for a line for which a *predicted* invalidation has previously been received (remember that this information was stored into the L2 entry for the line when the predicted invalidation was applied or into the *ILT* table

[2]This is true since our coherence protocol does not implement replacement hints for lines in the *Shared* state in order to increase throughput.

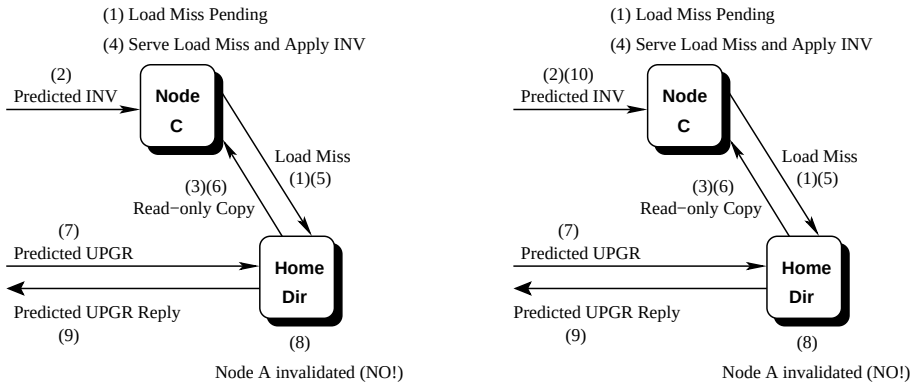


Figure 6.5: Case in which the second scenario occurs (left) and how it is solved (right)

if the entry was subsequently used). This is done by setting a particular bit into the load miss message. In this way, when the directory receives a load miss that has been originated after a *predicted* invalidation for certain line, it will stop the load miss until the corresponding *predicted* upgrade miss has been completed (that is, until node *A* does not appear as holding the line in the sharing code associated with the memory line).

Although strange and infrequent in most cases, a second scenario could still occur (see Figure 6.5). Assume the case of a predicted node (say node *C*), which has a pending load miss. As previously pointed out, a *NACK* response would be returned to the requesting node and the invalidation would be applied after completing the load miss. However, when the *predicted* invalidation message reaches node *C*, it is unknown whether the directory is going to satisfy the pending load miss before the *predicted* upgrade miss or after it. Thus, the line can not be marked as invalidated as a consequence of a prediction and a problem may arise if the load miss reaches the directory before the *predicted* upgrade miss. In this case, it is possible for the invalidated node to incur into another load miss for the line once the reply from the directory for the former load miss has been received and the line has been invalidated as a consequence of the predicted invalidation. If this new load miss reaches the directory once the predicted upgrade miss has been served, there is no problem. Otherwise, a read-only copy of the line would be given to node *C* and subsequently, on receiving the predicted upgrade miss, the directory would assume that the copy in node *C* has been invalidated by the *predicted* message sent previously by the requesting node. In our coherence protocol, this situation is solved by the requesting node. It takes notice of those nodes that having been predicted as sharers, replied with a *NACK* to the predicted invalidation. When the upgrade reply from the directory is received and

all the responses for the predicted nodes have been collected, some of the nodes are re-invalidated. In particular, those from which *NACK* replies were received but the directory identified them as having a copy of the line.

As mentioned previously, the requirement of having a *ILT* table in each one of the nodes of the multiprocessor is motivated by a lack of replacement hints for lines in the *Shared* state. This also causes that some of the invalidations that were predicted must be re-tried. If frequent, this re-send of invalidations could significantly increase the latency of upgrade misses. However, as we will see, this situation has been found to occur very infrequently in most cases.

	Requesting Node	**Sharing Node**	**Directory Node**

On a predicted Upgrade Miss	1. Send *INVs* to predicted nodes
	Send *Upgrade Miss* to Directory
	2. Get *ACK/NACK* from predicted nodes
	Get *Response* from Directory
	3. If it actually was an *Upgrade Miss*
	Invalidate those sharers confirmed by the directory that replied *NACK* (if any)
	
	Assume the *ownership*
		If (line in the *Shared* state)
		If (*pending upgrade* & have *directory reply*)
		Send *NACK* reply
		else
		Send *ACK* reply and *invalidate* the line
		If (!*pending upgrade*)
		Set *Predicted* bit
		If (line in the *Modified* state | line in the *Exclusive* state)
		Send *NACK* reply
		If (line is *not present*)
		Send *NACK* reply
		If (*pending load miss*)
		Note down *invalidation*
			If (line in the *Shared* state)
			If (!*predicted all sharers*)
			Invalidate non-predicted
			Update sharing information
			Send response
			else
			Process as usual

On other miss type		If (*load miss* & (*Predicted* bit = 1 | has an entry in *ILT*))
		Set Predicted bit on load miss
		If (*Predicted* bit = 1 | has an entry in *ILT*))
		Clear bit or free entry
		If (using L2 cache entry & *Predicted* bit is set)
		Insert address in *ILT* table
			If (*load miss* & *Predicted* bit = 1 & node found as *sharer*)
			Wait until *Upgrade* has been completed

Table 6.1: How the coherence protocol works

Table 6.1 summarizes all the situations that could appear as a consequence of applying prediction. The resulting coherence protocol has been thoroughly verified to guarantee its correctness.

6.4.3. Directory Requirements

As in owner prediction, it is important to define the requirements that the sharing codes used by the directories must meet to ensure that the performance benefits offered by our

proposal can be extracted. The technique presented in this chapter could be used in conjunction with compressed and multi-level directories (as well as with exact sharing codes as *full-map*) without limitations. However, if one were to obtain most of the potential benefits, directory information must be codified as accurately as possible.

Remember that one of the observations motivating the technique presented in this chapter is that the number of invalidations that are sent on an upgrade miss is very small. Having a highly compressed sharing code (such as *BT*, for example) would substantially increase this count, blurring the benefits obtained by directly sending invalidation messages from the requesting node to the sharers. In this way, the directory architecture presented in Chapter 4, which employs *BT-SuT* as the sharing code for the third-level directory, would be a good option.

6.5. Performance Evaluation

In this section, we present a detailed evaluation of the technique proposed in this chapter using extensive execution-driven simulations of a 16-node cc-NUMA multiprocessor[3]. For this, we run the same applications that were used in the previous chapter on top of RSIM, and as before, compare the base system (*Base*) with two configurations which use two instances of the prediction scheme presented previously. The first of these configurations, which has been called *UPT*, includes a prediction table with an entry per each one of the memory lines and an unlimited number of entries in the *ILT* table. Of course, this predictor has a prohibitive cost, but it completely eliminates the aliasing effect caused by the non-tagged nature of the prediction table.

The second configuration, called *LPT*, implements a "realistic" version of the predictor. In this case, the number of entries available in both the prediction table and in the *ILT* table is limited to 16K and 128 respectively, resulting in a total size of less than 48 KB. Access to the prediction table is carried out using the 14 bits resulting from computing the XOR between the 14 less significant bits of the line address and its most significant bits. As in the previous chapter, we use XOR-based placement to distribute the utilization of the entries in the prediction table, and thus, to reduce conflicts. Finally, due to its small number of entries, the *ILT* table is organized as a totally associative buffer structure.

Additionally, we also show an approximation to the performance benefits that would be obtained in the perfect case. That is, when the requesting node is able to invalidate all

[3]Again, similar results are obtained for larger configurations.

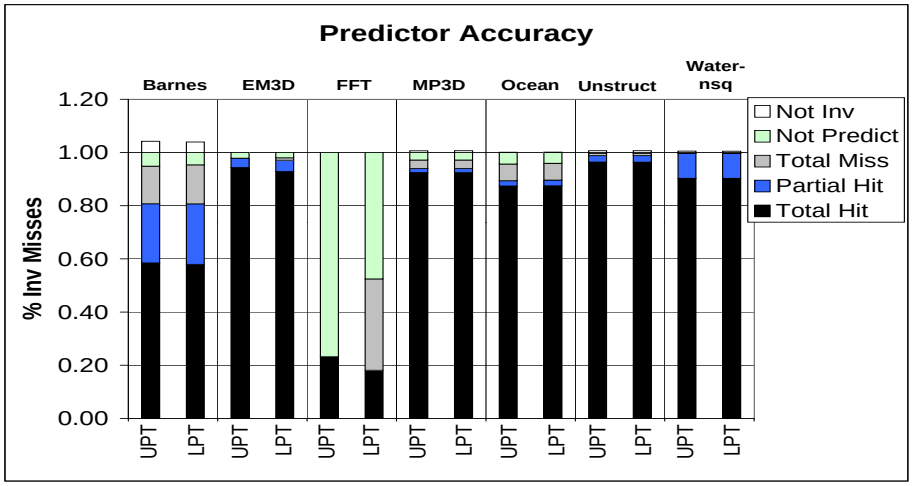


Figure 6.6: Percentage of upgrade misses predicted

the sharers in the case of an upgrade miss, and the directory is just accessed to check the prediction. We simulate this behavior by invalidating the sharers of a particular memory line (when an upgrade miss for the line takes place) at zero cost (that is to say, without spending any cycle). For this, we have modified RSIM to provide directories with a direct access to the processors' cache structures. In this way, on receiving an upgrade miss for a particular memory line, the directory identifies the sharers of the memory line (for which the corresponding directory entry is used) and invalidates the copies of the line by directly accessing the caches to set to *Invalid* the state of the corresponding cache lines. This configuration has been called *POT* and was previously used in Section 6.1 to show the potential of our proposal.

