Universidad de Murcia
Departamento de Ingeniería y
Tecnología de Computadores

Efficient and Scalable Cache Coherence for Many-Core Chip Multiprocessors

A dissertation submitted in fulfillment
of the requirements for the degree of
Doctor of Philosophy

By
Alberto Ros Bardisa

Advisors
José Manuel García Carrasco
Manuel Eugenio Acacio Sánchez

Murcia, July 2009

Universidad de Murcia
Departamento de Ingeniería y
Tecnología de Computadores

Coherencia de Cache Eficiente y Escalable para Multiprocesadores en un Solo Chip

Tesis propuesta para
la obtención del grado de
Doctor

Autor:
Alberto Ros Bardisa

Directores:
José Manuel García Carrasco
Manuel Eugenio Acacio Sánchez

Murcia, Julio de 2009

UNIVERSIDAD DE
MURCIA

D. José Manuel García Carrasco, Catedrático de Universidad del Área de Arquitectura y Tecnología de Computadores en el Departamento de Ingeniería y Tecnología de Computadores

y

D. Manuel Eugenio Acacio Sánchez, Profesor Titular de Universidad del Área de Arquitectura y Tecnología de Computadores en el Departamento de Ingeniería y Tecnología de Computadores

AUTORIZAN:

La presentación de la Tesis Doctoral titulada «*Efficient and Scalable Cache Coherence for Many-Core Chip Multiprocessors*», realizada por D. Alberto Ros Bardisa, bajo su inmediata dirección y supervisión, y que presenta para la obtención del grado de Doctor por la Universidad de Murcia.

En Murcia, a 26 de Junio de 2009.

Fdo: Dr. José Manuel García Carrasco

Fdo: Dr. Manuel Eugenio Acacio Sánchez

Departamento de Ingeniería y Tecnología de Computadores
Campus Universitario de Espinardo. 30100 Murcia
Tel. 868 884 665 – Fax 868 884 151 – **www.um.es/ditec**

UNIVERSIDAD DE
MURCIA

D. Antonio Javier Cuenca Muñoz, Profesor Titular de Universidad del Área de Arquitectura y Tecnología de Computadores y Director del Departamento de Ingeniería y Tecnología de Computadores, INFORMA:

Que la Tesis Doctoral titulada «*Efficient and Scalable Cache Coherence for Many-Core Chip Multiprocessors*», ha sido realizada por D. Alberto Ros Bardisa, bajo la inmediata dirección y supervisión de D. José Manuel García Carrasco y de D. Manuel Eugenio Acacio Sánchez, y que el Departamento ha dado su conformidad para que sea presentada ante la Comisión de Doctorado.

En Murcia, a 26 de Junio de 2009.

Fdo: Dr. Antonio Javier Cuenca Muñoz

Departamento de Ingeniería y Tecnología de Computadores

Campus Universitario de Espinardo. 30100 Murcia
Tel. 868 884 665 – Fax 868 884 151 – **www.um.es/ditec**

A mis padres

Abstract

The huge number of transistors that are currently offered in a single die has made major microprocessor vendors shift towards multi-core architectures in which several processor cores are integrated on a single chip. Although most current chip multiprocessors (CMPs) have a relatively small number of cores (2 to 8), in the near future chips with tens of cores, also known as many-core CMPs, will become more popular.

Since most many-core CMPs are expected to use the hardware-managed coherent caches model for the on-chip memory, the cache coherence protocol will be a key component for achieving good performance in these architectures. Nowadays, directory-based protocols constitute the best alternative to keep cache coherence in large-scale systems. Nevertheless, directory-based protocols have two important issues that prevent them from achieving better scalability: the directory memory overhead and the long cache miss latencies.

The memory overhead of a directory protocol mainly comes from the structures required to keep the coherence (or directory) information. Depending on how this information is organized, its memory overhead could be prohibitive for many-core CMPs. The long L2 miss latencies are a consequence of the access to the directory information (the indirection problem), and its distributed nature. The access to the directory information is necessary before performing coherence actions in directory-based protocols. On the other hand, since both data and directory caches are commonly distributed across the chip (non-uniform cache architecture or NUCA), and wire delays of many-core CMPs will cause cross-chip communications of the order of tens of cycles, the access latency to these structures will be dominated by the wire delay to reach each particular cache bank rather than the time spent accessing the bank itself.

Our efforts in this thesis have focused on these key issues. First, we present a *scalable distributed directory organization* that copes with the memory overhead of

directory-based protocols. In this directory organization, the memory required to keep all the necessary coherence information does not increase with the number of cores, up to a certain number, which corresponds with the number of sets of the private caches. In this way, our organization requires less area than a traditional directory to keep the same information and, therefore, similar performance is obtained. Besides, this organization allows the implementation of an *implicit replacements* mechanism which is able to remove coherence messages caused by replacements. Hence, this mechanism reduces the total network traffic generated by the coherence protocol which finally translates into savings in terms of power consumption.

Second, we propose a new family of cache coherence protocols called *direct coherence protocols*. These protocols are aimed at avoiding the indirection problem of traditional directory-based protocols, but without relying on broadcasting requests. The key property of these protocols is the assignment of the task of keeping cache coherence to the cache that provides the data block in a cache miss, instead of to the home cache, as happens in directory-based protocols. Indirection is avoided by directly sending requests to that cache. This indirection avoidance reduces the average cache miss latency and, consequently, the applications' execution time. Since requests are only sent to just one destination, the network traffic is kept low. Additionally, we also analyze the use of compressed sharing codes to reduce the memory overhead in direct coherence protocols.

Finally, we develop a novel mapping policy managed by the OS that reduces the long access latency to a NUCA cache. We name this policy as *distance-aware round-robin* and it tries to map memory pages to the local NUCA bank of the first core that requests a block belonging to that page. In this way, the average access latency to a NUCA cache is reduced. Furthermore, the proposed policy also introduces an upper bound on the deviation of the distribution of memory pages among cache banks, which lessens the number of off-chip accesses. In this way, we reduce the average cache access latency and the number of off-chip accesses, which translates into improvements in applications' execution time.

Agradecimientos

Mucho ha sido el esfuerzo y dedicación que he empleado para realizar esta tesis, especialmente durante el último año, y como consecuencia, muchas han sido las personas a las que he prestado menos atención de la debida durante este tiempo. Es por ello que a la vez de disculparme, quiero agradecerles la paciencia y comprensión que han tenido conmigo todos estos años.

Las primeras palabras de agradecimiento van dirigidas a mis padres. La educación, valores y ánimos que he recibido de ellos constantemente me han ayudado a cumplir los objetivos que me he ido proponiendo. Han pasado ya diez años desde que decidí estudiar Ingeniería Informática. Entonces también tenía en mente otras carreras, pero fueron ellos los que me animaron a tomar esta decisión de la cual no me arrepiento en absoluto. Mi hermana ha sido también un apoyo muy importante y todo un ejemplo de constancia. A ella también va dedicada esta tesis con especial cariño. Y cómo no, quiero agradecer también el apoyo recibido de mi cuñado Javi, mis tíos, mis primos y demás familiares, que me han ayudado a alcanzar una meta que hace meses veía casi imposible.

Mis amigos de Cartagena han sido un pilar fundamental para mi todos estos años y me han ayudado a despejarme en los momentos de más estrés. Mención especial se merecen Diego, Esteban y Juan Antonio, con los que he compartido muy buenos momentos. A ellos y los demás amigos que me han estado a mi lado todos estos años, muchas gracias.

Hace siete años me vine a vivir a Murcia. Muchos han sido los pisos por los que he pasado y otros tantos los compañeros que he tenido. Ellos han hecho más agradable mi estancia, especialmente Jose, Francisco y Antonio. Jose, compañero de carrera, piso, doctorado, y un gran amigo, ha estado siempre ahí, y ha sido un apoyo muy importante sobre todo en los momentos más difíciles.

También quiero transmitir mis agradecimientos a Elena, a sus padres, Lola y José Alberto, y a toda su familia, a la que tengo un gran aprecio, y que ha

conseguido que desde el primer día me sintiera en Murcia como en mi propia casa. Tampoco puedo olvidar el apoyo recibido de Rosa, Juan y su hijo Alex que me han tratado siempre como uno más de la familia.

Pero sin lugar a dudas, el sitio donde más tiempo he pasado estos años es la universidad y el apoyo recibido de mis amigos y compañeros del departamento ha sido crucial para hacer estos largos días más amenos. La ayuda de Ricardo durante los años de doctorado, e incluso antes, ha sido vital y me ha ahorrado muchos quebraderos de cabeza. La de Rubén, con el inglés y sus consejos, también ha sido esencial. Ellos dos y el resto de compañeros del laboratorio 3.01, Dani, Juan Manuel, Chema y Kenneth, han sido de mucha ayuda manteniendo el cluster siempre operativo, resolviendo mis dudas y compartiendo los descansos en la sala *pachá*. Asímismo, quiero agradecerle a Miguel Ángel M. su colaboración en la creación de las aplicaciones utilizadas en la tesis. Jose, Antonio y Manolo constituyen también una pieza fundamental en mi día a día. Por último, las vueltas al campus con Miguel Ángel N. y los partidos de baloncesto me han ayudado a desconectar después de un largo día de trabajo. A todos vosotros, muchas gracias.

Mi estancia en Edimburgo el verano pasado supuso un cambio importante en mi vida, tanto en lo profesional como en lo personal. Me alegro mucho de haber estado allí esos cuatro meses y de haber conocido a gente tan especial. *Thanks to the people working at the Informatics Forum, and especially to Marcelo for giving me chance of joining his research group, and providing very important technical advice. Chronis has been a great colleague and a better friend during my stay, and without him my days in Edinburgh would certainly not have been so enriching. He, Damon, George, Horacio, and the friends who came to my goodbye dinner have been like a family for me those four months. Thanks to everyone.*

Finalmente, quiero agredecer a mis directores, José Manuel y Manolo, el apoyo prestado y la confianza depositada en mí todos estos años. Gracias por darme la oportunidad de realizar la tesis en este departamento y en este campo que tanto me gusta.

Contents

Abstract **11**

Agradecimientos **13**

Contents **15**

List of Figures **21**

List of Tables **25**

List of Acronyms **27**

0 Resumen **31**

0.1 Introducción 31

0.1.1 Contribuciones de la tesis 33

0.2 Entorno de evaluación 34

0.3 Una organización de directorio escalable 37

0.3.1 Directorio escalable 38

0.3.2 Reemplazos implícitos 40

0.3.3 Resultados 41

0.4 Protocolos de coherencia directa 44

0.4.1 La coherencia directa 45

0.4.2 Reducción de la sobrecarga de memoria 49

0.4.3 Resultados 50

0.5 Política de mapeo de caches sensible a la distancia y a la tasa de fallos . . . 54
0.5.1 Mapeo sensible a la distancia y a la tasa de fallos . . . 55
0.5.2 Cambio en los bits de indexación de las caches . . . 57
0.5.3 Resultados . . . 58
0.6 Conclusiones y vías futuras . . . 61

1 Introduction and Motivation 63
1.1 The cache coherence problem . . . 65
1.2 Cache hierarchy organization . . . 68
1.3 Thesis contributions . . . 69
1.4 Thesis overview . . . 71

2 Background 73
2.1 Private versus shared organization . . . 74
2.2 Cache coherence protocols . . . 76
2.3 Design space for cache coherence protocols . . . 77
2.3.1 Optimization for migratory sharing . . . 82
2.4 Protocols for unordered networks . . . 82
2.4.1 Hammer protocol . . . 83
2.4.2 Token protocol . . . 84
2.4.3 Directory protocol . . . 85
2.4.4 Summary . . . 87
2.5 Sending unblock messages . . . 88

3 Evaluation Methodology 91
3.1 Simulation tools . . . 92
3.1.1 Simics-GEMS . . . 92
3.1.2 SiCoSys . . . 93
3.1.3 CACTI . . . 94
3.2 Simulated system . . . 94
3.3 Metrics and methods . . . 96
3.4 Benchmarks . . . 98

3.4.1 Scientific workloads 99
3.4.2 Multimedia workloads 102
3.4.3 Multi-programmed workloads 103

4 A Scalable Organization for Distributed Directories 105
4.1 Introduction 105
4.2 Background on directory organizations 107
4.2.1 Directory organizations for CMPs 111
4.3 Scalable directory organization 113
4.3.1 Granularity of directory interleaving 113
4.3.2 Conditions required for ensuring directory scalability 115
4.3.3 Directory structure 117
4.3.4 Changes in the cache coherence protocol 118
4.4 Implicit replacements 119
4.5 Evaluation results and analysis 121
4.5.1 Directory memory overhead 121
4.5.2 Reductions in number of coherence messages 123
4.5.3 Impact on execution time 125
4.6 Managing scalability limits and locality issues 126
4.6.1 Scalability limits 126
4.6.2 Locality of the accesses to the shared cache 128
4.7 Conclusions 129

5 Direct Coherence Protocols 131
5.1 Introduction 131
5.2 Related work and background 134
5.2.1 The lightweight directory architecture 136
5.3 Direct coherence protocols 139
5.3.1 Direct coherence basis 139
5.3.2 Changes to the structure of the tiles of a CMP 143
5.3.3 Description of the cache coherence protocol 145
5.3.4 Preventing starvation 148
5.4 Updating the L1 coherence cache 149

5.5 Area and power considerations 152
5.6 Evaluation results and analysis 155
5.6.1 Impact on the number of hops needed to solve cache misses 156
5.6.2 Impact on cache miss latencies 159
5.6.3 Impact on network traffic 161
5.6.4 Impact on execution time 163
5.6.5 Trade-off between network traffic and indirection 164
5.7 Conclusions 165

6 Traffic-Area Trade-Off in Direct Coherence Protocols 167
6.1 Introduction 167
6.2 Classification of cache coherence protocols 170
6.2.1 Traditional protocols 170
6.2.2 Indirection-aware protocols 170
6.2.3 Summary 171
6.3 Reducing memory overhead 171
6.4 Evaluation results and analysis 174
6.4.1 Impact on area overhead 174
6.4.2 Impact on network traffic 176
6.4.3 Trade-off between network traffic and area requirements . . 177
6.4.4 Impact on execution time 178
6.4.5 Overall analysis 179
6.5 Conclusions 180

7 A Distance-Aware Mapping Policy for NUCA Caches 183
7.1 Introduction 183
7.2 Background and related work 186
7.2.1 Background on mapping policies in NUCA caches 186
7.2.2 Related work 188
7.3 Distance-aware round-robin mapping 190
7.4 First-touch mapping and private cache indexing 192
7.5 Evaluation results and analysis 193
7.5.1 Private cache indexing and miss rate 195

7.5.2 Average distance to the home banks 198
7.5.3 Number of off-chip accesses 199
7.5.4 Trade-off between distance to home and off-chip accesses . 201
7.5.5 Execution time 202
7.5.6 Network traffic 202
7.6 Conclusions 205

8 Conclusions and Future Directions **207**
8.1 Conclusions 207
8.2 Future directions 213

A Direct Coherence Protocol Specification **217**

Bibliography **225**

List of Figures

0.1 Organización de un *tile* y un *tiled* CMP de 4×4 celdas. 35
0.2 Granularidad del intercalado de directorio y efecto que causa en el tamaño mínimo necesario para almacenar toda la información de las caches privadas. 38
0.3 Mapeo entre entradas de cache y directorio. 39
0.4 Diferencias entre los reemplazos tradicionales e implícitos. 41
0.5 Sobrecarga del área de directorio en mm^2 en función del número de nodos. 42
0.6 Reducción en el número de mensajes de coherencia. 43
0.7 Cómo se resuelven los fallos de transferencia de cache a cache en un protocolo de directorio y en un protocolo de coherencia directa. 45
0.8 Cómo se resuelven los fallos por actualización (*upgrades*) en un protocolo de directorio y en un protocolo de coherencia directa. 46
0.9 Modificaciones requeridas por los protocolos de coherencia directa en la estructura de un *tile*. 47
0.10 Sobrecarga del área de directorio en mm^2 en función del número de nodos. 51
0.11 Tráfico de red normalizado. 52
0.12 Tiempo de ejecución normalizado. 53
0.13 Ejemplo de la política de mapeo propuesta. 56
0.14 Cambios en los bits de indexación de las caches. 57
0.15 Tiempo de ejecución normalizado. 59