As in Chapter 5, we assume that a full-map single-level directory is used in each one of the nodes of the multiprocessor for all the cases. In this way, we focus on the impact that our technique has on application performance, avoiding the effects of having a compressed directory level.

6.5.1. Predictor Accuracy

Figure 6.6 presents accuracy results for the prediction scheme described in Section 6.3. Over the total number of upgrade misses, it shows the percentage of those for which prediction was applied and one of these situations occurred: (1) all the sharers were predicted and invalidated from the requesting node (*Total Hit*), (2) some of the sharers could be predicted but not all (*Partial Hit*), and finally, (3) others but none of the actual sharers

were predicted (*Total Miss*). Also, we show the percentage of upgrade misses that did not make use of prediction (*Not Predict*), as well as the percentage of predictions that were not resolved as upgrade misses by the directory (*Not Inv*). The latter percentage is computed over the total number of store misses and is shown as starting from 100% (since it corresponds to misses that are not upgrade misses), and takes place for example, when two nodes simultaneously try to gain exclusive ownership of a line in the *Shared* state in both caches. Only one of them will be served as an upgrade miss whereas the other will be subsequently satisfied with a cache-to-cache transfer. However, as derived from Figure 6.6, this situation appears very infrequently in all the applications.

The *Total Hit* and *Partial Hit* cases include those upgrade misses that can take advantage of the use of prediction, and consequently, performance benefits are expected of them (of course, the benefits obtained for the misses falling into the first category are greater). On the other hand, the upgrade misses belonging to the *Total Miss* and *Not Inv* categories could be slowed slightly as a consequence of the increased number of messages, especially when a significant number of invalidations are sent from the requesting node. This results in increased network and cache controller occupancies to create, send and process these invalidation messages and their associated responses, which could have negative consequences on the final performance.

Observe from Figure 6.6 that for all the applications except FFT, the percentage of upgrade misses that are completely or partially predicted represent more than 80% when the *UPT* configuration is used, reaching almost 100% for EM3D, MP3D and UNSTRUCTURED. In general, we have found that upgrade misses are concentrated on a small set of memory lines, which are frequently cached by the same nodes. For FFT only 23% of the upgrade misses could be predicted. The computation in FFT is split into two phases. The first performs the fast Fourier transform using the six-step FFT method, whereas during the second phase, the inverse FFT is computed in order to perform a test of the output. For the majority of the memory lines, a single upgrade miss takes place in each of the phases. In this way, prediction is not applied during the first phase and the majority of the predictions are done during the second one, which explains the large percentage of upgrade misses that were not predicted (almost 80%, when the *UPT* configuration is used).

As can be derived from the results shown in Figure 6.6, the small size of the predictor included in the *LPT* configuration (less than 48 KB) is enough to virtually obtain the same accuracy results than the unbounded predictor used in the *UPT* configuration for all the applications but FFT. Its limited size however, means some of the entries are used

by different memory lines, slightly increasing the number of upgrade misses for which prediction is applied. Whereas the *Partial Hit* and *Total Miss* cases suffer a small growth compared to the results obtained for the *UPT* configuration, this does not affect the *Total Hit* category. For FFT, an important fraction of the upgrade misses that are not predicted when the *UPT* configuration is used are now probed (30%, approximately). Unfortunately, most of them fail to include the correct sharers, causing the significant increase on the *Total Miss* category shown in Figure 6.6.

Finally, note that in most cases, the *Total Miss* category constitutes a very small percentage which remains lower than the *Not Predicted* case for all the applications but BARNES and OCEAN. For these applications, we have found that a small subset of the upgrade misses do not present the repetitive behavior observed for the majority. The initialization of the 2-bit counters of the predictor to 2, makes these upgrade misses be predicted and consequently, failed. Initializing the 2-bit counters to 1 (instead of 2) would eliminate these prediction misses, reducing, as the negative counterpart, the number of prediction hits in favor of the *Not Predicted* case. Thus, due to the small percentage of upgrade misses completely failed (*Total Miss* case), it is preferable to initially set these counters to 2.

6.5.2. Performance Analysis

This section analyzes the consequences that using prediction has on application performance. As carried out before, we first study how prediction affects the average latency of upgrade misses, then the effect on the average latency of load and store misses is presented, and finally, reductions in terms of execution time are also reported. In all the cases, we compare the base configuration which does not use prediction, with the two configurations for which accuracy results were given in the last section (that is, the *UPT* and *LPT* configurations). Additionally, the results obtained for the *POT* system, which represents the maximum benefit that could be obtained, are also presented in some cases.

Correctly predicting the sharers of a certain memory line on an upgrade miss saves the directory having to perform the invalidation process, and consequently, it significantly reduces the number of cycles that must be dedicated to process the upgrade miss. Figure 6.7 illustrates the normalized average latency for upgrade misses split into network latency, directory latency and miscellaneous latency (buses, cache accesses...), for the base, *POT*, *UPT* and *LPT* configurations. Network latency comprises the number of cycles that the

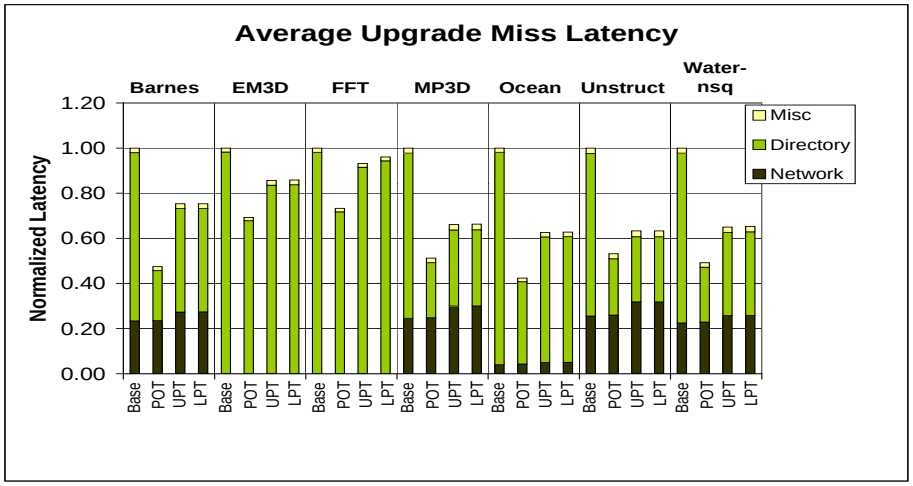


Figure 6.7: Normalized average *upgrade miss* latency

upgrade miss spends on the interconnection network. Directory latency represents the cycles needed by the directory to satisfy the miss. For *UPT* and *LPT* configurations, the directory component of the latency also includes the number of cycles between the missing node receiving the reply to an upgrade miss from the directory, and the response corresponding to the last predicted invalidation has been processed. Normalized average latencies are computed dividing the average latencies for each one of the configurations by the average latencies for the base case. Table 6.2 shows the average number of invalidations that are sent by the directory on an upgrade miss (for the base system), as well as the average number of invalidations that are sent by the requesting node when an upgrade miss is predicted.

As can be derived from Figure 6.7, the use of prediction has the potential of significantly reducing the directory component of the upgrade miss latency for BARNES (53%), MP3D (49%), OCEAN (58%), UNSTRUCTURED (47%), WATER-NSQ (51%) and, in less extension, for EM3D (31%) and FFT (27%). In practice, not all the upgrade misses can be correctly predicted and the benefits that can be obtained are smaller. Observe from Figure 6.7 that the *LPT* prediction scheme significantly reduces the latency of upgrade misses for BARNES (25%), MP3D (34%), OCEAN (37%), UNSTRUCTURED (37%), WATER-NSQ (35%). For EM3D, a reduction of 14% is gained. Finally, the small fraction of upgrade misses that could be successfully predicted in FFT translates into a reduction of 4% on the average upgrade miss latency. These improvements almost coincide with those obtained with the unrealistic predictor (*UPT*). Sending invalidations from the requesting node on an

Application	*Base*	*UPT*	*LPT*
BARNES-HUT	2.08	1.44	1.44
EM3D	1.97	1.81	1.80
FFT	1.00	1.00	1.01
MP3D	1.08	1.63	1.63
OCEAN	1.11	1.24	1.25
UNSTRUCTURED	1.40	1.76	1.76
WATER-NSQ	1.69	1.23	1.24

Table 6.2: Number of invalidations per upgrade miss (for *Base* case) and number of nodes included per prediction (for *UPT* and *LPT* cases)

upgrade miss accelerates the invalidation process, and consequently, the component of the latency caused by the directory is significantly decreased, as shown in Figure 6.7. Now the invalidation of the sharers occurs in parallel with the access to the directory whereas, in a traditional cc-NUMA multiprocessor, it can not start until the directory has determined the identity of the sharers.

Another important issue is the number of invalidations that are sent on a predicted upgrade miss. Ideally, this number should approach the actual number of sharers. On the one hand, sending an excessively large number of invalidations would increase network traffic and cache controllers occupancies. On the other hand, an excessively small number of messages would cause that the directory would still have to invalidate some of the sharers. Fortunately, as shown in Table 6.2, the regularity exhibited by upgrade misses allows our prediction scheme to obtain the high accuracy shown in Figure 6.6 without having to send an excessive number of unnecessary invalidation messages. Only for MP3D and UNSTRUCTURED we have found that the average number of invalidation messages sent on a predicted upgrade miss is slightly greater than the average number of sharers. This is the reason for the small increase on the network component of the average upgrade miss latency observed in Figure 6.7 for these applications (5% and 6%, respectively).