0.16 Tráfico de red normalizado. 60

1.1 Organization of a tile and a 4×4 tiled CMP. 65

2.1 Private and shared L2 cache organizations. 75
2.2 State transition diagram for a MSI protocol. 79
2.3 State transition diagram for a MESI protocol. 80
2.4 State transition diagram for a MOESI protocol. 81
2.5 A cache-to-cache transfer miss in each one of the described protocols. 84
2.6 Management of unblock messages for cache-to-cache transfer misses. 88

3.1 Multi-programmed workloads evaluated in this thesis. 103

4.1 Alternatives for storing directory information. 108
4.2 Granularity of directory interleaving and its effect on directory size. . 114
4.3 Mapping between cache entries and directory entries. 116
4.4 Finding coherence information. 118
4.5 Differences between the proposed coherence protocol and a traditional coherence protocol. 120
4.6 Directory memory overhead as a function of the number of tiles. . . . 122
4.7 Reductions in number of coherence messages. 124
4.8 Impact on execution time. 126
4.9 Directory memory overhead for two different systems and several configurations. 127

5.1 Trade-off between *Token* and directory protocols. 132
5.2 Cache design for the lightweight directory architecture. 139
5.3 How cache-to-cache transfer misses are solved in directory and direct coherence protocols. 140
5.4 Tasks performed in cache coherence protocols. 141
5.5 How upgrades are solved in directory and direct coherence protocols. 142
5.6 Modifications to the structure of a tile required by direct coherence protocols. 144
5.7 Example of ownership change upon write misses. 147

5.8 Example of a starvation scenario in direct coherence protocols. 149
5.9 Organization of the address signature mechanism proposed to send hints. 151
5.10 How each miss type is solved. 157
5.11 Normalized L1 cache miss latency. 159
5.12 Normalized network traffic. 161
5.13 Normalized execution time. 163
5.14 Trade-off between network traffic and indirection. 164

6.1 Traffic-area trade-off in cache coherence protocols. 168
6.2 Overhead introduced by the cache coherence protocols. 175
6.3 Normalized network traffic. 176
6.4 Traffic-area trade-off. 177
6.5 Normalized execution time. 178
6.6 Trade-off among the three main design goals for coherence protocols. 179

7.1 Trade-off between *round-robin* and *first-touch* policies. 185
7.2 Granularity of L2 cache interleaving and its impact on average home distance. 187
7.3 Behavior of the distance-aware round-robin mapping policy. 191
7.4 Changes in the L1 cache indexing policy. 192
7.5 Number of pages mapped to each cache bank in a first-touch policy. . 195
7.6 Impact of the changes in the indexing of the private L1 caches on parallel applications. 196
7.7 Impact of the changes in the indexing of the private L1 caches on multi-programmed workloads. 197
7.8 Average distance between requestor and home tile.. 199
7.9 Normalized number of off-chip accesses. 200
7.10 Trade-off between distance to home and off-chip accesses. 201
7.11 Normalized execution time. 203
7.12 Normalized network traffic. 204

List of Tables

0.1 Parámetros del sistema simulado. 36
0.2 Bits requeridos para almacenar la información de coherencia. 49
0.3 Resumen de los protocolos evaluados. 51

2.1 Properties of blocks according to their cache state. 78
2.2 Summary of cache coherence protocols. 87

3.1 System parameters. 95
3.2 Benchmarks and input sizes used in the simulations. 98

5.1 Memory overhead introduced by coherence information (per tile) in a 4x4 tiled CMP. 153
5.2 Area overhead introduced by coherence information (per tile) in a 4x4 tiled CMP. 154

6.1 Summary of cache coherence protocols. 171
6.2 Bits required for storing coherence information. 173

A.1 Coherence messages. 217
A.2 Processor messages. 218
A.3 L1 cache controller states. 219
A.4 L1 cache controller events. 219
A.5 L1 cache controller actions. 220
A.6 L1 cache controller transitions. 221

A.7 L2 cache controller states. 222
A.8 L2 cache controller events. 222
A.9 L2 cache controller actions. 223
A.10 L2 cache controller transitions. 223
A.11 Memory controller states. 224
A.12 Memory controller events. 224
A.13 Memory controller actions. 224
A.14 Memory controller transitions. 224

List of Acronyms

AS: Address Signatures.

ASR: Adaptive Selective Replication.

BT: Binary Tree.

CACTI: Cache Access and Cycle Time Information.

cc-NUMA: cache coherent Non Uniform Memory Access.

CCR: Complete and Concise Remote (directory).

CMP: Chip MultiProcessor.

CV: Coarse Vector.

DARR: Distance-Aware Round-Robin.

DCT: Discrete Cosine Transform.

DiCo: Direct Coherence.

FFT: Fast Fourier Transform.

FIFO: First In First Out.

FM: Full-Map.

FS: Frequent Sharers.

FT: First-Touch.

GEMS: General Execution-driven Multiprocessor Simulator.

HTML: HyperText Markup Language.

IDCT: Inverse Discrete Cosine Transform.

IEEE: Institute of Electrical and Electronics Engineers.

ILP: Instruction Level Parallelism.

INSO: In-Network Snoop Ordering.

IPC: Instructions Per Cycle.

L1C$: First-Level Coherence Cache.

L2C$: Second-Level Coherence Cache.

LP: Limited Pointers.

LRU: Least Recently Used.

MESI: Modified Exclusive Shared Invalid (set of cache coherence states).

MOESI: Modified Owned Exclusive Shared Invalid (set of cache coherence states).

MOSI: Modified Owned Shared Invalid (set of cache coherence states).

MPEG: Moving Picture Experts Group.

MSHR: Miss Status Holding Register.

MSI: Modified Shared Invalid (set of cache coherence states).

MSSG: MPEG Software Simulation Group.

NoSC: No Sharing Code.

NUCA: Non Uniform Cache Architecture.

NUMA: Non Uniform Memory Access.

OLTP: OnLine Transaction Processing.

OS: Operating System.

RR: Round-Robin.

SCI: Scalable Coherent Interface.

SGI: Silicon Graphics, Inc..

SGML: Standard Generalized Markup Language.

SiCoSys: Simulator of Communication Systems.

SLICC: Specification Language for Implementing Cache Coherence.

TLB: Translation Lookaside Buffer.

TLP: Thread Level Parallelism.

VLC: Variable Length Coding.

VLSI: Very Large Scale of Integration.

VM: Victim Migration.

VR: Victim Replication.

VTC: Virtual Tree Coherence.

Chapter 0

Resumen

0.1 Introducción

Los continuos avances en la escala de integración permiten reducir cada vez más el tamaño de los transistores y, por tanto, cada vez podemos encontrar chips con un mayor número de transistores disponibles. Los fabricantes de chips han decidido dedicar estos transistores a aumentar el número de procesadores en lugar de a incrementar el rendimiento de un único procesador, tarea mucho más laboriosa y con menores beneficios en términos de rendimiento, dando lugar a los multiprocesadores en un único chip o CMPs (*Chip-multiprocessors*) [91]. Los CMPs tienen importantes ventajas sobre los procesadores superescalares. Por ejemplo, tienen un poder computacional agregado mucho mayor y consumen menos energía que un único procesador mucho más complejo.

Muchos CMPs actuales, como el IBM Power6 [63] o el Sun UltraSPARC T2 [104], tienen un número de procesadores relativamente pequeño (entre 2 y 8), cada uno de ellos con al menos un nivel de caches privadas. Estos procesadores se comunican entre sí a través de una red de interconexión ordenada (habitualmente, un bus o un *crossbar*) que se encuentra dentro del chip. Sin embargo, estas redes de interconexión ordenadas tienen problemas de escalabilidad cuando el número de procesadores que tienen que soportar es relativamente alto. Estos problemas se deben principalmente al área que necesitan para ser implementadas y a su elevado consumo de energía [59]. Además, según la Ley de Moore [83], todavía válida, cabe esperar que el número de procesadores integrados en un mismo chip se doble cada 18 meses [22], por lo que estas redes de interco-

nexión no serán adecuadas para futuros CMPs. Para solucionar estos problemas de escalabilidad surgen los *tiled CMPs*, CMPs que se construyen en base a replicar bloques idénticos o casi idénticos a lo largo y ancho del chip, conectándolos mediante una red de interconexión escalable y punto a punto. Estos bloques de construcción poseen su propio procesador, jerarquía de memoria e interfaz de red. De este modo, se consigue un coste de fabricación menor, ya que su diseño, más sencillo, consiste en replicar un mismo patrón sucesivas veces.

Por otro lado, la mayoría de los CMPs actuales siguen un modelo de programación de memoria compartida. Este modelo proporciona una programación más amigable para el usuario que el modelo de paso de mensajes, pero requiere un soporte eficiente para la coherencia de las caches. Aunque en las últimas décadas se le ha prestado mucha atención a los protocolos de coherencia de caches en el ámbito de los multiprocesadores tradicionales, los parámetros tecnológicos y nuevas restricciones de los CMPs implican la necesidad de buscar nuevas soluciones al problema de la coherencia de caches [22].

Los multiprocesadores que emplean redes de interconexión punto a punto, como los *tiled CMPs*, suelen implementar un protocolo de coherencia de caches basado en directorio. Desafortunadamente, estos protocolos tienen dos problemas fundamentales que limitan su escalabilidad: La alta latencia de los fallos de cache debido a la indirección al nodo *home* y la elevada sobrecarga de memoria necesaria para mantener la información de coherencia.

- La indirección se produce por la necesidad de obtener la información de directorio antes de realizar las acciones de coherencia correspondientes. Esta información se encuentra habitualmente en el nodo *home* de cada bloque. De este modo, ante un fallo de cache se accede, en primer lugar, al nodo *home*. Una vez que se obtiene la información de directorio, se reenvían las peticiones a los destinatarios correspondientes, los cuales responden al nodo que generó el fallo de cache. De este modo, muchos fallos de cache necesitan tres *saltos* en el camino crítico.

- La memoria necesaria para mantener la información de coherencia puede llegar a requerir un área desmesurada cuando el número de nodos del sistema aumenta considerablemente [15]. Esto ocurre especialmente cuando se usa un vector de bits para mantener la información acerca de los compartidores de cada bloque de memoria.

Aunque existen soluciones para mantener la coherencia de las caches que evitan la indirección y tratan de reducir la sobrecarga de memoria, como por

ejemplo *Token-CMP* [78], estas soluciones están basadas en difusión total (*broadcast*), es decir, en inundar la red de interconexión con mensajes de coherencia, lo cual eleva la contención y, lo que es más importante, el consumo de energía de la red, el cual puede llegar a alcanzar en algunos casos el 50% del consumo total del chip [71, 116]. En esta tesis se trata tanto el problema de la indirección como el de la sobrecarga de memoria, pero además, teniendo en cuenta al mismo tiempo, el consumo de la red de interconexión.

Por último, existe otra restricción que puede afectar a la escalabilidad de los futuros CMPs. El retardo que introduce la red de interconexión al enviar un mensaje de un extremo del chip a otro, puede alcanzar las decenas de ciclos [49, 10]. Este retardo provoca que el acceso a una cache lógicamente compartida pero físicamente distribuida dependa en gran medida de la distancia entre el procesador que quiere acceder al dato y el banco de la cache donde esté almacenado dicho dato. Es lo que comúnmente se conoce como caches de acceso no uniforme (NUCA, o ***N**on-**U**niform **C**ache **A**rchitecture* [57]). Por tanto, esta tesis también aborda el problema de la alta latencia de acceso a caches NUCA.

0.1.1 Contribuciones de la tesis

Las principales contribuciones de esta tesis son las siguientes:

- *Organización de directorio escalable.* Esta organización está basada en duplicar en el directorio los *tags* de los bloques almacenados en las caches privadas. Además, mediante el uso de un particular intercalado de los bloques de memoria en los diferentes bancos que componen el directorio, se obtiene una organización en la que el tamaño de cada banco de directorio no depende del número de nodos o *tiles* del sistema. Esta propiedad se cumple siempre y cuando el número de nodos del sistema sea menor o igual al número de conjuntos de las caches privadas.

- *Protocolo de directorio con reemplazos implícitos.* Este protocolo permite eliminar todo el tráfico generado por los reemplazos de bloques en las caches privadas, lo cual se traduce en una reducción del consumo de la red de interconexión. La idea consiste en solapar los mensajes generados por los reemplazos con los mensajes generados por las peticiones que causan dichos reemplazos. Este mecanismo requiere una organización de directorio similar a la descrita en el punto anterior, particularmente, el mismo intercalado de directorio.

- *Protocolos de coherencia directa.* Esta familia de protocolos evita el problema de la indirección en los protocolos de directorio, pero sin inundar la red con peticiones, tal y como ocurre en los protocolos basados en *broadcast*. La principal propiedad de los protocolos de coherencia directa es que tanto el manejo de la información de los compartidores de cada bloque como el mantenimiento de la coherencia de caches, la realiza el nodo propietario, es decir, el nodo que provee el bloque ante un fallo de cache. La indirección se evita enviando las peticiones directamente al nodo propietario, de ahí el nombre de *coherencia directa*.

- *Política de mapeo de caches sensible a la distancia y a la tasa de fallos.* Esta política intenta mapear páginas de memoria a los bancos de cache pertenecientes al nodo que más veces accede a dichas páginas, con el fin de reducir la latencia de acceso a los bloques contenidos en dicha página. Adicionalmente, esta política introduce una cota superior para diferenciar entre el número de páginas mapeadas a cada banco, con el objetivo de distribuir uniformemente las páginas entre los bancos de cache, y así reducir la tasa de fallos de dicha cache.

Todas las contribuciones que aparecen en la presente tesis han sido publicadas o están siendo consideradas para su publicación en conferencias internacionales [95, 96, 97, 98, 99, 100], revistas [102] o capítulos de libro [101].

0.2 Entorno de evaluación

A la hora de llevar a cabo la evaluación de las propuestas presentadas en esta tesis, hemos decidido hacer uso del simulador GEMS 1.3 [77]. GEMS es un simulador que extiende a Virtutech Simics [72] y modela de forma lo suficientemente detallada el sistema de memoria, así como una amplia gama de protocolos de coherencia de caches. Además, para modelar con más precisión el comportamiento de la red de interconexión hemos reemplazado el simulador de red que proporciona GEMS 1.3 por SiCoSys [94], un simulador de red más detallado, el cual hemos modificado para dar soporte al envío de mensajes *multicast*. Asimismo, hemos hecho uso de la herramienta CACTI [115] con el fin de precisar las latencias de acceso a las caches y de medir el área requerida por dichas estructuras. Para ello, hemos asumido que el tamaño de las direcciones físicas es de 40 bits y que usamos una tecnología de proceso de 45nm.

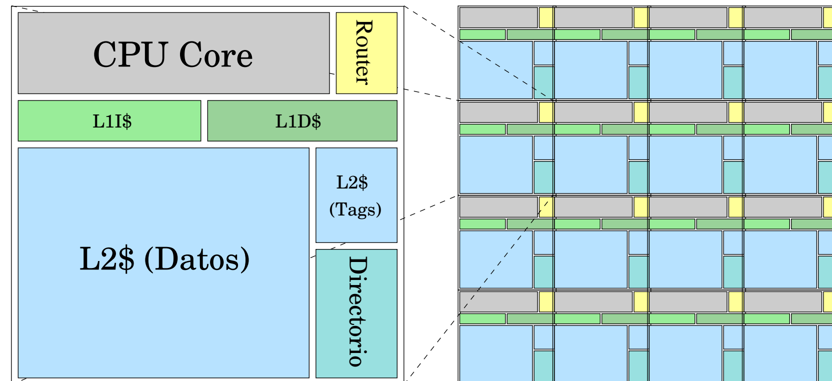


Figura 0.1: Organización de un *tile* y un *tiled* CMP de 4×4 celdas.

El sistema simulado en esta tesis es un *tiled* CMP de 16 nodos (o 32, en algunos casos) como el de la Figura 0.1. Cada *tile* contiene un procesador, o *core*, una cache de primer nivel privada para datos y otra para instrucciones, un banco o porción de la cache de segundo nivel compartida y una interfaz de red, o *router*, que conecta todos los nodos mediante una red de interconexión con topología de malla de dos dimensiones. El resto de parámetros del sistema simulado se pueden ver en la Tabla 0.2.

Todas las propuestas presentadas en esta tesis han sido implementadas utilizando para ello el simulador GEMS. Estas propuestas han sido chequeadas exhaustivamente para comprobar la ausencia de condiciones de carrera que pudieran provocar cualquier incoherencia, con el fin de validar su correcto funcionamiento.

Las diez aplicaciones científicas usadas en las simulaciones para evaluar las propuestas presentadas en esta tesis poseen una amplia variedad de patrones de computación y comunicación. Las aplicaciones junto con sus tamaños de entrada son las siguientes. *Barnes* (8192 cuerpos, 4 pasos), *FFT* (64K complejos), *Ocean* (océano de 130x130), *Radix* (512K claves, 1024 de radio), *Raytrace* (teapot), *Volrend* (head) y *Water-NSQ* (512 moléculas, 4 pasos) provienen del conjunto de aplicaciones SPLASH-2 [118]. *Unstructured* (Mesh.2K, 5 pasos) es una aplicación que modela el movimiento de fluidos [86]. *MPGdec* (525_tens_040.m2v) y *MPGenc* (salida de *MPGdec*) son aplicaciones multimedia que han sido obtenidas del conjunto de *benchmarks* ALPBench [66].

Aparte de estas aplicaciones paralelas, las cuales simulamos en un CMP con 16 nodos, también hemos simulado cargas multiprogramadas para un CMP con

Tabla 0.1: Parámetros del sistema simulado.

Parámetros de memoria (GEMS)	
Frecuencia del procesador	3GHz
Jerarquía de cache	No inclusiva
Tamaño de bloque	64 bytes
Cache L1 de datos e instrucciones	128KB, 4 vías
Tiempo de acceso a la L1	1 (tag) + 2 (datos) ciclos
Cache L2 compartida	1MB/celda, 8 vías
Tiempo de acceso a la L2	2 (tag) + 4 (datos) ciclos
Tiempo de acceso al directorio	2 ciclos
Tiempo de acceso a memoria	300 ciclos
Tamaño de página	4KB
Parámetros de red (SiCoSys)	
Frecuencia de red	1.5GHz
Topología	Malla de 2 dimensiones
Técnica de switching	Wormhole
Technique enrutamiento	X-Y determinista
Tamaño de los mensajes de control	1 flit
Tamaño de los mensajes de datos	4 flits
Tiempo de enrutamiento	1 ciclo
Tiempo de switch	1 ciclo
Latencia del enlace (un salto)	2 ciclos
Ancho de banda del enlace	1 flit/ciclo

32 nodos. En concreto, hemos creado cuatro cargas multiprogramadas. *Radix4* ejecuta cuatro instancias de *Radix* con 8 hilos cada una. Similarmente, *Ocean4* ejecuta cuatro instancias de *Ocean*. *Mix4* y *Mix8* simulan instancias de *Ocean*, *Raytrace*, *Unstructured* y *Water-Nsq*, con 8 y 4 hilos, respectivamente. *Mix8* ejecuta dos instancias de cada aplicación.

Finalmente, para obtener mayor precisión en los resultados, hemos realizado diversas pruebas para cada configuración y aplicación evaluada, las cuales insertan perturbaciones aleatorias en los accesos a memoria [12]. De este modo, los resultados presentados corresponden a la media de los valores obtenidos. Además, los resultados presentados en esta tesis corresponden a la fase paralela de las aplicaciones.

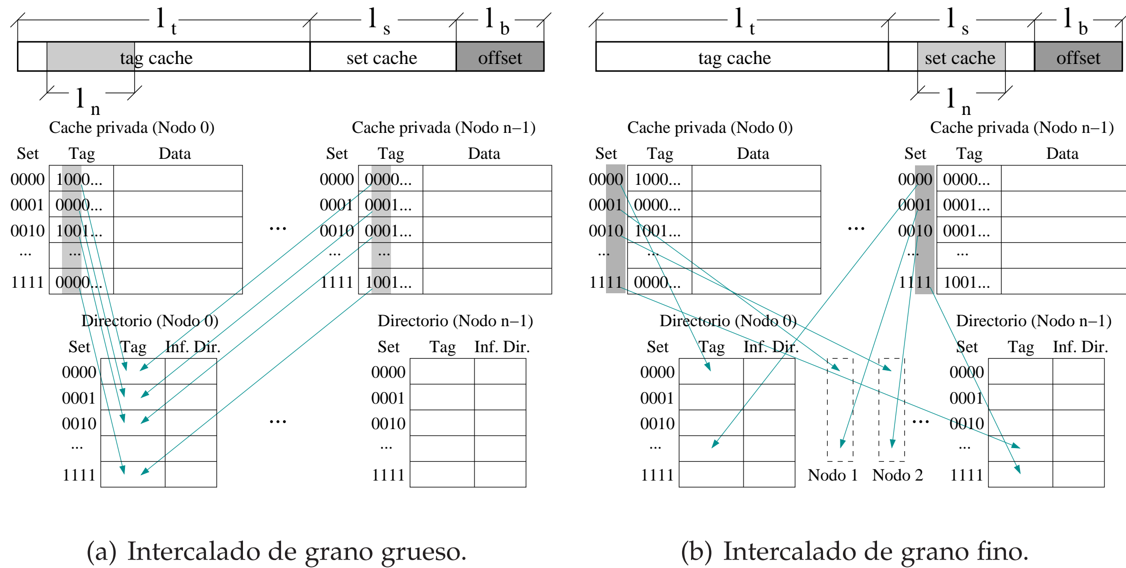
0.3 Una organización de directorio escalable

Como hemos apuntado en la Sección 0.1, los protocolos de coherencia de cache basados en directorio añaden a la implementación final del sistema una sobrecarga de memoria extra. Esta sobrecarga es debida al mantenimiento de la información acerca de los compartidores de cada bloque de memoria, es decir, la información de coherencia o información de directorio. Cuando esta información se organiza como un vector de bits, en el que cada bit indica la presencia o ausencia de un bloque de memoria en una determinada cache, la cantidad de memoria requerida por el directorio aumenta linealmente conforme aumenta el número de nodos. Es por ello por lo que numerosos autores han intentado reducir dicha sobrecarga de memoria a través de códigos de compartición comprimidos [1, 8, 26, 44, 85]. Sin embargo, estos códigos no logran una escalabilidad completa y a menudo introducen mensajes de coherencia extra que repercuten negativamente tanto en el rendimiento como en el consumo de energía del sistema.

En esta tesis proponemos una nueva organización de directorio que lo dota de escalabilidad [99], es decir, mantiene constante la memoria añadida a cada nodo para mantener la información de directorio, hasta un cierto número de procesadores. Además, esta organización almacena información precisa acerca de los compartidores. El tamaño de cada banco de directorio en la organización propuesta es $c \times (l_t + 2)$, donde c es el número de entradas de las caches privadas y l_t es el tamaño de la etiqueta, o *tag*, almacenado en la cache de directorio. Esta propuesta se detalla en la Sección 0.3.1.

Adicionalmente, partiendo de la organización de directorio diseñada, hemos propuesto un mecanismo de reemplazos implícitos [99]. Este mecanismo, que se describe en la Sección 0.3.2, permite eliminar la totalidad de los mensajes de coherencia causados por los reemplazos en las caches privadas, debido a que estos mensajes se manejan de forma implítita junto con el mensaje de petición que provoca cada reemplazo.

Los resultados, presentados en la Sección 0.3.3, muestran que la sobrecarga de área requerida por la organización de directorio escalable es de tan sólo 0.53% comparado con el área requerida por las caches de datos. Por otro lado, el mecanismo de reemplazos implícitos logra eliminar el 13% de los mensajes generados por un protocolo de directorio cuando los bloques en estado compartido no informan del reemplazo al directorio y es capaz de reducirlos hasta un 33%, de media, cuando se informa al directorio acerca de estos reemplazos.



(a) Intercalado de grano grueso.

(b) Intercalado de grano fino.

Figura 0.2: Granularidad del intercalado de directorio y efecto que causa en el tamaño mínimo necesario para almacenar toda la información de las caches privadas.

0.3.1 Directorio escalable

Esta sección describe un protocolo de directorio distribuido en el que cada banco de directorio tiene un tamaño fijo, que es independiente del número de nodos del sistema. Para ello, prestamos especial atención al intercalado del directorio, es decir, a qué banco de directorio mapea cada bloque de memoria. Al nodo donde se encuentra ese banco o cache de directorio se le denomina nodo *home* de dicho bloque. Normalmente, este mapeo se realiza tomando los $log_2 n$ (l_n) bits menos significativos de la dirección del bloque (sin contar con el desplazamiento, u *offset*, del bloque), donde n representa el número de nodos del sistema. Como muestra la Figura 0.2, dependiendo de la posición de los bits que se elijan para definir el nodo *home* de cada bloque, el número mínimo de entradas requerido por el directorio puede variar. De hecho, si se toman los bits más significativos de la dirección de memoria, se puede dar el caso de que todos los bloque almacenados en cache mapeen a un mismo *home*, por lo que se requeriría un número de entradas para el directorio de $n \times c$, y por tanto, el tamaño de dicho directorio dependería linealmente del número de nodos.

En cambio, si los bits tomados para asignar el nodo *home* están dentro de los bit usados para indexar las caches privadas l_s, nos aseguramos de que los

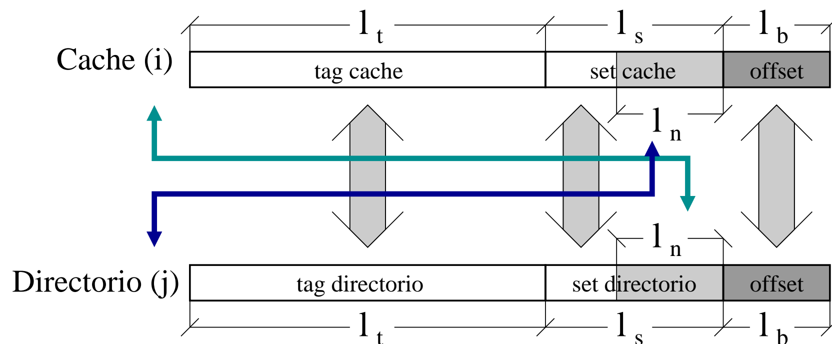


Figura 0.3: Mapeo entre entradas de cache y directorio.

bloques almacenados en las caches se distribuyan uniformemente entre los diferentes nodos *home*. De este modo, el número máximo de entradas requerido por cada banco de directorio sería c, valor que no depende del número de nodos, sino tan sólo del número de entradas de las caches privadas.

Por otro lado, para mantener constante su tamaño total necesitamos que, no sólo el número, sino también el tamaño de las entradas, sea independiente del número de nodos. Esto se consigue mediante el uso de *tags* duplicados, que han sido empleados en CMPs como el Piranha [16] o el Sun UltraSPARC T2 [111]. Estos *tags* duplicados mantienen en el directorio una copia de los *tags* almacenados en las caches privadas.

Una vez acotado el tamaño del directorio debemos buscar una función que mapee bloques a entradas de directorio, de tal forma que se cumplan las siguientes premisas:

1. Un bloque de memoria siempre debe tener su *tag* duplicado en el mismo banco de directorio, independientemente de la cache privada donde esté almacenado. Esta premisa asegura encontrar toda la información relativa a un mismo bloque en un único nodo.

2. La función debe ser inyectiva, es decir uno a uno. Esta premisa garantiza la escalabilidad en el número de entradas del directorio.

La Figura 0.3 muestra esta función. Los nodos *home* vienen definidos por un subconjunto de bits l_n de entre los usados para indicar el conjunto, o *set*, de la cache privada l_s donde el bloque debe ser almacenado. El conjunto donde se almacenará el *tag* duplicado dentro de ese banco de directorio viene dado por la

unión de los restante bits del campo l_s y el identificador del nodo que mantiene su copia en la cache privada. Estos últimos bits serán posteriormente usados para identificar los compartidores del bloque. Ya que los l_n bits deben ser un subconjunto de los l_s bits ($l_n \subseteq l_S$), este directorio mantiene su escalabilidad siempre y cuando se cumpla la regla:

$$num_nodos \leq num_conjuntos_en_caches_privadas.$$

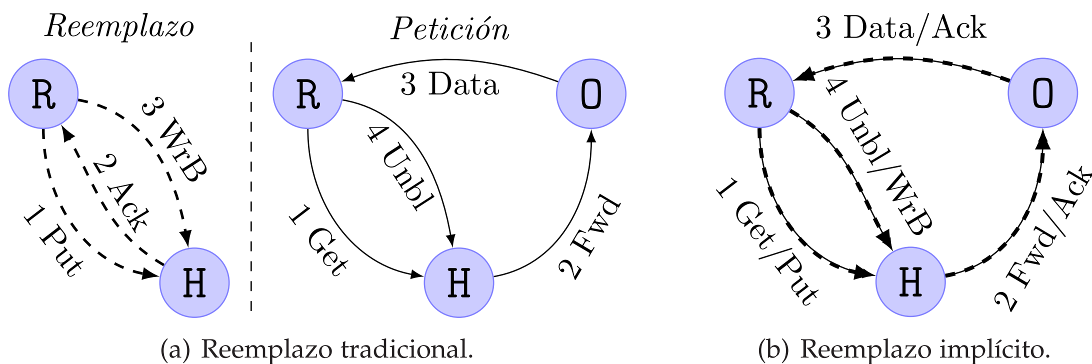
0.3.2 Reemplazos implícitos

En esta sección se describe el mecanismo de reemplazos implícitos, que consiste en manejar los reemplazos de forma implícita, uniendo los mensajes generados por ellos con los generados por las peticiones que provocan dichos reemplazos. La organización de directorio presentada anteriormente, asegura que ante el reemplazo de un bloque de una cache privada, se cumplan las dos características descritas a continuación, las cuales permiten implementar el mecanismo de reemplazos implícitos:

1. El bloque reemplazado y el bloque que causa su reemplazo siempre mapean al mismo banco de directorio. Esto se debe al intercalado utilizado, que toma los bits para definir el nodo *home* dentro del subconjunto de los bits usados para seleccionar el conjunto donde se almacena el bloque en las caches privadas.

2. Cada entrada de las caches privadas tiene asociada una única entrada en el directorio (una única vía también), y viceversa. De este modo cuando la petición que genera el reemplazo llega al directorio, este conoce tanto la dirección del dato solicitado como la del reemplazado. Por tanto, no es necesario indicar ambas direcciones en el mensaje, sino solamente la del bloque solicitado y el número de la vía donde estaba el bloque reemplazado, por lo que el tamaño de los mensajes no se incrementa considerablemente.