Normalized average latencies for load and store misses are also shown in Figure 6.8. Table 6.3 presents the fraction of L2 misses caused by load and store instructions. Store misses are in turn split into the upgrade miss and rest of store miss categories. As shown in Figure 6.8, for most applications, the important benefits found for upgrade misses also lead to reductions in the average latency of store instructions. For *EM3D*, most store misses find a read-only copy of the line in the local L2 cache causing an upgrade miss (as derived from Table 6.3). Thus, in this case, the reductions reported for upgrade misses

Application	**Load misses**	**Store misses**	
		Upgrade misses	*Rest store misses*
BARNES-HUT	58.96%	25.21%	15.82%
EM3D	73.08%	26.62%	0.30%
FFT	58.55%	4.42%	37.03%
MP3D	24.75%	18.75%	56.50%
OCEAN	45.33%	12.24%	42.43%
UNSTRUCTURED	41.62%	33.35%	25.03%
WATER-NSQ	63.66%	33.14%	3.20%

Table 6.3: Classification of the L2 misses according to the type of the instruction that caused them

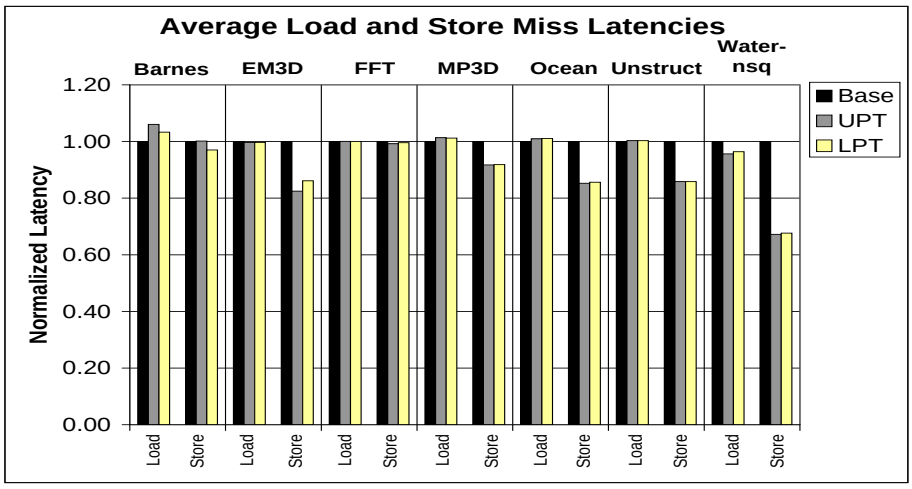


Figure 6.8: Normalized average load/store miss latency

are translated into reductions in terms of store miss latency (14%). For FFT, negligible reductions are obtained. For this application, a very small fraction of the store misses are upgrade misses and small reductions in upgrade miss latency have been reported. For the rest of applications, although more than 12% of the misses are upgrade misses however, part of the store misses do not cause an upgrade miss and thus, prediction can not be applied to them. As shown in Table 6.3, this percentage is small for WATER-NSQ and latency reductions close to those reported for upgrade misses are obtained (32%). On the other hand, reductions of 8%, 15% and 14% on average store miss latency are observed for MP3D, OCEAN and UNSTRUCTURED, respectively. Finally, reductions in terms of upgrade miss latency found in BARNES do not lead to significant improvements on the average latency of store misses. Note also that, in general, our proposal does not increase

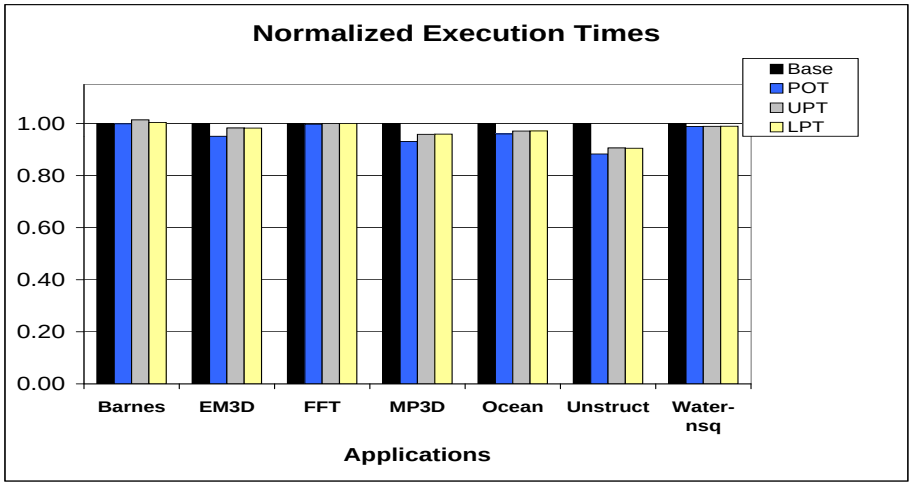


Figure 6.9: Normalized execution times

the latency of load misses. Only for BARNES, a degradation of 6% on average load miss latency is found when the *U*PT configuration is used. For this application, some of the upgrade misses that are predicted and for which some of the invalidation messages have to be re-tried are closely followed by load misses from other nodes, which will have to wait longer to be served.

Finally, for all the applications except EM3D and FFT, the percentage of predicted upgrade misses for which some of the invalidations must be re-tried is lower than 10%. This percentage becomes virtually zero for MP3D, UNSTRUCTURED and WATER-NSQ. For FFT and EM3D, 64% and 41%, respectively, of the upgrade misses that are predicted require invalidation re-tries. This is a consequence of the significant number of replacements for lines in the *Shared* state found for these applications. Remember that re-tries are needed since our coherence protocol does not send replacement hints for memory lines in the *Shared* state. Sending replacement hints in these cases would completely remove the need to re-try some of the invalidations.

The final metric for application performance is the execution time. Figure 6.9 shows the normalized execution times for the *POT*, *UPT* and *LPT* configurations with respect to those obtained for the base system. As can be observed, reductions of 2% for EM3D, 4% for MP3D, 3% for OCEAN, 10% for UNSTRUCTURED and 1% for WATER-NSQ are obtained when both the *UPT* and *LPT* configurations are used. For these applications, the obtained reductions are very close to those obtained with the *POT* configuration. For FFT and BARNES the use of prediction has no positive impact on the final execution times, as the

results obtained for the *POT* configuration demonstrate. Whereas a small degradation (1%) is found for BARNES when the *UPT* configuration is used, this degradation completely disappears in the *LPT* case.

6.6. Conclusions

Upgrade misses are caused by a store instruction for which the local L2 cache holds a read-only copy of the memory line, and the directory is accessed in order to gain exclusive ownership of the line. This requires a message sent from the missing node to the directory, a directory lookup in order to find the set of sharers, invalidation messages being sent to the sharers and responses to the invalidations being sent back. Only when the directory has received all the replies to the invalidations, the requesting node is given the ownership of the line, and consequently, the penalty paid by these misses is not negligible.

In this chapter, we propose sharers prediction, a technique that significantly accelerates upgrade misses by directly performing the invalidation of the sharers from the requesting node. The idea is the following: on an upgrade miss, the requesting node predicts the current sharers for the memory line and sends invalidation messages to them. In order to check the prediction, the upgrade miss is also sent to the directory. However, differently from a conventional coherence protocol, the access to the directory is now performed in parallel with the invalidation process. If all sharers were correctly predicted, the directory immediately gives the ownership of the line. On the contrary, if any of the sharers were not sent the corresponding invalidation message, the directory invalidates it before responding to the requesting node. The requesting node assumes the ownership of the line once it has received all the responses from the predicted nodes as well as the acknowledgment message from the directory.

Our proposal is based on the observation that upgrade misses present a repetitive behavior. Additionally, the number of sharers that must be invalidated on an upgrade miss has been previously reported to be very small (one, in some cases), which reveals that more efficient treatment of these misses could be rewarding.

The prediction-based technique proposed in this chapter consists of two main components. The first is an address-based prediction engine able to find out the identity of up to three potential sharers in case of an upgrade miss. The second component is a four-state MESI coherence protocol, similar that used in the SGI Origin 2000, properly extended (with minimal changes) to support the use of prediction.

The effects of our proposal were evaluated executing several shared memory applications on top of the RSIM detailed execution-driven simulator. First, we analyzed the accuracy of the proposed predictor and found that for a great percentage of the upgrade misses (which reaches almost 100% for some applications) the set of sharers was successfully predicted. Then, we analyzed our proposal in terms of its impact on application performance. The high percentage of upgrade misses correctly predicted is translated into reductions on average upgrade miss latency of more than 35% in some cases. These benefits translated into reductions in the average latency of store instructions whereas load latency remained virtually unaffected. Finally, application execution times were accelerated by up to 10%. These results can be obtained including a predictor with a total size of less than 48 KB in every one of the nodes of the multiprocessor.

Chapter 7

Conclusions and Future Directions

7.1. Conclusions

Although cc-NUMA constitutes an attractive architecture for building high-performance scalable shared memory multiprocessors, far away from the limits imposed by snooping protocols, there are a number of factors that constrain the amount of nodes that can be offered at a good price/performance ratio. Two of these factors are the increased cost in terms of hardware overhead and the long L2 cache miss latencies.

The most important component of the hardware overhead is the amount of memory required to store directory information, which depends, to a large extent, on the selection of the sharing code used in each one of the entries of the directory. Having a sharing code as full-map, whose size in bits is a linear function on the number of nodes, results in very high memory requirements for large-scale configurations of a parallel machine (for example, directory memory overhead becomes 100% for a system with 1024 nodes and 128-byte memory lines).