La Figura 0.4 muestra el mecanismo de reemplazos implícitos comparándolo con el mecanismo de reemplazos tradicional. Normalmente, cuando se produce un fallo de cache, previamente se debe reemplazar otro bloque almacenado en el mismo conjunto con el fin de dejar espacio para el nuevo bloque. Los reemplazos se suelen hacer en tres fases para evitar condiciones de carrera difíciles de tratar. Primero, la cache pide permiso de reemplazo al nodo *home* (*1 Put*). Cuando el nodo *home* confirma el reemplazo (*2 Ack*), el bloque es enviado al siguiente nivel



(a) Reemplazo tradicional.

(b) Reemplazo implícito.

Figura 0.4: Diferencias entre los reemplazos tradicionales e implícitos.

de cache (*3 WrB*). Por otro lado, las peticiones de bloque que fallan en cache solicitan el bloque al nodo *home* (*1 Get*). Esta solicitud es reenviada al nodo propietario (*2 Fwd*), el cual envía los datos al peticionario (*3 Data*). Una vez resuelto el fallo, se envía un mensaje (*4 Unbl*) al nodo *home* indicándole que ya puede procesar otras peticiones para ese bloque.

Mediante el mecanismo de reemplazos implícitos se solapan estos mensajes, tal y como se muestra en la Figura 0.4(b). Ante un fallo, se almacena tanto la dirección del bloque reemplazado como la del solicitado en el MSHR (*Miss Status Hold Register* o registro de fallos pendientes), y se envía la petición (*1 Get/Put*) al nodo *home*. Del mismo modo, el nodo *home* almacena ambas direcciones (la del reemplazo la obtiene del directorio) en su MSHR y reenvía al propietario del bloque solicitado la petición (*2 Fwd/Ack*). Cuando el causante del fallo recibe el mensaje con los datos y la confirmación del reemplazo (*3 Data/Ack*) ambas entradas del MSHR se liberan y se procede al reemplazo del bloque, a la vez que se desbloquea el nodo *home* (*4 Unbl/WrB*). Finalmente, el nodo *home* libera análogamente ambas entradas de su MSHR.

0.3.3 Resultados

En esta sección mostramos los resultados obtenidos para las dos propuestas presentadas previamente. En primer lugar veremos como la organización de directorio escalable es capaz de reducir la sobrecarga de memoria del protocolo de coherencia. Después, mostraremos las reducciones en el número de mensajes de coherencia obtenidas por el mecanismo de reemplazos implícitos.

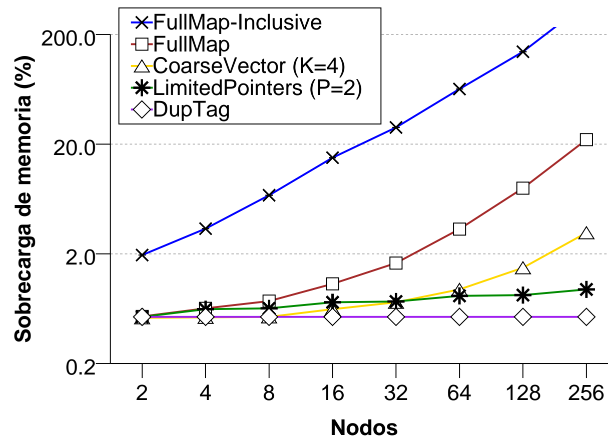


Figura 0.5: Sobrecarga del área de directorio en mm^2 en función del número de nodos.

0.3.3.1 Sobrecarga de memoria

La Figura 0.5 compara diversas organizaciones de directorio. En particular, muestra la sobrecarga que introduce el área requerida por la información de directorio comparada con el área requerida por las caches de datos. Esta sobrecarga se muestra en función del número de nodos del sistema, desde 2 hasta 256 nodos.

Las organizaciones representadas en la gráfica son: *FullMap-Inclusive*, donde la información de directorio está incluida en los *tags* de la cache L2; *FullMap*, que usa una cache de directorio con el mismo número de entradas que la cache L1 y con un vector de bits, o *full-map*, en cada entrada; *CoarseVector (K=4)*, que usa la cache de directorio anterior comprimiendo el código de compartición usando un bit para representar cuatro nodos [44]; *Limited pointers (P=2)*, que comprime el código de compartición usando dos punteros para los dos primeros compartidores [26]; y *DupTag*, que es la organización propuesta en esta sección.

Podemos ver que las organizaciones que usan un código de compartición *full-map* no escalan con el número de nodos del sistema. Las otras propuestas que usan códigos de compartición comprimidos, tampoco escalan completamente, y como veremos en la siguiente sección introducen tráfico extra en la red debido a que almacenan información imprecisa. Por último, nuestra propuesta, no sólo almacena información precisa sino que también escala perfectamente hasta 256 nodos. La sobrecarga de esta organización es de 0.53% para todas las configuraciones mostradas.

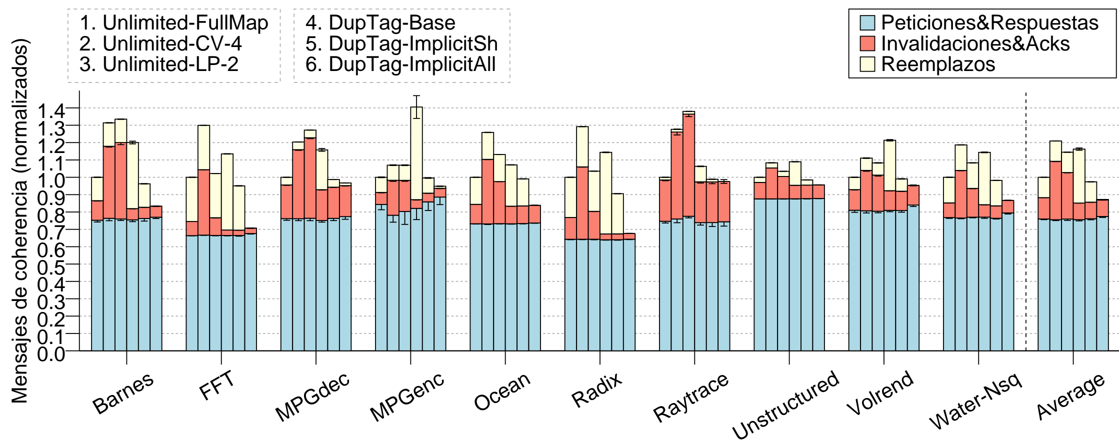


Figura 0.6: Reducción en el número de mensajes de coherencia.

0.3.3.2 Tráfico de red

La Figura 0.6 muestra la variación en el número de mensajes generados por el protocolo de coherencia, normalizado respecto a un protocolo de directorio que usa caches de directorio ilimitadas con un código de compartición *full-map* (*Unlimited-FullMap*). Además mostramos resultados para *Unlimited-CV-4* y *Unlimited-LP-2* que representan protocolos con caches de directorio ilimitadas y con códigos de compartición comprimidos (*coarse vector* [44] y *limited pointers* [26], respectivamente). Por último, mostramos resultados para protocolos que usan nuestra organización de directorio escalable, usando reemplazos tradicionales tanto de bloques compartidos como modificados (*DupTag-Base*), haciendo implícitos los reemplazos compartidos (*DupTag-ImplicitSh*), y realizando todos los reemplazos de forma implícita (*DupTag-ImplicitAll*).

En primer lugar podemos observar que el uso de códigos de compartición comprimidos incrementa el tráfico en la red, debido a un aumento en el número de invalidaciones enviadas por cada fallo. Por otro lado, el uso de *tags* duplicados requiere la notificación de reemplazos en estado compartido, lo que incrementa el tráfico debido a los reemplazos. Los reemplazos implícitos eliminan completamente este tráfico, reduciendo así el tráfico de red en un 13% comparado con un protocolo que no informa de los reemplazos de bloques en estado compartido y hasta un 33% cuando sí se informa de estos reemplazos.

0.4 Protocolos de coherencia directa

Otro problema importante de los protocolos de directorio es la indirección en el acceso al nodo *home*. El acceso al nodo *home* para obtener la información de directorio causa elevadas latencias en los fallos de cache, degradando el rendimiento final del sistema. Por otro lado, existen otros protocolos que eliminan esta indirección, como por ejemplo *Token-CMP*, pero que a cambio generan una cantidad de tráfico muy importante, especialmente cuando el número de nodos del sistema es elevado.

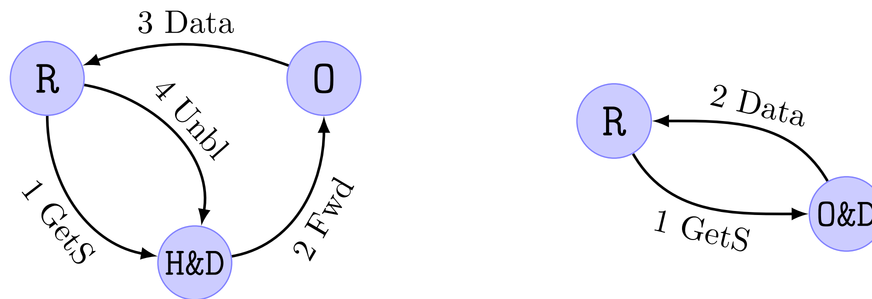
Esta tesis presenta los protocolos de coherencia directa, una nueva familia de protocolos de coherencia de cache que eliminan la indirección para la mayoría de los fallos de cache, tal y como ocurre con *Token-CMP*, pero enviando la petición a un único destinatario, como hacen los protocolos de directorio. En particular, proponemos y evaluamos un protocolo basado en el concepto de coherencia directa para *tiled* CMPs que llamamos *DiCo-CMP*.

Los protocolos de coherencia directa, explicados en la Sección 0.4.1, asignan la tarea de almacenar la información de directorio y de ordenar las peticiones de los diferentes procesadores sobre un mismo bloque de memoria a la cache que provee el bloque en cada fallo. El nodo que contiene esa cache se llama nodo *owner* o propietario. La indirección se evita enviando la petición de fallo de cache directamente al nodo propietario, de ahí el nombre de coherencia directa. De este modo, la coherencia directa reduce la latencia de los fallos de cache respecto a un protocolo de directorio, ya que envía la petición directamente al nodo que debe proveer el dato, a la vez que reduce el tráfico en la red respecto a *Token-CMP*, puesto que la petición es enviada a un único destinatario.

Además, dado que la organización de directorio escalable propuesta en la sección anterior no puede ser utilizada para organizar el código de compartición en los protocolos de coherencia directa, hemos estudiado el uso de diversos códigos comprimidos para estos protocolos (Sección 0.4.2), con el fin de reducir no sólo la indirección y el tráfico en la red, sino también la sobrecarga de memoria requerida por estos protocolos.

Los resultados, presentados en la Sección 0.4.3, muestran que *DiCo-CMP* reduce el tiempo de ejecución de las aplicaciones en un 9% (media de todas las aplicaciones) comparado con un protocolo de directorio, y un 8% comparado con *Token-CMP*. Además, reduce el tráfico en la red de interconexión, y por tanto el consumo de energía, hasta en un 37% comparado con *Token-CMP*.

Por otro lado, el uso de códigos de compartición comprimidos junto con los protocolos de coherencia directa ofrecen un buen compromiso entre tráfico de



(a) Protocolos de directorio.

(b) Protocolos de coherencia directa.

Figura 0.7: Cómo se resuelven los fallos de transferencia de cache a cache en un protocolo de directorio y en un protocolo de coherencia directa. R=*Requester* o peticionario; H=*Home*; D=Directorio; O=*Owner* o propietario.

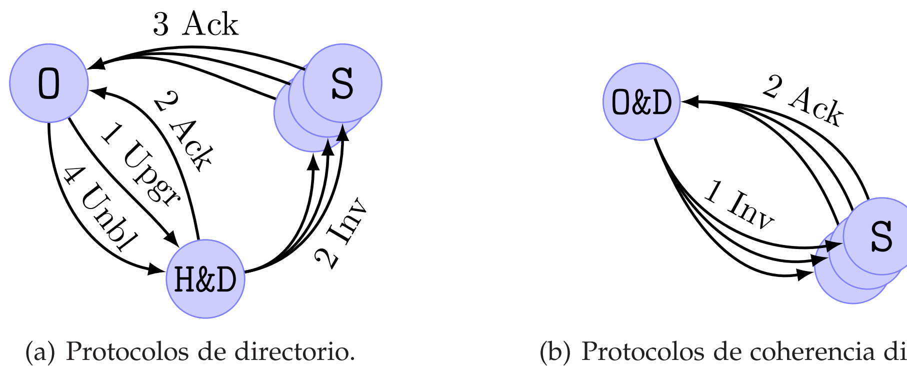
red y área requerida, sin degradar sustancialmente el tiempo de ejecución. En particular, un código de compartición basado en árboles binarios [1] (*DiCo-BT*) obtiene importantes reducciones en la sobrecarga de memoria respecto a *DiCo-CMP*, reduciendo además su orden de crecimiento de $O(n)$ a $O(log_2 n)$, a cambio de incrementar ligeramente el tráfico en la red en un 9%.

0.4.1 La coherencia directa

En los protocolos de directorio, la coherencia de cache es mantenida por el nodo *home* y todos los fallos de cache son enviados a dicho nodo, el cual debe redirigir la petición al nodo propietario. Esto introduce el problema de la indirección, que provoca fallos de cache con mayor latencia. En contrapartida, la coherencia directa almacena la información de coherencia en el nodo propietario de cada bloque y asigna la tarea de mantener la coherencia a dicho nodo. De este modo la indirección se puede evitar si el nodo que comete el fallo envía la petición directamente al propietario del bloque en lugar de al nodo *home*.

Las figuras 0.7 y 0.8 comparan cómo se resuelven dos tipos distintos de fallos de cache tanto en un protocolo de directorio como en los protocolos de coherencia directa, indicado los números el orden en el que se generan los mensajes. En estas figuras se pueden apreciar las principales ventajas de los protocolos de coherencia directa sobre los protocolos de directorio, que son las siguientes.

- Consiguen que un mayor número de fallos se resuelvan en solamente dos saltos, en lugar de en tres (el mensaje *Unbl*, que aparece en los protocolos



(a) Protocolos de directorio.

(b) Protocolos de coherencia directa.

Figura 0.8: Cómo se resuelven los fallos por actualización (*upgrades*) en un protocolo de directorio y en un protocolo de coherencia directa. O=*Owner* o propietario; H=*Home*; D=Directorio; S=*Sharers* o compartidores.

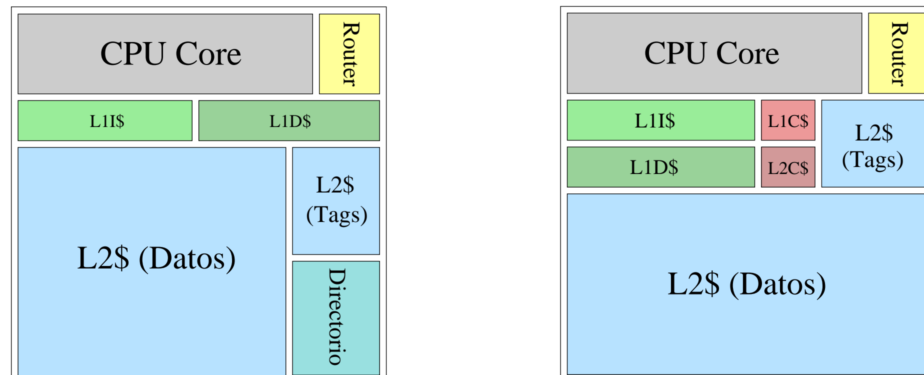
de directorio, se encuentra fuera del camíno crítico del fallo), reduciendo así su latencia.