On the other hand, the long L2 miss latencies that characterize cc-NUMA architectures are a consequence of the inefficiencies that the distributed nature of directory-based coherence protocols and the underlying scalable network imply. As a result of the increased distance to memory, the most important of such inefficiencies is the indirection introduced by the access to the directory, which is usually stored in main memory. These inefficiencies, which do not appear in SMP designs, are one of the main reasons why many efforts have been made to extend the SMP designs beyond the number of processors supported by a single bus.

Our efforts in this thesis have focused on these two key issues. In this way, techniques

aimed at reducing the directory memory overhead and the long L2 miss latencies that characterize cc-NUMA architectures have been proposed and evaluated, and important conclusions can be extracted from the results that have been presented.

The first conclusion we can draw is that organizing the directory as a two-level structure combining both a small first-level directory and a complete *BT-SuT* second-level directory, drastically reduces the amount of memory required by the directory while achieving the performance of a non-scalable full-map directory. *Two-level directories*, which has been presented in Chapter 3, combine a very small full-map directory cache (the first-level directory) with a full and compressed directory structure (the second-level directory). Our proposal is motivated by the locality found for directory accesses, even when they are performed by different nodes. The idea behind a two-level directory architecture is to have precise sharing information for the most recently referenced memory lines (which will be probably accessed again soon) at the same time that updated compressed sharing information is available for all the memory lines. The second-level directory acts as a backup of the first level: it is used whenever an entry in the first-level directory was not found. Therefore, the sharing code used by this level has to be small enough to reduce the memory overhead and consequently, increase the scalability, and precise enough to preserve the performance. We showed that *BT-SuT* meets these requirements and, thus, is a good candidate for this level. *BT-SuT* is one of the three new compressed sharing codes that are derived from the *multi-layer clustering* concept presented in Chapter 3. This sharing code obtains performance numbers comparable with some previously proposed sharing codes while having less memory requirements.

Recent processor designs, such as the Alpha 21364 EV7, already integrate on-chip some key components, such as the memory controller, coherence hardware, and network interface and router. Assuming that a cc-NUMA multiprocessor is formed based on these processors, the second conclusion we extract is that the additional inclusion on-chip of a small first-level directory and a shared data cache has a significant impact on the latency of L2 misses, leading to reductions up to 66% in some cases.

An important property of the two-level directory architecture proposed in Chapter 3 (multi-level directories, in general) is the possibility of implementing the small first-level directory using fast cache memories. In this way, directory accesses could be accelerated in those cases in which directory information were provided by the first level by avoiding the access to the slower main memory. This would reduce the time needed to satisfy L2 cache misses, and constitutes the starting point for Chapter 4.

The chapter begins with a *taxonomy of the L2 cache misses* found in cc-NUMA multiprocessors in terms of the actions executed by directories to serve them. In this way, L2 cache misses are classified into four categories: *$-to-$* misses, *Mem* misses, *Inv* misses and *Inv+Mem* misses.

Then, we propose a *novel node architecture* which exploits current integration scale through the introduction of some key components of the system inside the processor chip, as a means of significantly accelerating L2 cache misses. The main objective is to reduce the time a particular L2 cache miss spends at the home node. For this, we extend the two-level directory evaluated in Chapter 3 to a three-level directory architecture in which a small first-level directory is integrated into the processor chip. Additionally, our design includes a small shared data cache in the processor die which is accessed when the home node has to provide a copy of the memory line. The on-chip integration of the small first-level directory (which uses a limited number of pointers as the sharing code) and the shared data cache ensures performance, since the access to main memory is avoided when the directory controller finds the information in these structures. On the other hand, the memory overhead entailed by the use of directories is significantly decreased by having the second and third-level directories out of the processor chip. The third-level directory constitutes the complete directory structure (one entry per memory line) and uses the *BT-SuT* compressed sharing code to drastically cut down memory requirements, whereas the second-level directory is a small directory cache (which uses full-map as the sharing code) that tries to minimize the negative effects of having an imprecise third level as well as to quickly provide directory information when it is not present in the first level.

Finally, we show how a cc-NUMA multiprocessor using the proposed node architecture can achieve important reductions in the average miss latency of each one of the L2 miss types when compared with a traditional cc-NUMA multiprocessor in which each node includes the coherence controller into the processor chip. Latency reductions up to 66% for *$-to-$* misses, 45% for *Mem* misses, 45% for *Inv* misses and 55% for *Inv+Mem* misses are obtained. These reductions translate into important improvements in the application execution times (reductions of up to 53%).

The proposal presented in Chapter 4 constitutes a general framework focusing on reducing the latency of the four categories of the classification by avoiding accesses to the slower main memory. Additionally, we observe that the role played by the home node in two of the four categories (i.e. for *$-to-$* and *Inv* misses) basically consists in posting the appropriate coherence messages (transfer requests and invalidations, respectively) to the

nodes that hold a copy of the memory line (the memory line is not needed). For these misses, the directory is accessed to discover the identity of the nodes that have to receive the corresponding coherence messages. An additional conclusion derived from this thesis is that 3-hop misses (*$-to-$* misses) and upgrade misses (*Inv* misses) are significantly accelerated when the associated coherence messages are sent directly from the requesting node, instead of the directory (latency reductions of up to 45% and 35%, respectively).

First, in Chapter 5, we propose the use of prediction to send directly 3-hop misses to the node where the single valid copy of the line resides. The technique, called *owner prediction*, consists of two main components. The first one is a two-step prediction scheme that includes a first-step predictor, an instruction-based predictor (i.e. indexed using the PC of the instruction that caused the miss), which is used to detect whether a certain L2 cache miss is 3-hop or not, and a second-step predictor, a predictor indexed using both the PC and the address of the miss, which is used to provide a list of the potential owners of the memory line. The second component is a coherence protocol properly extended to support the use of prediction. We use, as the starting point, the four-state MESI coherence protocol included in RSIM (which is similar to the one used in the SGI Origin 2000) and show that minimal changes are required. Finally, we find that significant reductions (up to 45%) on the average latency of 3-hop misses are derived from owner prediction, and that these results can be obtained using a predictor with a total size of less than 64 KB.

Later, in Chapter 6, we study how a similar technique can also be applied for accelerating upgrade misses. With *sharers prediction*, the technique presented in Chapter 6, on suffering an upgrade miss, invalidation messages are sent directly from the requesting node to those nodes predicted as being the current sharers of the line. In parallel and in order to check the predictions, upgrade misses are also sent to the corresponding home directory. However, in contrast to conventional coherence protocols, the access to the directory is performed in parallel with the invalidation process. If all sharers have been correctly predicted, the directory immediately gives ownership of the line. On the contrary, if any of the sharers has not been sent the corresponding invalidation message, the directory invalidates it before responding to the requesting node. The requesting node assumes ownership of the line once it has received all the responses from the predicted nodes as well as the acknowledgment message from the directory.

As with owner prediction, the prediction-based technique proposed in Chapter 6, consists of two main components. The first is an address-based prediction engine able to find out the identity of up to three potential sharers in case of an upgrade miss. The second

component is a four-state MESI coherence protocol, similar to that used in the SGI Origin 2000, properly extended (with minimal changes) to support the use of prediction. We find that since a large fraction of the upgrade misses are successfully predicted, sharers prediction can significantly reduce the latency of upgrade misses (latency reductions of more than 35% in some cases) and that these results can be obtained including a predictor with a total size of less than 48 KB in every node.

In summary, in this thesis we have presented several techniques aimed at reducing the high costs, in terms of both memory overhead and increased L2 miss latencies, that characterize directory-based shared memory multiprocessors. We have seen how multi-level directories can be used to significantly reduce the memory overhead of using directories without hurting performance. At the same time, they allow for implementations which can significantly accelerate L2 misses by reducing the time needed by the home node to obtain directory information and thus, the negative effects caused by the indirection introduced by directories can be minimized. Moreover, we have shown how prediction could be applied in some cases to completely remove the access to the directory from the critical path of the misses. Different from other previous proposals, the predictors proposed in this work are concise enough to be implemented using small memories.

7.2. Future Directions

A natural extension of this work would be to combine the proposals presented in this thesis to orchestrate an architecture able to compete in terms of both cost and performance with SMPs of medium size (64-256 nodes), at the same time that the scalability to larger systems is preserved.

Additionally, it would be interesting to study the possibility of combining both multicast network support and compressed directories. Multicast schemes could be used to cut down the negative impact that unnecessary coherence messages have on the final performance by significantly reducing the number of messages sent on every coherence event. If successful, this could allow for multi-level directory organizations with less memory requirements by using more aggressive compressed sharing codes for the complete directory structure.

Some other future research directions related to the area of this thesis include:

- Apart from the two issues studied in this thesis, availability has become increasingly important as cache-coherent shared memory multiprocessors has seen widespread

use in commercial, technical and scientific applications. Recently, techniques for providing fault-tolerance in these architectures has been the focus of several works, such as ReVive [76] and SafetyNet [83].