- Eliminan los mensajes de comunicación entre el nodo propietario y el nodo *home* ya que el mantenimiento de la coherencia de caches y el envío de los datos se realizan ahora en un único nodo.

- Reducen el tiempo de espera de las otras peticiones al mismo bloque, como consecuencia de eliminar el tiempo que se pierde en la comunicación entre el nodo propietario y el nodo *home*. Aunque este tiempo se encuentra fuera del camino crítico del fallo, sí que afecta a los siguientes fallos sobre el mismo bloque.

Para implementar los protocolos de coherencia directa se requieren una serie de cambios en la organización de los nodos del sistema. En primer lugar, toda cache que pueda ser propietaria de un bloque debe ser capaz de almacenar el código de compartición de dicho bloque. Por tanto, es necesario ampliar la parte de las etiquetas (*tags*) de las caches de primer nivel con un nuevo campo para almacenar los compartidores (las caches de segundo nivel ya incluyen este campo en un protocolo de directorio). En cambio, la coherencia directa no necesita almacenar esta información en la estructura de directorio del nodo *home*.

Al contrario que en un protocolo de directorio, en la coherencia directa el nodo que mantiene la coherencia no es fijo ya que los fallos de escritura pueden provocar que el nodo propietario cambie. Se necesita, por tanto, un mecanismo para poder identificar al nodo propietario en cualquier momento. Para ello, es



(a) Organización de un nodo para protocolos de directorio.

(b) Organización de un nodo para protocolos de coherencia directa.

Figura 0.9: Modificaciones requeridas por los protocolos de coherencia directa en la estructura de un *tile*.

necesario añadir dos estructuras que almacenan información que identifica al nodo propietario ($log_2 n$ bits), como se puede apreciar en la parte derecha de la Figura 0.9:

- *Cache de coherencia de primer nivel* (*L1C$*): Esta cache almacena información sobre los nodos propietarios de un conjunto de bloques y es accedida por el procesador que genera la petición al producirse un fallo de cache. Si la información sobre el propietario del bloque solicitado está almacenada en la cache, se enviará la petición directamente a dicho nodo. Esta cache se puede actualizar de diversas maneras dependiendo del tráfico de red que se desee generar, tal y como se explicará más adelante.

- *Cache de coherencia de segundo nivel* (*L2C$*): Esta cache es necesaria debido a que el nodo encargado de mantener la coherencia puede cambiar durante la ejecución de las aplicaciones. En esta cache se almacena la identidad actual del nodo propietario para todos los bloques almacenados en cualquier cache privada. La información almacenada debe ser actualizada en cada cambio de propietario mediante mensajes de control.

0.4.1.1 Evitando la inanición

En los protocolos basados en directorio se consigue evitar la inanición encolando las peticiones en el nodo encargado de ordenarlas (el nodo *home*) y procesán-

dolas en orden FIFO (*first in, first out*). Sin embargo en los protocolos de coherencia directa el nodo que ordena las peticiones es el nodo propietario, el cual puede cambiar ante una petición de escritura. Debido a esta situación, una petición puede tardar algún tiempo en encontrar el nodo propietario, incluso aunque la petición la envíe el nodo *home*. Además, una vez encontrado el propietario este puede moverse a otro nodo debido a una petición anterior, y por tanto sería necesario tener que encontrarlo de nuevo. Esto puede provocar inanición, es decir que unas peticiones se resuelvan rápidamente mientras que otras permanezcan buscando al propietario mucho más tiempo.

Para solucionar este problema proponemos detectar y evitar la inanición de un modo muy simple. Cada vez que una petición llega al nodo *home* se incrementa un contador incluido en el mensaje. Cuando este contador alcanza un cierto umbral, se marca la petición como en estado de inanición y se procede a procesarla lo antes posible. Siempre que se detecte una petición en estado de inanición se impide que el propietario del bloque solicitado por dicha petición cambie de un nodo a otro. De esta manera se garantiza que el nodo *home* sea capaz de encontrar el propietario y resolver el fallo.

0.4.1.2 Actualizando la cache de coherencia de primer nivel

Como comentábamos, los protocolos de coherencia directa usan la cache de coherencia de primer nivel (*L1C$*) para evitar la indirección al nodo *home*. Esta cache almacena infomación imprecisa acerca del nodo propietario de un conjunto de bloques y puede ser actualizada de diversas maneras.

Una primera política consitiría en almacenar la identidad del último procesador que invalidó o proporcionó cada bloque. Este procesador será el propietario mientras otro no solicite permiso de escritura para ese bloque o sea reemplazado por otro bloque. A esta política la hemos llamado *Base* y no introduce tráfico extra en la red de interconexión.

Otra política alternativa, que incrementa el tráfico en la red mediante el uso de mensajes de control extra, consistiría en enviar *hints* o *pistas* a modo de mensajes de control informando del cambio de propietario a un conjunto de compartidores frecuentes, por ejemplo, los procesadores que han solicitado el bloque con anterioridad o han fallado en la predicción. A esta política la hemos llamado *Hints*. Particularmente, hemos implementado dos políticas de *hints*. La primera de ellas, *Hints-FS* (*Frequent Sharers*), añade un código de compartición junto al que ya acarrea el protocolo, el cual almacena la identidad de todos los procesadores que han solicitado cada bloque. De este modo, ante un cambio de

propietario, se envían mensajes de *hint* a todos estos nodos informando acerca del cambio. Como esta propuesta requiere una cantidad de memoria extra considerable (la adición de un nuevo código de compartición), hemos propuesto un mecanismo de *hints* basado en *address signatures* [25, 121] (*Hints-AS*). Cada *signature* almacena las direcciones de los bloques para las cuales se ha predicho incorrectamente el nodo propietario. Cuando estos bloques cambian de propietario se envían *hints* a todos los núcleos del sistema. Este mecanismo mantiene información menos precisa que el anterior y, por consiguiente, genera un poco más de tráfico a cambio de reducir significativamente el área requerida.

0.4.2 Reducción de la sobrecarga de memoria

Como hemos comentado en la sección anterior, *DiCo-CMP* necesita dos estructuras que mantienen información acerca de propietario de cada bloque. Puesto que cada entrada de estas estructuras almacena solamente el *tag* y un puntero al propietario ($log_2 n$ bits), tienen un orden de crecimiento bajo –$O(log_2 n)$–. El problema aparece con el código de compartición añadido a cada entrada de las caches de datos. Si este código es un *full-map*, el protocolo dejaría de ser escalable en cuanto al área necesaria para implementarlo. Es por ello que proponemos el uso de códigos de compartición comprimidos en DiCo-CMP.

En concreto, hemos evaluado DiCo-CMP con los códigos de compartición mostrados en la Tabla 0.2, en la que se presentan, además, el número de bits requeridos por cada campo de la información de coherencia del protocolo y el orden de crecimiento total en cuanto al área. Como todos los protocolos de coherencia directa precisan de las caches de coherencia, el menor orden de crecimiento será $O(log_2 n)$.

Tabla 0.2: Bits requeridos para almacenar la información de coherencia.

Protocolo	**Código de Compartición**	**Bits cache L1 y cache L2**	**Bits L1C$ y L2C$**	**Orden**
DiCo-FM	*Full-map*	n	$log_2 n$	$O(n)$
DiCo-CV-K	*Coarse vector*	$\frac{n}{K}$	$log_2 n$	$O(n)$
DiCo-LP-P	*Limited pointers*	$1 + P \times log_2 n$	$log_2 n$	$O(log_2 n)$
DiCo-BT	*Binary Tree*	$\lceil log_2(1 + log_2 n) \rceil$	$log_2 n$	$O(log_2 n)$
DiCo-NoSC	Ninguno	0	$log_2 n$	$O(log_2 n)$

DiCo-FM implementa un código de compartición preciso (*full-map*). *DiCo-CV-K* reduce el tamaño del código de compartición usando un *coarse vector* [44], en el que cada bit representa un grupo de *K* nodos, en lugar de sólo uno. Aunque reduce el tamaño, su orden de crecimiento sigue siendo $O(n)$. Concretamente usamos un valor de *K* igual a 2. *DiCo-LP-P* usa un esquema de punteros limitados (*limited pointer* [26]), en el que se almacenan un número limitado de punteros que representan a los primeros compartidores. Cuando el número de compartidores es mayor que el número de punteros se envían las invalidaciones a todos los nodos del sistema. Este esquema crece en orden $O(log_2 n)$. Concretamente usamos un valor de *P* igual a 1. *DiCo-BT* usa un código de compartición basado en un árbol binario, o *binary tree* [1], en el que los nodos son conceptualmente agrupados de manera recursiva en grupos de dos elementos, formando así un árbol binario. La información que se almacena en el código de compartición consiste en el menor nivel del árbol que abarca todos los compartidores. Este código de compartición requiere sólo 3 bits en un CMP de 16 nodos. Por último, *DiCo-NoSC* (*no sharing code*) no emplea ningún código de compartición y por tanto envía las invalidaciones a todos los nodos del sistema ante un fallo de escritura para un bloque con más de un compartidor. Aunque es el esquema que más tráfico genera, tiene la ventaja de que no requiere modificar la estructura de las caches de datos.

0.4.3 Resultados

En esta sección comparamos DiCo-CMP con otros protocolos existentes tanto en la literatura como en los multiprocesadores comerciales: *Hammer* [92], *Directorio* [38] y *Token* [78]. Sus características, junto con las de los protocolos de coherencia directa (*DiCo*), están resumidas en la Tabla 0.3. En particular, en esta sección mostramos los resultados para la implementación de DiCo-CMP que usa un mecanismo de *hints* por medio de *address signatures* y que emplea los diferentes tipos de códigos de compartición presentados en la sección anterior.

En primer lugar mostramos resultados referentes a la sobrecarga de memoria, después las reducciones en el tráfico de red y, por último, el impacto final en el tiempo de ejecución de las aplicaciones.

0.4.3.1 Sobrecarga de memoria

La Figura 0.10 muestra la sobrecarga de memoria de los protocolos evaluados. Esta sobrecarga se muestra variando el número de nodos del sistema desde 2

Tabla 0.3: Resumen de los protocolos evaluados.

	Broadcast	**Indirección**
Hammer	Sí	Sí
Token	Sí	No
Directorio	No	Sí
DiCo	No	No

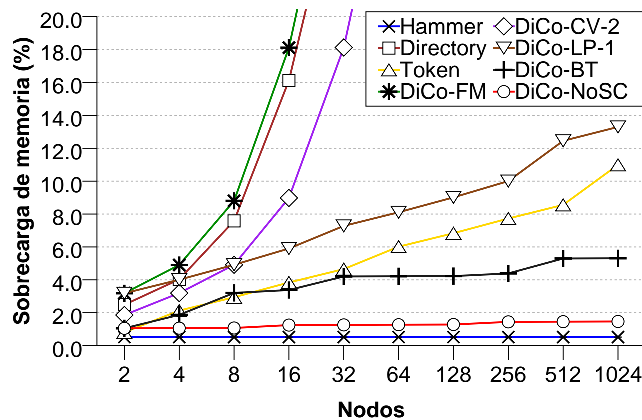


Figura 0.10: Sobrecarga del área de directorio en mm^2 en función del número de nodos.

hasta 1024. *Hammer* es el protocolo que menos sobrecarga tiene, ya que no mantiene ninguna información acerca de los compartidores. El protocolo de directorio (*Directory*) almacena la información tanto en los *tags* de la cache L2 como en las caches de directorio. Como consideramos un código *full-map* para este protocolo, se requieren n bits por entrada y, por tanto, su tamaño crece linealmente con el número de nodos. *Token* mantiene el conteo de los tokens que posee cada bloque junto con dicho bloque. Por tanto añade a las caches L1 y L2 un nuevo campo de $\lceil log_2(n+1) \rceil$ bits para contar tanto los *non-owner* tokens como el *owner* token. En este protocolo, el área requerida escala de forma aceptable.

Aunque vemos que *DiCo-CMP* es el protocolo que más sobrecarga introduce cuando usa un código de compartición *full-map* (*DiCo-FM*), si se consideran otros códigos de compartición comprimidos, esta sobrecarga se puede reducir

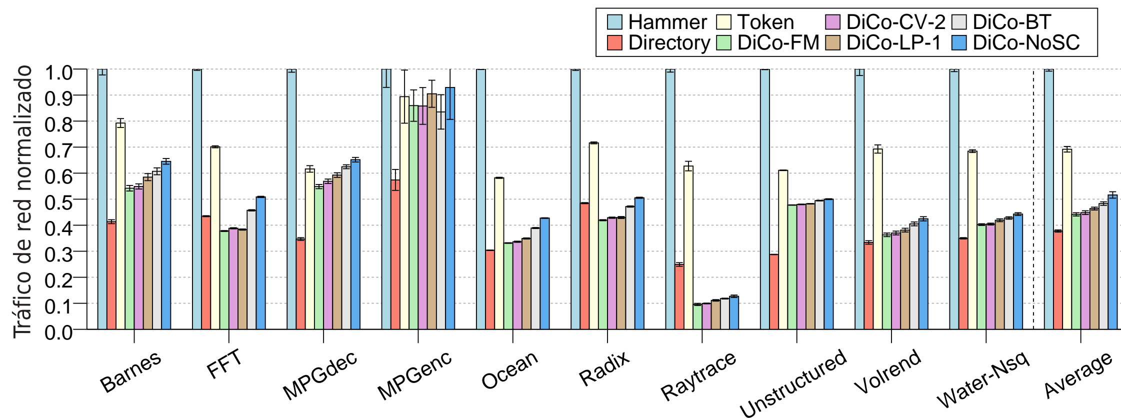


Figura 0.11: Tráfico de red normalizado.

de forma significativa. Así, vemos que *DiCo-CV-2* requiere menos área, pero es todavía poco escalable. En cambio, *DiCo-LP-1*, que añade sólo un puntero a cada entrada de cache, tiene mejor escalabilidad. *DiCo-BT* reduce más el área requerida, por debajo incluso de las necesidades de *Token*. Por último, *DiCo-NoSC*, que no necesita modificaciones en las caches, obtiene resultados similares a *Hammer*.

0.4.3.2 Tráfico de red

La Figura 0.11 compara el tráfico de red generado por todos los protocolos evaluados. Cada barra muestra el número de bytes transmitidos por cada *switch* de la red de interconexión, normalizado respecto al protocolo *Hammer*.

Como cabía esperar, *Hammer* es el protocolo que más tráfico genera, ya que no almacena ninguna información acerca de los compartidores y, por consiguiente, necesita inundar la red con invalidaciones o peticiones de bloque ante la mayoría de los fallos de cache. Además, todas estas peticiones requieren una respuesta, que es enviada por cada nodo de manera independiente. *Directory* reduce considerablemente el tráfico inyectado por el protocolo de coherencia, ya que mantiene información precisa acerca de los compartidores. *Token* genera más tráfico que *Directory*, pero menos que *Hammer*, ya que los nodos que no poseen ningún token no tienen que responder a los mensajes de petición. *DiCo-FM* incrementa el número de mensajes comparado con *Directory* debido al envío de *hints* para mejorar las predicciones del nodo propietario y, así, el tiempo de ejecución final de las aplicaciones.