- Another issue which is currently the subject of significant research is reduced energy consumption. High power dissipation has historically been a concern of mobile system designers as it reduces battery life, diminishing the utility of these platforms. Nowadays, however, high power dissipation has also become an important issue for server designers. It causes that more expensive packaging and cooling technology are required, increases cost, and decreases reliability [64]. In state-of-the-art processors, a significant fraction of the power is dissipated in the caches. Moshovos *et al.* found that in SMP servers lower cache hierarchy levels (such as the L2) dissipate considerable amounts of power [64]. In these systems, all bus-side cache controllers must *snoop* the bus upon every load/store miss, substantially increasing the access frequency to lower-level caches as compared to uniprocessors. There are optimizations to cut down the energy consumed in L2 caches, such as using a dedicated tag array for snoops. However, these optimizations only reduce energy consumption in the data array not in the tag array. Since SMPs incorporate large L2 caches with high associativity, the energy dissipated in tag arrays is also substantial.

Bibliography

[1] M. E. Acacio, J. González, J. M. García and J. Duato. "A New Scalable Directory Architecture for Large-Scale Multiprocessors". *Proc. of the 7th Int'l Symposium on High Performance Computer Architecture (HPCA-7)*, pp. 97–106, January 2001.

[2] M. E. Acacio, J. González, J. M. García and J. Duato. "A Novel Approach to Reduce L2 Miss Latency in Shared-Memory Multiprocessors". *Proc. of the 16th Int'l Parallel and Distributed Processing Symposium (IPDPS'02)*, April 2002.

[3] M. E. Acacio, J. González, J. M. García and J. Duato. "Owner Prediction for Accelerating Cache-to-Cache Transfer Misses in cc-NUMA Multiprocessors". *Proc. of the Int'l SC2002 High Performance Networking and Computing*, November 2002.

[4] M. E. Acacio, J. González, J. M. García and J. Duato. "Reducing the Latency of L2 Misses in Shared-Memory Multiprocessors through On-Chip Directory Integration". *Proc. of the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing (PDP'2002)*, pp. 368–375, January 2002.

[5] M. E. Acacio, J. González, J. M. García and J. Duato. "The Use of Prediction for Accelerating Upgrade Misses in cc-NUMA Multiprocessors". *Proc. of the 2002 Int'l Conference on Parallel Architectures and Compilation Techniques (PACT 2002)*, pp. 155–154, September 2002.

[6] S. V. Adve and K. Gharachorloo. "Shared Memory Consistency Models: A Tutorial". *WRL Research Report 95/7*, September 1995.

[7] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie and D. Yeung. "The MIT Alewife Machine: Architecture and Performance". *Proc. of the 22nd Int'l Symposium on Computer Architecture (ISCA'95)*, pp. 2–13, May/June 1995.

[8] A. Agarwal, R. Simoni, J. Hennessy and M. Horowitz. "An Evaluation of Directory Schemes for Cache Coherence". *Proc. of the 15th Int'l Symposium on Computer Architecture (ISCA'88)*, pp. 280–289, May 1988.

[9] G. S. Almansi and A. Gottlieb. *"Highly Parallel Computing"*. Addison-Wesley Publishing Company, Inc., 1994.

[10] E. Anderson, J. Brooks, C. Grassl and S. Scott. "Performance of the CRAY T3E Multiprocessor". *Proc. of the Int'l SC1997 High Performance Networking and Computing*, November 1997.

[11] L. A. Barroso, K. Gharachorloo and E. Bugnion. "Memory System Characterization of Commercial Workloads". *In Proc. of the 25th Int'l Symposium on Computer Architecture (ISCA'98)*, pp. 3–14, June 1998.

[12] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets and B. Verghese. "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing". *Proc. of the 27th Int'l Symposium on Computer Architecture (ISCA'00)*, pp. 282–293, June 2000.

[13] E. E. Bilir, R. M. Dickson, Y. Hu, M. Plakal, D. J. Sorin, M. D. Hill and D. A. Wood. "Multicast Snooping: A New Coherence Method Using a Multicast Address Network". *Proc. of the 26th Int'l Symposium on Computer Architecture (ISCA'99)*, pp. 294–304, May 1999.

[14] J. Boyle, R. Butler, T. Disz, B. Glickfield, E. Lusk, R. Overbeek, J. Patterson and R. Stevens. *"Portable Programs for Parallel Processors"*. Holt, Rinehart and Winston, Inc., 1987.

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

[16] D. Chaiken, J. Kubiatowicz and A. Agarwal. "LimitLESS Directories: A Scalable Cache Coherence Scheme". *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pp. 224–234, April 1991.

[17] Y. Chang and L. Bhuyan. "An Efficient Hybrid Cache Coherence Protocol for Shared Memory Multiprocessors". *IEEE Transactions on Computers*, pp. 352–360, March

1999.

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

[19] J. H. Choi and K. H. Park. “Segment Directory Enhancing the Limited Directory Cache Coherence Schemes”. *Proc. of the 13th Int’l Parallel and Distributed Processing Symposium (IPDPS’99)*, pp. 258–267, April 1999.

[20] F. T. Chong, B. H. Lim, R. Bianchini, J. Kubiatowicz and A. Agarwal. “Application Performance on the MIT Alewife Multiprocessor”. *IEEE Computer*, 29(12):57–64, December 1996.

[21] Convex Computer Corp. “Convex Exemplar Architecture”. *dhw-014 edition*, November 1993.

[22] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, S. Luna, T. von Eicken and K. Yelick. “Parallel Programming in Split-C”. *Proc. of the Int’l SC1993 High Performance Networking and Computing*, pp. 262–273, November 1993.

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

[24] D. Dai and D. K. Panda. “Reducing Cache Invalidation Overheads in Wormhole Routed DSMs Using Multidestination Message Passing”. *Proc. of International Conference on Parallel Processing (ICPP’96)*, 1:138–145, August 1996.

[25] J. Duato, S. Yalamanchili and L. Ni. *“Interconnection Networks: An Engineering Approach”*. Morgan Kaufmann Publishers, Inc., 2002.

[26] M. Durbhakula, V. S. Pai and S. V. Adve. “Improving the Accuracy vs. Speed Tradeoff for Simulating Shared-Memory Multiprocessors with ILP Processors”. *Proc. of the 5th Int’l Symposium on High Performance Computer Architecture (HPCA-5)*, pp. 23–32, January 1999.

[27] F. Gabbay and A. Mendelson. “Speculative Execution Based on Value Prediction”. *Technical Report #1080, Electrical Engineering Department*, 1996.

[28] K. Gharachorloo, M. Sharma, S. Steely and S. V. Doren. “Architecture and Design

of AlphaServer GS320". *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pp. 13–24, November 2000.

[29] A. González, M. Valero, N. Topham and J. M. Parcerisa. "Eliminating Cache Conflict Misses through XOR-Based Placement Functions". *Proc. of the Int'l Conference on Supercomputing (ICS'97)*, pp. 76–83, 1997.

[30] J. González and A. González. "The Potential of Data Value Speculation to Boost ILP". *Proc. of the Int'l Conference on Supercomputing (ICS'98)*, pp. 21–28, 1998.

[31] J. Goodman. "Using Cache Memories to Reduce Processor-Memory Traffic". *Proc. of the Int'l Symposium on Computer Architecture (ISCA'83)*, June 1983.

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

[33] A. Gupta, W.-D. Weber and T. Mowry. "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes". *Proc. Int'l Conference on Parallel Processing (ICPP'90)*, pp. 312–321, August 1990.

[34] D. Gustavson. "The Scalable Coherent Interface and Related Standards Projects". *IEEE Micro*, 12(1):10–22, Jan/Feb 1992.

[35] L. Gwennap. "Alpha 21364 to Ease Memory Bottleneck". *Microprocessor Report*, 12(14):12–15, October 1998.

[36] H. Hadimioglu, D. Kaeli and F. Lombardi. "Introduction to the Special Issue on High Performance Memory Systems". *IEEE Transactions on Computers*, 50(11):1103–1105, November 2001.

[37] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen and K. Olukotun. "The Stanford Hydra CMP". *IEEE Micro*, 20(2):71–84, March/April 2000.

[38] M. A. Heinrich. "The Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols". *Ph.D. Thesis, Stanford University*, 1998.

[39] J. L. Hennessy and D. A. Patterson. "*Computer Architecture: A Quantitative Approach*". Morgan Kaufmann Publishers, Inc., third edition, 2002.

[40] M. D. Hill. “Multiprocessors Should Support Simple Memory-Consistency Models”. *IEEE Computer*, 31(8):28–34, August 1998.

[41] M. D. Hill, N. P. Jouppi and G. S. Sohi. *“Readings in Computer Architecture”*. Morgan Kaufmann Publishers, Inc., 2000.

[42] T. Hosomi, Y. Kanoh, M. Nakamura and T. Hirose. “A DSM Architecture for a Parallel Computer Cenju-4”. *Proc. of the 6th Int’l Symposium on High Performance Computer Architecture (HPCA-6)*, pp. 287–298, January 2000.

[43] C. J. Hughes, V. S. Pai, P. Ranganathan and S. V. Adve. “RSIM: Simulating Shared-Memory Multiprocessors with ILP Processors”. *IEEE Computer*, 35(2):40–49, February 2002.

[44] L. Iftode, J. P. Shing and K. Li. “Understanding Application Performance on Shared Virtual Memory Systems”. *Proc. of the 23rd Int’l Symposium on Computer Architecture (ISCA’96)*, pp. 122–133, May 1996.

[45] R. Iyer and L. N. Bhuyan. “Switch Cache: A Framework for Improving the Remote Memory Access Latency of CC-NUMA Multiprocessors”. *Proc. of the 5th Int’l Symposium on High Performance Computer Architecture (HPCA-5)*, pp. 152–160, January 1999.

[46] R. Iyer, L. N. Bhuyan and A. Nanda. “Using Switch Directories to Speed Up Cache-to-Cache Transfers in CC-NUMA Multiprocessors”. *Proc. of the 14th Int’l Parallel and Distributed Processing Symposium (IPDPS’00)*, pp. 721–728, May 2000.

[47] S. Kaxiras. “Identification and Optimization of Sharing Patterns for Scalable Shared-Memory Multiprocessors”. *Ph.D. Thesis, University of Wisconsin-Madison*, 1998.

[48] S. Kaxiras and J. R. Goodman. “Improving CC-NUMA Performance Using Instruction-Based Prediction”. *Proc. of the 5th Int’l Symposium on High Performance Computer Architecture (HPCA-5)*, pp. 161–170, January 1999.

[49] S. Kaxiras and C. Young. “Coherence Communication Prediction in Shared-Memory Multiprocessors”. *Proc. of the 6th Int’l Symposium on High Performance Computer Architecture (HPCA-6)*, pp. 156–167, January 2000.

[50] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin,

D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum and J. Hennessy. "The Stanford FLASH Multiprocessor". *Proc. of the 21st Int'l Symposium on Computer Architecture (ISCA'94)*, pp. 302–313, April 1994.

[51] A. C. Lai and B. Falsafi. "Memory Sharing Predictor: The Key to a Speculative DSM". *Proc. of the 26th Int'l Symposium on Computer Architecture (ISCA'99)*, pp. 172–183, May 1999.

[52] A. C. Lai and B. Falsafi. "Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction". *Proc. of the 27th Int'l Symposium on Computer Architecture (ISCA'00)*, pp. 139–148, May 2000.

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

[54] J. Laudon and D. Lenoski. "The SGI Origin: A ccNUMA Highly Scalable Server". *Proc. of the 24th Int'l Symposium on Computer Architecture (ISCA'97)*, pp. 241–251, June 1997.

[55] A. R. Lebeck and D. A. Wood. "Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors". *Proc. of the 22nd Int'l Symposium on Computer Architecture (ISCA'95)*, pp. 48–59, June 1995.

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

[57] D. E. Lenoski and W.-D. Weber. *"Scalable Shared-Memory Multiprocessing"*. Morgan Kaufmann Publishers, Inc., 1995.

[58] M. H. Lipasti and J. P. Shen. "Exceeding the Dataflow Limit via Value Prediction". *Proc. of the Int'l Symposium on Microarchitecture (MICRO-1996)*, pp. 226–237, 1996.

[59] T. Lovett and R. Clapp. "STiNG: A CC-NUMA Computer System for the Commercial Marketplace". *Proc. of the 23rd Int'l Symposium on Computer Architecture (ISCA'96)*, pp. 308–317, 1996.

[60] M. P. Malumbres, J. Duato and J. Torrellas. "An Efficient Implementation of Tree-based Multicast Routing for Distributed Shared-Memory Multiprocessors". *Proc. of the 8th Int'l Symposium on Parallel and Distributed Processing (SPDP'96)*, pp. 186–189, 1996.

[61] M. M. Martin, D. J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore, M. Plakal, M. D. Hill and D. A. Wood. "Timestamp Snooping: An Approach for Extending SMPs". *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pp. 25–36, November 2000.

[62] M. M. Martin, D. J. Sorin, M. D. Hill and D. A. Wood. "Bandwidth Adaptive Snooping". *Proc. of the 8th Int'l Symposium on High Performance Computer Architecture (HPCA-8)*, pp. 251–262, February 2002.

[63] M. M. Michael and A. K. Nanda. "Design and Performance of Directory Caches for Scalable Shared Memory Multiprocessors". *Proc. of the 5th Int'l Symposium on High Performance Computer Architecture (HPCA-5)*, pp. 142–151, January 1999.

[64] A. Moshovos, G. Memik, B. Falsafi and A. Choudhary. "JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers". *Proc. of the 7th Int'l Symposium on High Performance Computer Architecture (HPCA-7)*, pp. 85–96, January 2001.

[65] S. S. Mukherjee and M. D. Hill. "An Evaluation of Directory Protocols for Medium-Scale Shared-Memory Multiprocessors". *Proc. of the 8th Int'l Conference on Supercomputing (ICS'94)*, pp. 64–74, July 1994.

[66] S. S. Mukherjee and M. D. Hill. "Using Prediction to Accelerate Coherence Protocols". *Proc. of the 25th Int'l Symposium on Computer Architecture (ISCA'98)*, pp. 179–190, July 1998.

[67] S. S. Mukherjee, S. D. Sharma, M. D. Hill, J. R. Larus, A. Rogers and J. Saltz. "Efficient Support for Irregular Applications on Distributed-Memory Machines". *Proc. of the 5th Int'l Symposium on Principles & Practice of Parallel Programming (PPOPP'95)*, pp. 68–79, July 1995.

[68] A. K. Nanda, A.-T. Nguyen, M. M. Michael and D. J. Joseph. "High-Throughput Coherence Controllers". *Proc. of the 6th Int'l Symposium on High Performance Com-*

puter Architecture (HPCA-6), pp. 145–155, January 2000.

[69] A. K. Nanda, A.-T. Nguyen, M. M. Michael and D. J. Joseph. "High-Throughput Coherence Control and Hardware Messaging in Everest". *IBM Journal of Research and Development*, 45(2):229–244, March 2001.

[70] H. Nilsson and P. Stenström. "The Scalable Tree Protocol – A Cache Coherence Approach for Large-Scale Multiprocessors". *Proc. of 4th Int'l Symposium on Parallel and Distributed Processing (SPDP'92)*, pp. 498–506, December 1992.

[71] J. Nilsson and F. Dahlgren. "Reducing Ownership Overhead for Load-Store Sequences in Cache-Coherent Multiprocessors". *Proc. of the 14th Int'l Parallel and Distributed Processing Symposium (IPDPS'00)*, pp. 684–692, May 2000.

[72] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, W. Radke and S. Vishin. "The S3.mp Scalable Shared Memory Multiprocessor". *Proc. of the Int'l Conference on Parallel Processing (ICPP'95)*, pp. I:1–10, July 1995.

[73] B. O'Krafka and A. Newton. "An Empirical Evaluation of Two Memory-Efficient Directory Methods". *Proc. of the 17th Int'l Symposium on Computer Architecture (ISCA'90)*, pp. 138–147, May 1990.

[74] V. Pai, P. Ranganathan and S. Adve. "RSIM Reference Manual version 1.0". *Technical Report 9705, Department of Electrical and Computer Engineering, Rice University*, August 1997.

[75] V. S. Pai, P. Ranganathan, H. Abdel-Shafi and S. Adve. "The Impact of Exploiting Instruction-Level Parallelism on Shared-Memory Multiprocessors". *IEEE Transactions on Computers*, 48(2):218–226, February 1999.

[76] M. Prvulovic, Z. Zhang and J. Torrellas. "ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors". *Proc. of the 29th Int'l Symposium Computer Architecture (ISCA'02)*, pp. 111–122, May 2002.

[77] P. F. Reynolds, C. Williams and R. R. Wagner. "Isotach Networks". *IEEE Transactions on Parallel and Distributed Systems*, 8(4):337–348, April 1997.

[78] R. M. Russell. "The CRAY-1 Computer System". *Communications of the ACM*, 21(1):63–72, January 1978.

[79] Y. Sazeides and J. E. Smith. "The Predictability of Data Values". *Proc. of the Int'l Symposium on Microarchitecture (MICRO-97)*, pp. 248–258, 1997.

[80] R. Simoni. "Cache Coherence Directories for Scalable Multiprocessors". *Ph.D. Thesis, Stanford University*, 1992.

[81] R. Simoni and M. Horowitz. "Dynamic Pointer Allocation for Scalable Cache Coherence Directories". *Proc. Int'l Symposium on Shared Memory Multiprocessing*, pp. 72–81, April 1991.

[82] J. Singh, W.-D. Weber and A. Gupta. "SPLASH: Stanford Parallel Applications for Shared-Memory". *Computer Architecture News*, 20(1):5–44, March 1992.

[83] D. J. Sorin, M. M. Martin, M. D. Hill and D. A. Wood. "SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery". *Proc. of the 29th Int'l Symposium Computer Architecture (ISCA'02)*, pp. 123–134, May 2002.

[84] P. Stenström, M. Brorsson, F. Dahlgren, H. Grahn and M. Dubois. "Boosting the Performance of Shared Memory Multiprocessors". *IEEE Computer*, 30(7):63–70, July 1997.

[85] J. Tendler, J. Dodson, J. Fields, H. Le and B. Sinharoy. "POWER4 System Microarchitecture". *IBM Journal of Research and Development*, 46(1):5–25, January 2002.

[86] The BlueGene/L Team. "An Overview of the BlueGene/L Supercomputer". *Proc. of the Int'l SC2002 High Performance Networking and Computing*, November 2002.

[87] J. Torrellas, L. Yang and A. T. Nguyen. "Toward A Cost-Effective DSM Organization that Exploits Processor-Memory Integration". *Proc. of the 6th Int'l Symposium on High Performance Computer Architecture (HPCA-6)*, pp. 15–25, January 2000.

[88] A. J. van der Steen and J. J. Dongarra. "Overview of Recent Supercomputers". *Available at http://www.euroben.nl/reports/web02/overview02.html*, July 2002.

[89] W.-D. Weber, S. Gold, P. Helland, T. Shimizu, T. Wicki and W. Wilcke. "The Mercury Interconnect Architecture: A Cost-Effective Infrastructure for High-Performance Servers". *Proc. of the 24th Int'l Symposium on Computer Architecture (ISCA'97)*, pp. 98–107, June 1997.