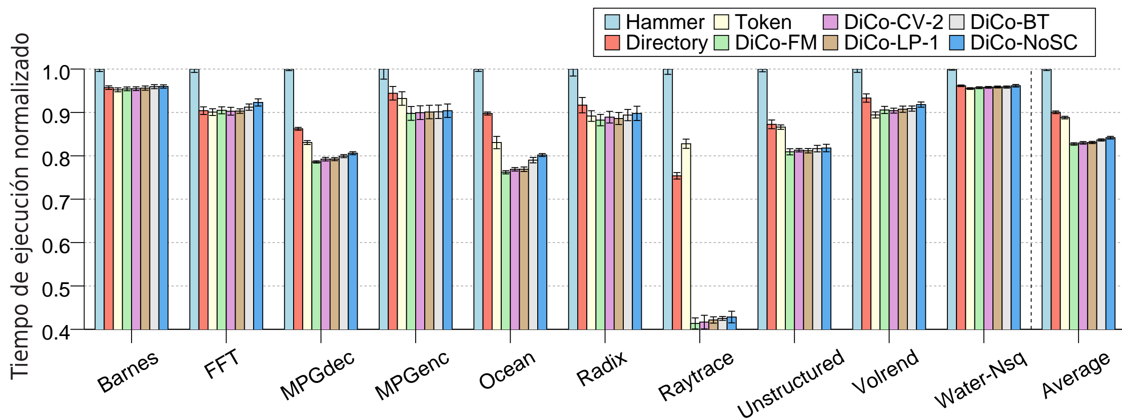


Figura 0.12: Tiempo de ejecución normalizado.

En general podemos ver que al usar un código de compartición más comprimido, el número de mensajes inyectados en la red se va aumentando. Así, *DiCo-LP-1*, *DiCo-BT* y *DiCo-NoSC* incrementan el tráfico en un 5%, 9% y 19% comparado con *DiCo-FM*, respectivamente. Incluso *DiCo-NoSC*, que no necesita modificaciones en la estructura de las caches de datos, obtiene un tráfico aceptable, reduciendo los números obtenidos por *Token* en un 25%.

0.4.3.3 Tiempo de ejecución

La Figura 0.12 muestra los resultados en tiempo de ejecución para las aplicaciones evaluadas en esta tesis. Estos resultados han sido normalizados respecto a los obtenidos por el protocolo *Hammer*.

Podemos observar que *Directory* mejora el rendimiento de *Hammer* para todas las aplicaciones, debido a las reducciones en tráfico y en latencia, ya que hay que esperar un menor número de confirmaciones. *Token* mejora ligeramente los números obtenidos por *Directory* (1%), ya que evita la indirección, pero a su vez incrementa el tráfico en la red, lo que puede incrementar su congestión.

Finalmente, los protocolos de coherencia directa obtienen mejoras significativas en tiempo de ejecución. *DiCo-FM* mejora en un 17%, 9% y 8% los tiempos obtenidos por *Hammer*, *Directory* y *Token*, respectivamente. Por otro lado, el uso de códigos de compartición comprimidos no incrementa considerablemente el tiempo de ejecución de la mayoría de las aplicaciones. En particular, *DiCo-NoSC*, que es el protocolo de coherencia directa que peor tiempo de ejecución obtiene, sólo empeora respecto a *DiCo-FM* en un 2% de media.

0.5 Política de mapeo de caches sensible a la distancia y a la tasa de fallos

Otro problema de aparece en los *tiled* CMPs de gran escala es la alta latencia de acceso al nivel de cache compartida, en nuestro caso, la cache L2. Dado que en un *tiled* CMP esta cache se encuentra físicamente distribuida entre los diferentes nodos del sistema (cache NUCA), su tiempo de acceso dependerá de lo cerca que se encuentre el nodo *home* del procesador que solicita el bloque de datos.

Cuando se usa una política de mapeo de caches en las que los bits menos significativos de la dirección del bloque (sin contar el *offset* de bloque) definen el nodo *home* de dicho bloque, los bloques se reparten entre los diferentes bancos de cache de un modo cíclico *round-robin*, tal y como se lleva a cabo tanto en CMPs actuales [63, 104] como en la literatura reciente [52, 123]. Esta distribución de bloques no se preocupa en absoluto de la distancia entre los bancos de cache donde se mapean los bloques y los procesadores que más frecuentemente acceden a ellos.

Otra forma de distribuir los bloques en las caches, propuesta por Cho y otros [34], consiste en definir el nodo *home* usando bits más significativos de la dirección del bloque, en concreto, un subconjunto de los bits que definen el número de página física. De este modo, cuando el sistema operativo realiza la traducción de página virtual a física, se puede elegir un número de página física tal que el mapeo de dicha página corresponda al mismo nodo que solicitó por primera vez el bloque. A esta política se le llama *first-touch*. La mejor granularidad para esta política es la de página, ya que es la más fina de las que posibilita esta política. Por tanto, se deben coger para la elección del nodo *home* los bits menos significativos sin contar con el *offset* de página. El problema de esta política consiste en que si la carga de trabajo de cada procesador no está bien balanceada, habrá unos bancos de cache con muchas más páginas que otros, con lo que se limitaría la capacidad total de la cache compartida y se incrementaría la tasa de fallos de dicha cache, que normalmente es la última en la jerarquía de memoria del chip y cause accesos costosos. Esto ocurre muy a menudo en los servidores que ejecutan varias instancias de aplicaciones, con requerimientos de memoria completamente distintos, en un mismo CMP. Por tanto, también evaluaremos este tipo de situaciones mediante cargas multiprogramadas.

Como solución a este problema proponemos una política de mapeo de caches, controlada por el sistema operativo, que sea sensible a la distancia y a la tasa de fallos. Esta política, llamada *DARR* (*distance-aware round-robin*), es ex-

plicada en la Sección 0.5.1. *DARR* intenta mapear páginas a los bancos de cache que son locales al procesador que realiza la primera petición para un bloque de esa página, pero a la vez intenta balancear la distribución de las páginas en los diferentes bancos de cache. De este modo se reduce la latencia de acceso a la cache sin incrementar excesivamente la tasa de fallos.

Sin embargo, las políticas manejadas por el sistema operativo que tienen en cuenta esta distancia pueden incrementar la tasa de fallos de las caches privadas si no se tiene cuidado. Esto ocurre cuando los bits que definen el nodo *home* son los mismos que definen el conjunto de las caches privadas donde se almacena cada bloque, tal y como se explica en el Sección 0.5.2. Para solucionar este problema es necesario cambiar los bits usados para indexar las caches privadas, evitando usar los mismos bits que identifican al nodo *home*.

Los resultados mostrados en la Sección 0.5.3 indican que mediante la combinación de estas dos técnicas se pueden obtener mejoras en tiempo de ejecución de un 5% para aplicaciones paralelas y de un 14% para cargas multiprogramadas respecto a un política *round-robin*, reduciendo el tráfico en la red en un 31% con aplicaciones paralelas y un 68% usando cargas multiprogramadas. Comparado con una política *first-touch* se obtienen mejoras de un 2% para aplicaciones paralelas y de un 7% para cargas multiprogramadas, incrementando ligeramente el tráfico en la red.

0.5.1 Mapeo sensible a la distancia y a la tasa de fallos

El algoritmo de mapeo que proponemos reduce la distancia media de acceso a la cache L2 sin incrementar considerablemente su tasa de fallos. Además, otra ventaja importante de este mecanismo es que no necesita ningún hardware extra para ser implementado.

El encargado de manejar este mecanismo de mapeo es el sistema operativo. Ante un fallo de página, el sistema operativo necesita asignar una dirección física a la página accedida. Nuestro algoritmo elige una dirección física de tal modo que mapee al banco de cache local al procesador que está accediendo al bloque que provocó el fallo de página, tal y como se realizaría en una política *first-touch*. El sistema operativo posee un contador para cada banco de cache que le informa del número de páginas mapeadas a cada una de ellas. Al mapear una nueva página a un banco de cache, su contador se incrementa.

Para garantizar la distribución uniforme de páginas a bancos de cache, se define un umbral que limita la máxima diferencia de número de páginas mapeadas entre dos bancos cualquiera. Cuando un banco alcanza este umbral, ya

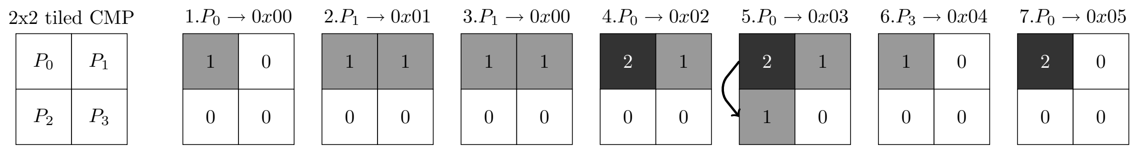


Figura 0.13: Ejemplo de la política de mapeo propuesta.

no puede alojar más páginas y tiene que delegar el mapeo a un banco vecino, es decir, a un banco que está a un salto según la topología de la red de interconexión. Si los bancos vecinos también han alcanzado el valor umbral, la página se intentará mapear a un banco que esté a dos saltos, y así sucesivamente hasta encontrar un banco que pueda alojar la página. Cuando el contador de todos los bancos es mayor que cero, se decrementan en una unidad todos los contadores, con el fin de que nunca lleguen todos los bancos al umbral. Este umbral define el comportamiento del algoritmo. Si es muy bajo, su comportamiento será similar a una política *round-robin*. Si por el contrario es muy alto, el algoritmo funcionará de manera similar a una política *first-touch*.

La Figura 0.13 muestra, de izquierda a derecha, el comportamiento de este algoritmo para un *tiled* CMP de 2×2 nodos con un valor umbral de dos. En primer lugar, el procesador P_0 accede a un bloque de la página $0x00$, la cual no se encuentra en memoria y provoca un fallo de página (1). El sistema operativo elige una dirección física para esta página que mapee al banco perteneciente al nodo 0. Además, el contador de este nodo se incrementa. Después, el procesador P_1 accede a la página $0x01$ y se realiza una operación análoga a la anterior (2). Cuando el procesador P_1 accede a la página $0x00$ no se realiza ninguna acción ya que no se produce un fallo de página (3). Posteriormente, el procesador P_0 accede a una nueva página, que se almacena en el banco 0, el cual alcanza el valor umbral (4). Cuando este procesador vuelve a acceder a otra página, esta deberá mapear a otro banco vecino, en este caso el local a P_2 ya que posee el valor más bajo (5). Más tarde, el procesador P_3 accede a una página nueva, que puede mapear a su banco local. Como ahora todos los valores son mayores que cero, se decrementan todos en una unidad (6). De esta manera el procesador P_0 puede volver a alojar una nueva página en su banco local (7).

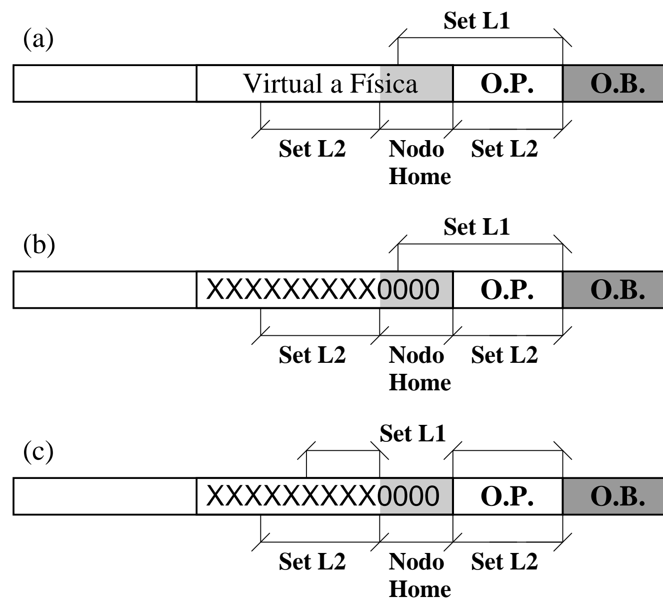


Figura 0.14: Cambios en los bits de indexación de las caches (O.P.=*Offset* de página, O.B.=*Offset* de bloque).

0.5.2 Cambio en los bits de indexación de las caches

Como comentábamos, las políticas que son manejadas por el sistema operativo de la forma propuesta por Cho y otros [34] que además son sensibles a la distancia pueden tener un efecto negativo en la tasa de fallos de las caches privadas. La Figura 0.14(a) muestra tanto el indexado usado por las caches privadas (la cache L1 en nuestro caso) como por la compartida (la cache L2). Decíamos que es importante usar la menor granularidad de intercalado posible, con el fin de obtener una mayor reducción en la distancia media de acceso a los bancos de la cache L2. Además, los bits usados para indexar las caches privadas suelen ser los menos significativos, como se muestra en la Figura 0.14(a). Cuando estos bits se sobreponen, se puede provocar un incremento de la tasa de fallos de las caches L1, debido al siguiente motivo.

En la Figura 0.14(b) se puede apreciar que una política sensible a la distancia intenta mapear los bloques pedidos por el procesador P_0 al banco 0, cambiando estos bits al elegir la dirección física. De este modo, la mayoría de los bloques accedidos por este procesador van a tener esos bits a 0, y por tanto caerán todos en los mismos conjuntos de la cache, provocando así un incremento de los fallos por conflicto.

Este problema se acentúa más cuanto más agresiva es la política en términos de reducción de distancia. Por tanto, proponemos evitar estos bits y usar los siguientes menos significativos, como se muestra en la Figura 0.14(c). Este cambio mejora la utilización de las caches L1 y, por consiguiente, la tasa de aciertos y el tiempo de ejecución de las aplicaciones. Por esta razón, en la evaluación de nuestra política de mapeo y de la *first-touch* usamos este mecanismo de indexado para las caches L1.

0.5.3 Resultados

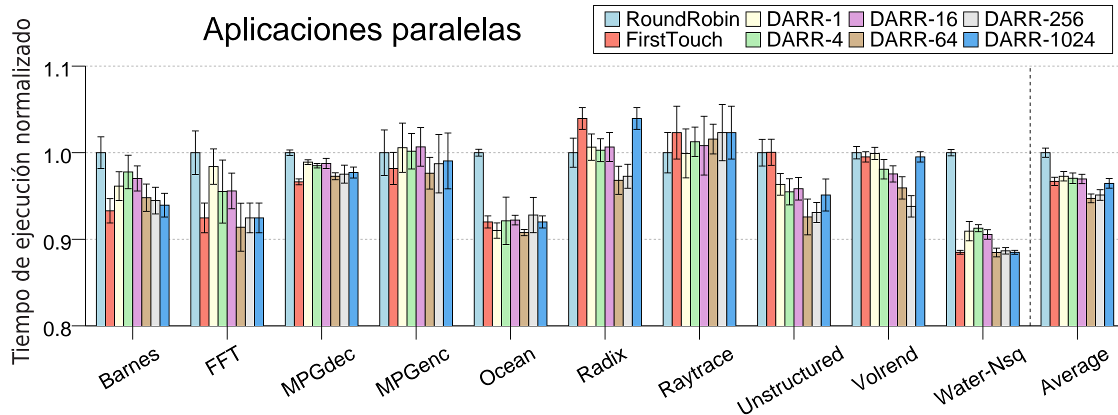
Debido a que en los servidores de aplicaciones es muy frecuente encontrar desbalanceo en el conjunto de trabajo de los procesadores y el mecanismo propuesto intenta realizar un balanceo adecuado de la memoria usada por el sistema, añadimos a la evaluación de esta tesis cargas multiprogramadas, las cuales evaluamos teniendo en cuenta un CMP con 32 nodos.

En primer lugar mostramos el tiempo de ejecución de las aplicaciones para las dos políticas base explicadas anteriormente (*round-robin* y *first-touch*) y nuestra propuesta con diferentes valores para el umbral (entre 2^0 y 2^{10}). Del mismo modo, mostramos las diferencias de tráfico de red para todas estas políticas.

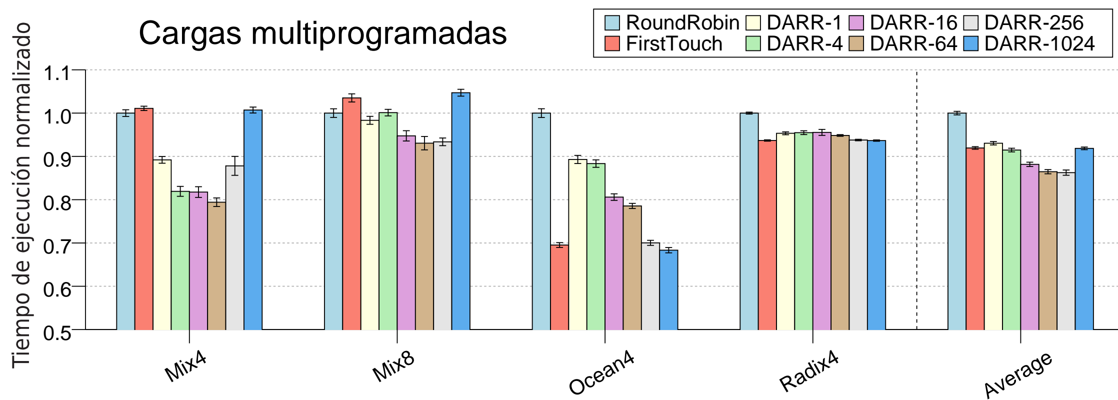
0.5.3.1 Tiempo de ejecución

La Figura 0.15 muestra los resultados en tiempo de ejecución normalizados con respecto a la política *round-robin*. La política *first-touch* mejora los tiempos obtenidos por *round-robin* para aplicaciones con homogeneidad en el conjunto de trabajo accedido por los diferentes procesadores que forman el sistema, como es el caso de *Barnes*, *FFT*, *Ocean*, *Water-Nsq*, *Ocean4* y *Radix4*. Por el contrario, cuando esta distribución es heterogénea, como sucede en *Radix*, *Raytrace*, *Mix4* y *Mix8*, la política *first-touch* sufre un degradación en el rendimiento debido a un aumento en la tasa de fallos de la cache L2. Podemos ver que nuestra propuesta (*DARR*) obtiene un rendimiento óptimo para valores del umbral entre 64 y 256.

De este modo se obtienen mejoras de un 5% de media para las aplicaciones paralelas y un 14% para las cargas multiprogramadas comparado con una política *round-robin*. Si nos comparamos con la política *first-touch* obtenemos un 2% de mejora para las aplicaciones paralelas y un 7% para las cargas multiprogramadas.



(a) Aplicaciones paralelas (16 nodos).

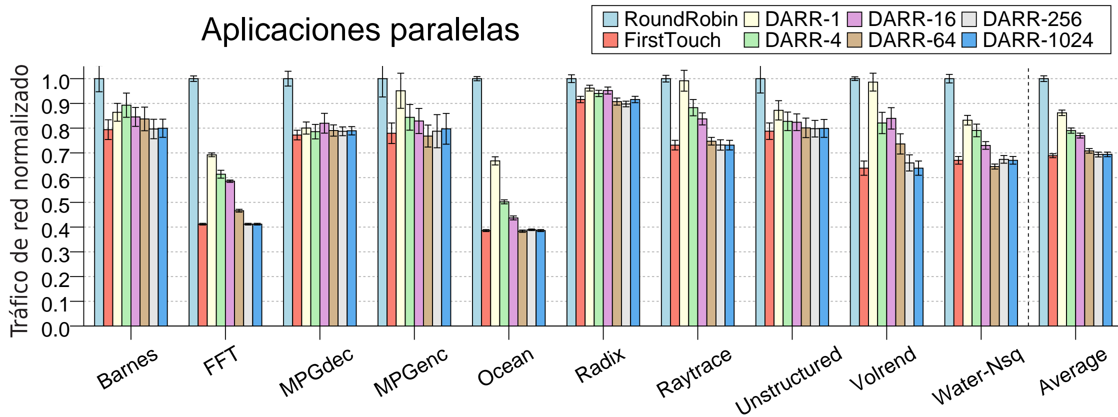


(b) Cargas multiprogramadas (32 nodos).

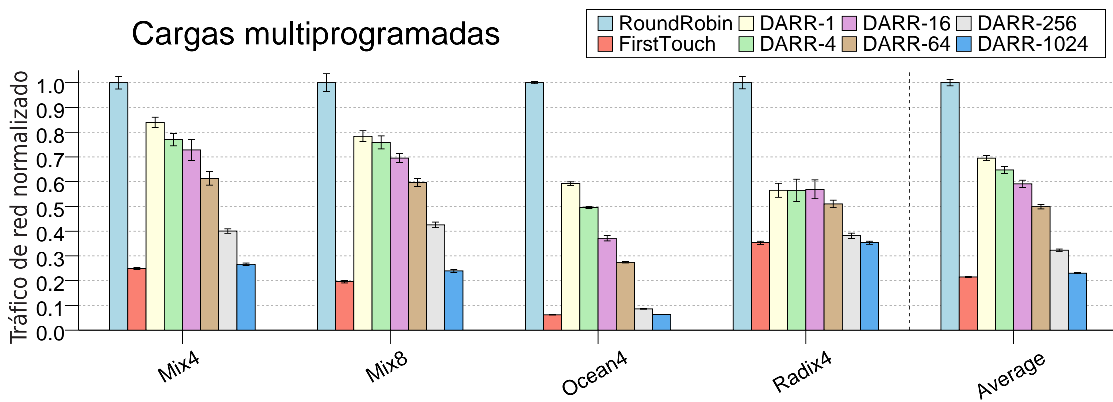
Figura 0.15: Tiempo de ejecución normalizado.

0.5.3.2 Tráfico de red

La Figura 0.16 compara el tráfico de red para las políticas evaluadas en esta sección. Los resultados están normalizados con respecto a la política *round-robin*, que es la que más tráfico genera, ya que no tiene en cuenta la distancia. Por otro lado, podemos apreciar que el tráfico puede ser tremendamente reducido cuando se implementa una política *first-touch*. Estas reducciones son de un 34% considerando aplicaciones paralelas y hasta 79% para cargas multiprogramadas,



(a) Aplicaciones paralelas (16 nodos).



(b) Cargas multiprogramadas (32 nodos).

Figura 0.16: Tráfico de red normalizado.

ya que la mayoría de los bloques sólo son accedidos por una pequeña región del chip.

La política que proponemos (*DARR*) reduce el tráfico respecto a *round-robin* en todos los casos, hasta para un valor umbral de 1. Al aumentar este valor las reducciones se acentúan aún más. Podemos observar que para un valor umbral de 256, que obtiene un buen rendimiento en tiempo de ejecución, el tráfico generado se reduce un 31% para aplicaciones paralelas y un 68% para cargas multiprogramadas comparado con una política *round-robin*. Obviamente

la política *first-touch* reduce el tráfico respecto a nuestra política, pero a cambio de incrementar la tasa de fallos de la cache L2.

0.6 Conclusiones y vías futuras

Las arquitecturas *tiled* CMP han surgido recientemente como una alternativa escalable a los CMPs con un reducido número de nodos y probablemente los futuros CMPs tengan este diseño. Por otro lado, el mantenimiento de la coherencia de cache en los CMPs actuales se realiza usando técnicas similares a las usadas en los multiprocesadores tradicionales. Sin embargo, las nuevas restricciones de área y consumo de los CMPs demandan soluciones innovadoras para problema de la coherencia.

En esta tesis hemos presentado diversas propuestas que se centran en mejorar el rendimiento y escalabilidad de los futuros CMPs a través de la optimización de los diferentes mecanismos que afectan a la coherencia de las caches. En particular, hemos propuesto una organización de directorio distribuido escalable, un mecanismo de reemplazos que reduce el número de mensajes de coherencia emitidos por el protocolo, una nueva familia de protocolos de coherencia que evitan la indirección al nodo *home* a la vez que generan poco tráfico de red, y una política de mapeo para caches NUCA, lógicamente compartida pero físicamente distribuida, que mejora su latencia de acceso sin incrementar la tasa de fallos. Todas estas propuestas mejoran sensiblemente las prestaciones de los CMPs, ya sea en términos de tiempo de ejecución, tráfico de red o área demandada por el protocolo de coherencia, que constituyen las restricciones más destacadas que impondrán los futuros CMPs.

Pero el trabajo de esta tesis no acaba aquí, ya que deja abiertas numerosas vías futuras en el ámbito de la coherencia de caches. Por ejemplo, sería interesante el estudio de los protocolos coherencia directa con redes de interconexión heterogéneas [32], ya que la política de *hints* incrementa el tráfico en la red con mensajes que están fuera del camino crítico del fallo. Por otro lado, la optimización de los protocolos de coherencia directa para servidores de aplicaciones podría traer importantes beneficios ya que la información de directorio de cada bloque siempre estaría en la región abarcada por la aplicación que accede al bloque, en contra de lo que ocurre en un protocolo de directorio. Finalmente, el estudio de los bits más convenientes para seleccionar el conjunto donde se almacena un bloque en cache podría suponer mejoras en su tasa de aciertos.

Chapter 1

Introduction and Motivation

Computers have evolved rapidly in the last decades. Advancements in semiconductor manufacturing technologies, which allow for increased clock rates and larger number of transistors in a single chip, have contributed to a great extent to this evolution. Gordon E. Moore stated in 1965 that the number of transistors per silicon area doubled every eighteen months due to the transistors getting smaller every successive process technology [83]. This prediction, commonly referred to as Moore's Law, is still regarded as true.

Performance improvements in microprocessors have been partly caused by the increase of the frequency at which the processor works. However, manufacturers also have to organize the growing number of transistors in an efficient way to improve even more computers performance. In the past, hardware optimizations focused on increasing the amount of work performed in each cycle, e.g., by executing multiple instructions simultaneously (instruction-level parallelism or ILP). These optimizations led to deeply pipelined, highly speculative, out-of-order processors with large on-chip cache hierarchies.

However, these optimizations made processors more complex up to the point that obtaining slightly performance improvements requires an important number of extra transistors. Empirically, performance improvements have been close to the square root of the number of required transistors [47]. Differently, both area requirements and power consumption grow linearly with the number of transistors. Since the existing techniques for achieving more ILP cannot obtain significant performance improvements and physical constrains make increasing

the frequency impractical, it seems that the end of the road for uniprocessor microarchitectures is arriving [10].

Nowadays, billions of transistors are available in a single chip [81], and the most efficient way of organizing this vast number of transistors is to integrate multiple processing cores in the same chip. These multi-core or chip multiprocessors (CMPs) [91] try to improve system performance by exploiting thread-level parallelism (TLP) while avoiding the technology issues of complex monolithic processors by implementing multiple simpler processing cores. CMPs have important advantages over very wide-issue out-of-order superscalar processors. In particular, they provide higher aggregate computational power, multiple clock domains, better power efficiency, and simpler design. Additionally, the use of simpler cores reduces the cost of design and verification in terms of time and money. On the other hand, most contemporary CMPs are implemented based on the well-known shared-memory model which is expected to be maintained in the future [65]. Recent examples of these CMPs are, among others, the 2-core IBM POWER6 [63] and the 8-core Sun UltraSPARC T2 [104]. These current CMPs have a relatively small number of cores (2 to 8), every one with at least one level of private cache. These cores are typically connected through an on-chip shared network (e.g., a bus or a crossbar).

Now that Moore's Law will make it possible to double the number of processing cores per chip every 18 months [22], CMP architectures that integrate tens of cores (usually known as many-core CMPs) are expected for the near future. In fact, Intel recently unveiled the 80-core Polaris prototype [15]. In these many-core CMPs, elements that could compromise system scalability are undesirable. One of such elements is the interconnection network. As stated in [59], the area required by a shared interconnect as the number of cores grows has to be increased to the point of becoming impractical. Hence, it is necessary to turn to a scalable interconnection network.

Particularly, tiled CMP architectures [113, 123], which are designed as arrays of identical or close-to-identical building blocks known as *tiles*, are the scalable alternative to current small-scale CMP designs and will help in keeping complexity manageable. In these architectures, each tile is comprised of a processing core (or even several cores), one or several levels of caches, and a network interface or router that connects all tiles through a tightly integrated and lightweight point-to-point interconnection network (e.g., a two-dimensional mesh). Differently from shared networks, point-to-point interconnects are suitable for many-core CMPs because their peak bandwidth and area overhead scale with the number of cores. Figure 1.1 shows the organization of a tile (left) and

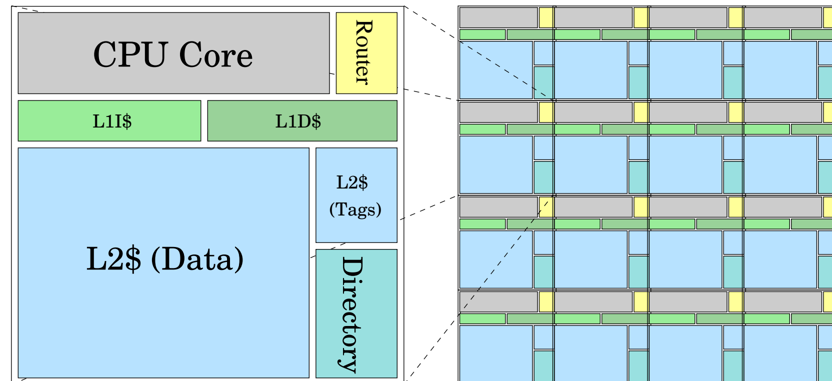


Figure 1.1: Organization of a tile and a 4×4 tiled CMP.

a 16-tile CMP (right). Tiled CMPs can easily support families of products with varying number of tiles, including the option of connecting multiple separately tested and speed-binned dies within a single package. Therefore, it seems that they will be the choice for future many-core CMPs.

This thesis assumes a tiled CMP model similar to the one depicted in Figure 1.1 both for explaining the proposed ideas and for evaluating them.

1.1 The cache coherence problem

Although CMP architectures are very suitable for throughput computing [29] where several independent programs run in different cores, the success of CMPs also relies on the ability of programmers and programs to extract thread-level parallelism. Shared-memory multiprocessors are quite popular since the communication among the different cores that conform the machine occurs implicitly as a result of conventional memory access instructions (i.e., loads and stores), which makes them easier to program than message-passing multiprocessors. In this way, all processing cores can read and write to the same shared address space and they will eventually see changes done by other processors, according to a particular memory consistency model [6].

Therefore, most CMP systems provide programmers with the intuitive shared-memory programming model, which is very familiar to them. In fact, most parallel software in the commercial market relies on this programming model. Hence, we think that shared-memory multiprocessors will remain the dominant architecture for a long time.

Although processing cores logically access the same address space, CMPs usually include one or several levels of caches which are private to each core. On-chip cache hierarchies are crucial to avoid the increasing gap between processor and memory speeds (the well-known memory wall problem [120]). Small and, therefore, faster caches can capture the great majority of data accesses due to the temporal and spatial locality of memory accesses exhibited by applications. The fast access to these caches reduces the average memory access latency. Unfortunately, the implementation of the shared-memory programming model is complicated by the existence of private caches.

Since processors store data in their private caches to take advantage of the locality of memory accesses, several copies of those memory blocks are held in different caches at the same time. Therefore, if a processing core modifies a data block stored in its local private cache and the other cores do not notice it, they could be accessing different values for the same data block, resulting in data incoherence. Caches are made transparent to software through a cache coherence protocol. They ensure that writes to shared memory are eventually made visible to all cores and also that writes to the same memory location appear to be seen in the same order by all processors [41], even in presence of private caches. This implies the exchange of data and access permissions among cores by means of coherence messages that travel across the interconnection network. Supporting cache coherence in hardware, however, requires important engineering efforts, and performance of future parallel applications will depend upon the efficiency of the protocol employed to keep cache coherence.