[90] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh and A. Gupta. "The SPLASH-2 Programs: Characterization and Methodological Considerations". *Proc. of the 22nd Int'l Symposium on Computer Architecture (ISCA'95)*, pp. 24–36, June 1995.

[91] K. C. Yeager. "The MIPS R10000 Superscalar Microprocessor". *IEEE Micro*, 16(2):28–40, April 1996.

[92] Z. Zhang. "Architectural Sensitive Application Characterization: The Approach of High-Performance Index-Set (HP-Set)". *Technical Report HPL-2001-75, HP Laboratories Palo Alto*, March 2001.

[93] Z. Zhou, W. Shi and Z. Tang. "A Novel Multicast Scheme to Reduce Cache Invalidation Overheads in DSM Systems". *Proc. of the 19th IEEE Int'l Performance, Computing and Communications Conference*, pp. 597–603, February 2000.

Appendix A

RSIM Coherence Protocol and Compressed Directories

This appendix presents the invalidation-based, four-state MESI coherence protocol included in RSIM [43]. It is described how the protocol operates using cache states, directory states and network transactions. Additionally, we show the extensions that had to be introduced to support the use of compressed sharing codes. Finally, we point out the main differences between this protocol and the one implemented by the SGI Origin 2000 multiprocessor [54].

A.1. Original Coherence Protocol

RSIM simulates a hardware cache-coherent distributed shared memory system, with variations of a full-mapped invalidation-based directory coherence protocol. The coherence protocol is orchestrated by *directory controllers* which access directory information in order to satisfy L2 cache misses.

Similarly to [23] and chapters 5 and 6, the following terminology will be used for our discussion of the coherence protocol. For a given cache or memory line:

- The *directory node* or *home node* is the node to whose main memory the line is allocated.
- The *exclusive node* or *owner node* is the node that currently holds the single valid copy of a line and must supply the data when needed.

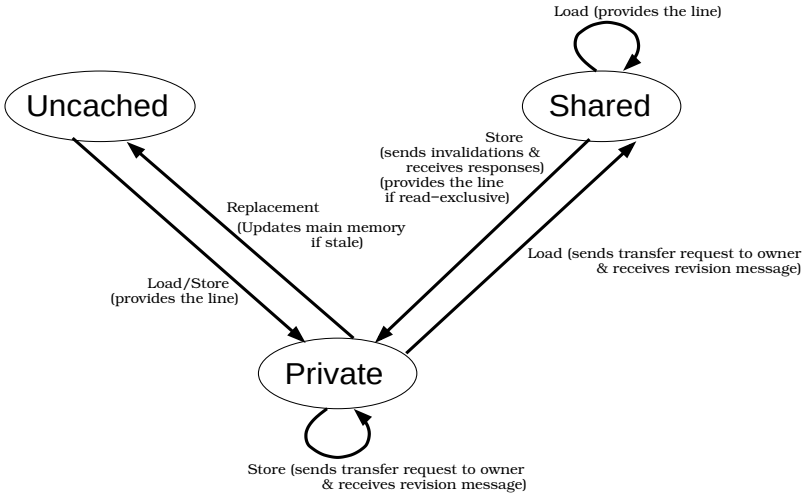


Figure A.1: State transition diagram for a memory line at the directory in the coherence protocol implemented by RSIM

- The *requesting node* is the node containing the processor that issues a request for the line.
- The *sharing nodes* are those nodes that hold read-only copies of the line.

A.1.1. Protocol States

In general, there are two operations that a directory-based cache coherence protocol must implement: handling a load miss and handling a store miss. To implement these operations, the directory must track the state of each memory line. In the protocol implemented by RSIM, a memory line may be in one of these three states at the directory (without considering *busy* states):

- *Uncached*: No processor has a copy of the memory line.
- *Shared*: Several processors have a copy of the memory line cached, and the value in main memory is up-to-date (as well as in the caches).
- *Private*: Exactly one processor has a copy of the memory line and it may be written the line, so that the copy in main memory copy may be stale.

Figure A.1 presents how the transitions between these states are performed for the different types of requests. In the caches however, the protocol employees the MESI states:

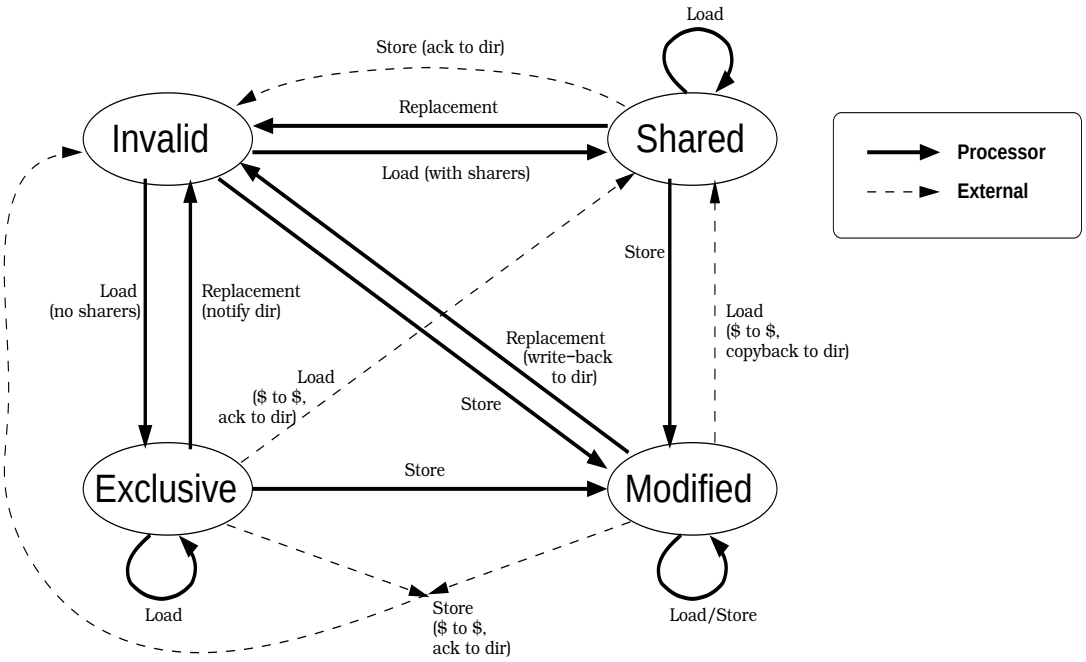


Figure A.2: State transition diagram for a memory line at the caches in the coherence protocol implemented by RSIM

- *Invalid (I)*: The cache does not hold a copy of the line.
- *Shared (S)*: The cache has a read-only copy of the memory line which is being cached by several processors (the value in main memory is up-to-date).
- *Exclusive (E)*: Only this cache has a copy of the memory line and it has not been modified (that is, the value in main memory is still up-to-date).
- *Modified (M)*: Also called *dirty*, means that only this cache has a valid copy of the line and the copy in main memory is now stale.

Figure A.2 gives the simplified state diagram for the caches. It shows the MESI states and the way in which the transitions between them are performed. Transitions due to local processor requests are shown in bold arcs, while those due to external coherence actions are shown in dashed arcs. Next subsections present detailed descriptions of how load, store and replacement requests are handled.

A.1.2. Handling Load Requests

Every time a processor issues a load that misses in its cache hierarchy, the address of the miss is examined to determine the identity of the corresponding home node and then, the load miss is sent to it. At the home directory, an entry in the directory buffers is reserved for the miss. If no buffer entry is available at that moment, a *RAR* message is returned to the requesting node. This message indicates that the miss could not be served at that time, and must be re-tried.

Otherwise, the directory entry associated with the memory line is looked up. At the same time, main memory is speculatively accessed to obtain a copy of the memory line. The directory entry lookup may indicate that the line is in one of the three states presented in Figure A.1 (or even busy), and different actions are taken in each case.

- The line is in the *Shared* or *Uncached* states. This means that main memory at the home has the latest copy of the data (so the speculative access was successful). If the state is *Shared*, the bit representing the requesting node is set in the full-map sharing code[1]. Otherwise, if the line is *Uncached*, the directory state is changed to *Private* and the requesting node is registered as having the line. Finally, the home node completes the miss by sending the line back to the requesting node in a reply message. Note that in those cases in which the home and requesting nodes coincide, then no network messages are generated and the miss is locally satisfied.

- A miss for the line is currently in progress. This situation is indicated by a particular bit in the corresponding directory entry, which is called the *pending* bit. This bit is set each time the directory starts serving a request for the line, and reset after it has finished. In this case, the miss will wait until the preceding miss has been completed. Once the ongoing request has been completed, the directory wakes up the misses for the line that were waiting.

- The line is in the *Private* state. In this case, the home directory cannot provide the memory line since its copy may be stale. Instead, the *pending* bit is set and the request is forwarded (as a *transfer request*) to the corresponding owner node, which directly replies to the requesting node and sets the state of its local copy of the line

[1] In those cases in which the bit was already set, a replacement for the memory line previously took place at the requesting node. Since replacement hints are not sent for lines in the *Shared* state, the directory assumes this and serves the miss.

to *Shared*. The identity of the current owner is obtained from the corresponding directory entry. Additionally, the owner node sends a revision message to the home node indicating that the transfer was performed and includes a copy of the memory line if it was modified. Once this revision message has been received, the home directory changes the state of the memory line to *Shared*, adds the requesting node as a sharer of the line, and finally, resets the *pending* bit. Additionally, main memory is updated in those cases in which the memory line was included into the revision message (i.e. main memory was stale). An interesting situation takes place when the directory notices that the requesting node is supposed to be the owner node. In this case, the directory assumes that a prior replacement for the memory line occurred and that the request has overtaken the corresponding replacement hint (write-back or replacement message). The request will wait until the replacement has been received. Another interesting case happens when the owner node does not hold the memory line (for example, because it previously replaced it). In this case, the owner sends a negative acknowledgment (*NACK*) to the home directory. On receiving the *NACK* response, the directory checks whether the expected replacement message has been received or not. In the second case, the original request that started the coherence transaction sequence is sent back to its node (the requesting node) with an *RAR* to be retried.

A.1.3. Handling Store Requests

Store misses that invoke the protocol may generate either read-exclusive requests, which need both copy of the line and ownership, or upgrade requests, that need only ownership since the requesting node already has a valid copy of the line. Similarly to a load miss, the address of the store miss is examined to determine the identity of the home node, and the miss is sent to this node. At the home directory, an entry in the directory buffers is reserved, and as for load misses, a *RAR* message is returned to the requesting node if all entries were being used at that moment.

Otherwise, the corresponding directory entry is looked up to determine the state of the memory line and consequently, what actions to take to serve the request. If the state at the directory is anything but *Uncached* (and of course, the corresponding *pending* bit is not set), the copies held in other caches must be invalidated. To preserve the ordering model, invalidations must be explicitly acknowledged. The situations contemplated at the

directory when a store miss must be served are the following.

- The line is in the *Uncached* state. If the request is an upgrade, the state that is expected for the memory at the directory is *Shared*, not *Uncached*. The state being *Uncached* means that some other processor previously acquired the ownership of the line (which, after the upgrade miss was generated, invalidated the copy of the line at the requesting node), and after this, replaced this line (this is possible since the protocol implemented in RSIM does not assume point-to-point network order). Therefore, the directory can assume that the copy of the line held by the requesting node was previously invalidated and changes the upgrade miss to read exclusive, which can be served at that time. If the request is a read exclusive, the directory state is changed to *Private* and the presence bit associated with the requestor is set. The home node replies with the copy of the line from memory (which, as in the case of load misses, was accessed in parallel with the directory lookup).

- The line is in the *Shared* state. If the request is an upgrade, all copies of the memory line in the caches, except the one held by the requesting node, must be invalidated before the directory can reply to the request. If the requesting node does not appear as being one of the sharing nodes, the upgrade miss is converted to a read-exclusive miss[2]. On the other hand, for a read-exclusive request, the line must be invalidated in all the caches that hold copies of it. The directory controller at the home first makes a list of the sharers that are to be sent invalidations, for which it uses the corresponding directory entry. Then, it sets the *pending* bit and creates and sends the invalidation messages. Finally, once all the responses to the invalidations have been received, the response to the requesting node is sent.

- The line is in the *Private* state. If the request is an upgrade, another store miss has beaten this request to the home. An upgrade miss is no longer the appropriate request, and it is converted and processed as a read-exclusive miss. For a read-exclusive request, the directory sets the corresponding *pending* bit and redirects the miss, as a transfer request, to the current owner node (whose identity is obtained from the corresponding directory entry). On receiving the transfer request from the home node, the owner replies with the memory line to the requesting node and invalidates its local copy. Also, it sends the revision message to the home node indicating that

[2] In this case, a prior store request invalidated the local copy of the memory line and, subsequently, a load miss changed the state to *Shared*.

the transfer was performed. In this case, the revision message does not need to carry a copy of the memory line. Once the revision message has been received, the state of the line is not changed (i.e. it is kept *Private*), the requesting node is set as the single node holding the line, and finally, the *pending* bit is reset. Additionally, the two situations previously identified for load misses, can also arise for store misses, and would be handled in the same way.

- A miss for the line is currently in progress. As before, the directory can identify this situation when the corresponding *pending* bit was previously set. In this case, the miss will wait until the ongoing miss has been completed.

A.1.4. Handling Write-Back Requests and Replacements

Each time a certain node replaces a modified line (dirty) in its cache, it generates a write-back request which goes to the home directory. Since the copy at the home is stale, this request carries the memory line. On the other hand, when the replacement is for a line that is exclusively held but was not modified (that is, it is in the *Exclusive* state), a replacement request is generated and sent to the directory. In this case, the memory line is not included into the message since the home directory has a valid copy of the line.

On receiving a write-back/replacement request, the *pending* bit associated with the memory line can be 0 or 1 at that time. If it is not set, directory information is updated by changing the state of the memory line to *Uncached* and clearing the sharing code. Additionally, main memory is written with the copy of the line included in the request for write-backs. Otherwise, if the *pending* bit is set, the write-back/replacement request is processed first, after which the outstanding request is resumed.

The protocol implemented in RSIM does not send replacement hint messages to the home directory in those cases in which the line being replaced is in the *Shared* state. Sending replacement hints in these situations would avoid the following useless invalidations to those lines, but they incur in directory controller occupancy and do not reduce traffic. In fact, if the line is not written again by other node, then the replacement hints are wasteful.

A.2. Extensions to Support Compressed Directories

The coherence protocol provided by RSIM was originally designed based on the use of a precise sharing code, in particular a full-map one. As previously explained, the use of

full-map directories, which exactly codify the identity of those nodes that hold a copy of a particular line, allowed RSIM to infer that some situations are taking place on receiving a request for the line. For example, when a particular memory line is in the *Private* state and the directory receives a request from the current owner of the line, it assumes that a replacement for the line previously took place in that node and stops the miss until the corresponding replacement message has been received.

Unfortunately, the use of compressed sharing codes precludes the directory from doing these kind of assumptions and the coherence protocol must be extended to detect that these situations are happening. In particular, we have to provide support for two situations that cannot always be detected by the original coherence protocol when a compressed directory is used.

- First, in those cases in which the sharing code is not wide enough to exactly codify the identity of a single node, the case presented in the previous example cannot always be detected, so that the miss cannot be stopped until the replacement message has been received. In this case, the request would not be delayed and coherence messages would be sent to all the possible owners (that is, those indicated by the compressed sharing code). Just once all these nodes have replied with a negative acknowledgment message (*NACK*), the directory realizes that the replacement has happened. If the corresponding replacement message has already been received, the miss can be satisfied as usual at that time; otherwise, the miss will be served after the replacement message has been received.

- Second, in the original coherence protocol, the directory performs a transparent conversion from an upgrade miss to a read-exclusive one in several situations. One of them is when the line is in the *Shared* state and the requesting node does not happen to be one of the current sharers. In this case, the directory knows that some preceding store miss invalidated the read-only copy held by the requesting node, and therefore, that the miss must be converted to read-exclusive. Unfortunately, this case cannot always be detected when a compressed sharing code is used, since more sharers than the existing ones are usually included. In our extended protocol, the upgrade miss would be satisfied as usual and only after the requesting node has received the ownership of the line from the directory, would the situation be detected. At that time, the requesting node would change the upgrade to a read-exclusive and would

inform the directory that the conversion was done, for which a particular bit into the message is set.

A.3. Differences with the SGI Origin 2000 Protocol

Although the coherence protocol provided by RSIM is similar in most aspects to that implemented in the SGI Origin 2000 [54] (for example, in both cases, replacement hints are not sent for lines in the *Shared* state), some differences can be enlightened. This section presents the most important issues in which these two protocols differ.

- Even though the directory state is *Private*, the protocol implemented by the Origin sends a copy of the memory line to the requesting node. The home optimistically assumes that the line will be in the *Exclusive* rather than *Modified* state in the owner's cache, and sends the speculatively accessed memory line at the home as a *speculative reply*. At the same time, as is done in RSIM, the home forwards the transfer request to the owner. The owner checks the state in its cache and if dirty, includes the memory line in the reply message sent to the requesting node. Then, this data will overwrite the stale speculative reply sent by the home. Note that the use of speculative replies does not have any significant performance advantage, since the requesting node has to wait to know the real state at the exclusive node anyway before it can use the data. However, the cache controller of the R10000 processor [91] that the Origin uses, does not return data when it receives a transfer request for a line in the *Exclusive* state, which motivates the use of speculative replies [23].

- Another discrepancy between the protocols is the place in which responses to invalidation messages are meant to be collected. Whereas in RSIM the directory must receive all the responses to the invalidation messages before replying to the requesting node, the protocol included in the SGI Origin 2000 performs the collection at the requesting node. For this, once the home directory has sent the corresponding invalidations, it returns a response to the requesting node containing the number of sharing nodes from which invalidation acknowledgments are expected, and the requestor waits for all the acknowledgments to come in before it completes the operation.

- As previously explained, each time the home directory must process a new miss for a particular memory line and finds that another request for the same line is being

served at that time (that is, the corresponding *pending* bit is 1), it stops the incoming request into a buffer entry until the ongoing one has been completed. Instead, the SGI Origin 2000 sends a negative acknowledgment (*NACK*) message each time the home finds that a previous request for the line is still in progress. The *NACK* message asks the requesting node to try again later.

- As we showed, a transparent conversion from an upgrade request to a read-exclusive one is directly performed by the home directory in some situations (for example, when an upgrade request is received and the memory line happens to be in the *Uncached* state). Contrary to this, in the protocol provided by the Origin 2000, each time the directory finds a situation in which an upgrade request is not appropriated, it *NACKs* the request asking the requestor to convert the upgrade into a different request.