Although a great deal of attention was devoted to cache coherence protocols in the last decades in the context of traditional shared-memory systems comprised of multiple single-core processors (e.g., cc-NUMA machines [38]), the technological parameters and power constrains entailed by CMPs demand new solutions to the cache coherence problem [22]. This thesis addresses among other things possible solutions to this problem in this new context.

In general, there are several approaches to solve the cache coherence problem in hardware. Snooping-based cache coherence protocols [38] typically rest on one or several buses to broadcast coherence messages to all processing cores. In this way, coherence messages go directly from the requesting cores to those ones whose caches hold a copy of the corresponding memory block, which obtains optimal cache miss latencies. Unfortunately, snooping-based protocols require the total order property of the interconnection network to serialize requests from different cores to the same memory block. Since point-to-point interconnects do

not guarantee total order of messages, they are not suitable for implementing snooping-based cache coherence protocols.

Some proposals have focused on using snooping-based protocols with unordered networks. Martin *et al.* [76] present a technique that allows snooping-based protocols to utilize unordered networks by adding logical timing to coherence requests and reordering them at the destinations to establish a total order. Later on, Martin *et al.* also propose *TokenB* coherence protocol [75] that avoids the need of a totally ordered network by means of both token counting and arbitration mechanisms. Although these proposals enable low-latency cache-to-cache transfer misses, as snooping-based protocols, by using unordered interconnection networks, they also rely on broadcasting coherence actions, which increases network traffic exponentially with the number of cores. This increase in network traffic results in higher power consumption in the interconnection network, which has been previously reported to constitute a significant fraction (approaching 50% in some cases) of the overall chip power [71, 116]. Therefore, the traffic requirements of broadcast-based protocols restricts their scalability, and other solutions to the coherence problem are required for large-scale CMPs.

Directory-based cache coherence protocols [24, 38] have been typically employed in large-scale systems with point-to-point unordered networks (as tiled CMPs are). In these protocols, each memory block is assigned to a *home* tile. The home tile keeps the directory information (i.e., track of sharers) for its memory blocks and acts as serialization point for the different requests issued by several cores for the same block. When a cache miss takes place, the request is sent to the corresponding home tile, which determines when the request must be processed and, then, performs the coherence actions that are necessary to satisfy the cache miss. These coherence actions mainly involves forwarding the request to the cache that must provide the data block and sending invalidation messages to all sharers in case of a write miss. Since the directory information is stored at the home tile, invalidation messages are only sent to the tiles that must observe them (i.e., the tiles sharing the requested block), thus reducing network traffic compared to broadcast-based protocols.

Unfortunately, directory-based cache coherence protocols have two main issues that limit their efficiency and scalability: the indirection of cache misses and the directory memory overhead.

- The indirection problem is introduced by directory-based protocols because every cache miss must reach the home tile before any coherence action can be performed. This indirection to the home tile adds unnec-

essary hops into the critical path of the cache misses, finally resulting in longer cache miss latencies compared to broadcast-based protocols, since they directly send requests to sharers.

- The directory memory overhead required by directory-based protocols to keep the track of sharers for each memory block could be intolerable for large-scale configurations. The most common way of organizing the directory information is through a full-map or bit-vector sharing code (one bit per core). In this case, the area requirements of the directory structure grow linearly with the number of cores. For example, for a 64-byte (512 bits) block size, directory memory overhead for a system with 512 tiles is 100%, which is definitely prohibitive.

In this thesis we address these two problems separately. Particularly, we design a novel family of cache coherence protocols aimed at avoiding the indirection without relying on broadcasting requests. On the other hand, we study how a directory organization based on duplicate tags can be fully scalable up to a certain number of tiles.

1.2 Cache hierarchy organization

The increasing number of transistors per chip will be employed for allocating more processing cores, as for example in the 8-core Sun UltraSPARC T2 [104], but also for adding more cache storage, as happens, for example, in the dual-core Intel Montecito [81] in which each core has its own 12MB L3 cache. Since very large monolithic on-chip caches are not appropriate due to growing access latency, power consumption, wire resistivity, thermal cooling, and reliability considerations, caches designed for CMPs are typically partitioned into multiple smaller banks.

On the other hand, an important decision when designing CMPs is how to organize the multiple banks that comprises the last-level on-chip cache, since cache misses at this level result in long-latency off-chip accesses. The two common ways of organizing this cache level, are *private* to the local core or *shared* among all cores [68]. The private cache organization, implemented, for example, in Intel Montecito [81] and AMD Opteron [56], has the advantage of offering a fast access to the cache. However, it has two main drawbacks that lead to an inefficient use of the aggregate cache capacity. First, local banks keep a copy of the blocks requested by the corresponding core, potentially replicating blocks

in multiple cache banks. Second, load balancing problems can appear when the working set accessed by the cores is heterogeneously distributed among threads, i.e., some banks may be over-utilized whilst others are under-utilized.

The shared cache organization logically manages all cache banks as a single shared cache. This organization is implemented by several commercial CMPs, such as IBM POWER6 [63], Sun UltraSPARC T2 [104] and Intel Merom [103]. The main perk of the shared design is that it achieves more efficient use of the aggregate cache capacity (1) by storing only one copy of each block and (2) by uniformly distributing data blocks across the different banks.

However, wire delay of future CMPs will cause cross-chip communications to reach tens of cycles [49, 10]. Therefore, the access latency to a multibanked shared cache will be dominated by the wire delay to reach each particular cache bank rather than the time spent accessing the bank itself. In this way, the access latency to the shared cache can be drastically different depending on the cache bank where the requested block maps. The resulting cache design is what is known as non-uniform cache architecture (NUCA) [57]. The main downside of this organization is the long cache access latency (on average), since it depends on the bank wherein the block is allocated, i.e., the home bank or tile for that block. This thesis also addresses long NUCA latencies through a novel cache mapping policy.

1.3 Thesis contributions

This thesis presents several contributions aimed at addressing the three main constraints of cache coherence protocols for CMPs: area requirements, power consumption and execution time. Particularly, we reduce area requirements by minimizing the memory overhead of the directory structure in directory-based protocols. Power consumption in the interconnection network is decreased either by removing some coherence messages or by lowering the distance between the source and the destination tiles. Finally, execution time is improved by reducing the average cache miss latency. This reduction can be obtained by avoiding the indirection of directory-based cache coherence protocols, by preventing expensive off-chip accesses, or by reducing the average distance from the requesting core to the home tile.

The main contributions of this thesis are:

- A *scalable distributed directory organization* based on both managing the directory information as duplicate tags of the blocks stored in the private

caches and using a particular interleaving for the mapping of memory blocks to the different banks of the distributed shared cache. This directory organization can scale (i.e., its memory overhead does not increase with the number of cores) up to a certain number of cores. The maximum number of cores allowed depends on the number of sets of private caches. When the number of cores is less or equal than the number of private cache sets, the proposed organization requires a fixed size to store directory information, which is significantly smaller than the one required by a full-map directory. Therefore, this proposal reduces the area requirements of a directory-based protocol.

- An *implicit replacements* mechanism that allows the cache coherence protocol to remove all messages caused by replacements. This mechanism requires a directory organization similar to the one previously mentioned. The interleaving used by this directory organization forces that each cache entry is associated to a particular directory entry. In this way, the requesting tile does not have to inform the directory about replacements because the directory already knows which block is being replaced when the request for a new block arrives to it. Therefore, this proposal reduces the total network traffic generated by the coherence protocol, which at the end will result in savings in terms of power consumption.

- A new family of cache coherence protocols called *direct coherence* protocols aimed at avoiding the indirection of traditional directory-based protocols, but without relying on broadcasting requests. The key property of this family of cache coherence protocols is that both the role of ordering requests from different processors to the same memory block and the role of storing the coherence information is moved from the home tile to the tile that provides the data block in a cache miss, i.e., the *owner* tile. Therefore, indirection is avoided by directly sending requests to the owner tile instead of to the home one. Indirection avoidance reduces the number of hops required to solve cache misses and, as a consequence, the average cache miss latency. Moreover, we also study the traffic area trade-off of direct coherence protocols by using compressed sharing codes instead of full-map ones.

- A *distance-aware round-robin* mapping policy that tries to map memory pages to the cache bank belonging to the core that most frequently accesses the blocks within that page. In this way, the average access latency

to a NUCA cache is reduced. In addition, the proposed policy also introduces an upper bound on the deviation of the distribution of memory pages among cache banks, which lessens the number of off-chip accesses. The mapping policy is managed by the OS and, therefore, it is performed at the granularity of the page size. In this way, we reduce the average cache miss latency and, consequently, applications' execution time. Since the distance between requesting cores and home tiles is reduced, an important fraction of network traffic is also saved.

- We have evaluated all the proposals presented in this thesis in a common framework using full-system simulation. We have found that our proposals improve the performance of applications, reduce the network traffic or save some extra storage required to keep coherence information.

All the contributions of this thesis have been published or are currently being considered for publication in international peer reviewed conferences [95, 96, 97, 98, 99, 100], peer reviewed journals [102] or book chapters [101].

1.4 Thesis overview

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

- Chapter 2 presents a background on cache hierarchy organizations and cache coherence protocols for tiled CMPs. This chapter discusses several choices when designing cache coherence protocols which we think that are essential to ensure full understanding of the contents of this thesis.

- Chapter 3 describes the evaluation methodology employed throughout this thesis. It discusses the tools, workloads and metrics used for the evaluation of the proposed ideas.

- Chapter 4 faces the directory memory overhead problem. A scalable distributed directory organization for tiled CMPs is proposed. This chapter also proposes the implicit replacements mechanism that allows the protocol to remove the coherence messages caused by evictions.

- Chapter 5 proposes direct coherence protocols, a family of cache coherence protocols aimed at avoiding the indirection problem of directory-based

protocols, but without relying on broadcast. We discuss the area requirements of direct coherence protocols, and some mechanisms that help to avoid the indirection for a larger fraction of misses.

- Chapter 6 copes with the traffic-area trade-off in cache coherence protocols by discussing and evaluating several implementations of direct coherence protocols. These implementations differ in the amount of coherence information that they store. By using compressed sharing codes, the area required by direct coherence protocols can be considerably reduced at the cost of slightly increasing network traffic.

- Chapter 7 addresses the long cache access latencies to NUCA caches. Particularly, we propose an OS-managed mapping policy called distance-aware round-robin mapping policy. Moreover, we show that the private cache indexing commonly used in many-core architectures is not the most appropriate for OS-managed distance-aware mapping policies, and we propose to employ different bits for such indexing.

- Finally, Chapter 8 summarizes the main conclusions of the thesis and suggests future directions to be explored.

Chapters 4, 5 and 7 also include descriptions of related works that can be found in the literature on each one of the topics covered in this thesis.

Chapter 2

Background

This chapter presents an overview of current cache coherence protocols and discuss several alternatives to design the cache hierarchy in tiled CMP architectures. We explain the decisions taken for the base configuration used for the evaluation of the ideas proposed in this thesis. Some of these aspects are further discussed according to each one of the proposals and, therefore, it is fundamental to introduce them previously.

Particularly, the chapter discusses the design choices that can be found in the literature both for the organization of the last-level on-chip caches and for the design of cache coherence protocols for tiled CMP architectures. Regarding the cache hierarchy, we discuss the two main ways of organizing the last-level on-chip cache, i.e., the *private* and the *shared* organizations. We also discuss the inclusion properties between private and shared caches. With respect to cache coherence protocols, we discuss several alternative protocols for the maintenance of cache coherence and some variations for the design of these cache coherence protocols.

First of all, we define the three main actors that take part in cache coherence for a better understanding of the remaining document: *requester*, *home* and *owner*.

- *Requester* or requesting core. This is the core that accesses a block that misses in its private cache and, as consequence, it generates a memory request for that block. We also use the term requesting tile to refer to the tile wherein that core is placed.

- *Home* tile. Each memory block has a home tile. The home tile serializes the requests for its memory blocks, and it also keeps cache coherence for those blocks. Therefore, it stores the information necessary to keep cache coherence.

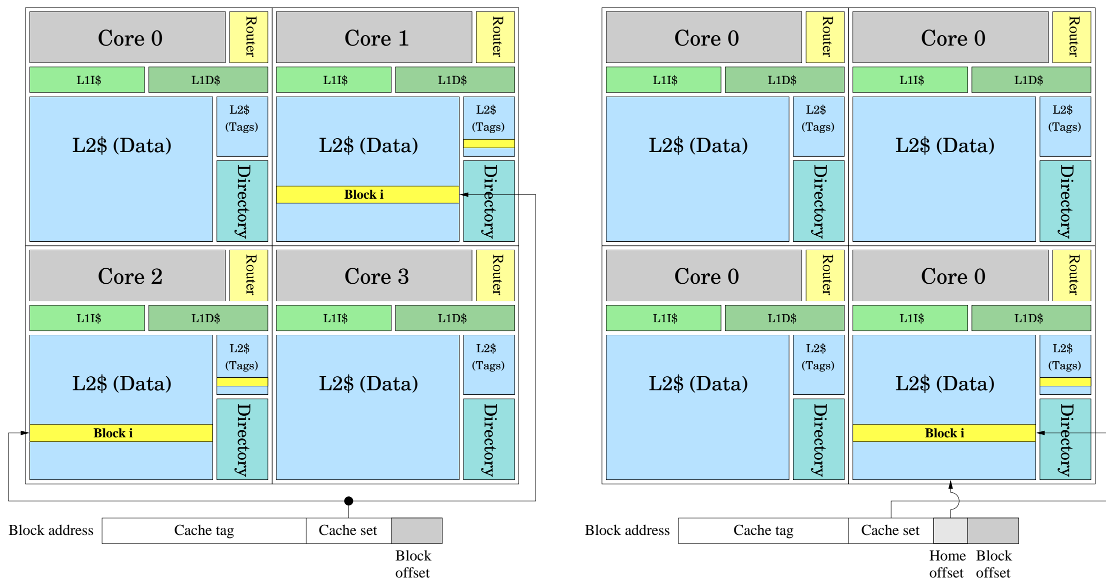
- *Owner* tile or provider tile. It is the tile that holds the *fresh* copy of the block, i.e., the tile that must provide a copy of the requested block in a cache miss. We also name the cache that holds that copy as the owner cache.

2.1 Private versus shared organization

As discussed in the introduction of this thesis, there are two major alternatives for organizing the last-level on-chip cache in a tiled CMP [68]: the *private* organization and the *shared* organization. In this section we detail and analyze both organizations for the L2 cache, which is the last-level on-chip cache for the CMP organization considered in this thesis (see Figure 1.1).

The private organization is equivalent to simply shrinking a traditional multi-chip multiprocessor (e.g., a cc-NUMA machine [62]) onto a single chip. In this organization, the total L2 cache storage is partitioned into several banks distributed among the processing cores. Each cache bank is private to a particular core. Therefore, each core can access its local L2 cache bank without going across the on-chip interconnect and it attracts to its cache banks these memory blocks that it requests. Blocks are stored in that cache as in a traditional one, i.e., according to a set field (see Figure 2.1(a)). Cache coherence is maintained at the L2 cache level and, consequently, upon an L2 cache miss, other private L2 banks must be also consulted (depending on coherence information) to find out if they have valid data or, otherwise, access off-chip. Some CMPs, as Intel Montecito [81], AMD Opteron [56], or IBM POWER6 [63] organize the last-level on-chip cache in this way.

On the other hand, in a shared organization, all the L2 cache banks are logically shared among the processing cores, thus having the same address space. Differently from the private configuration, in a distributed shared cache, blocks can only be stored in a particular cache bank, i.e., the home bank. The home bank of each block is commonly obtained from the address bits. Particularly, the bits usually chosen for the mapping to a particular bank are the less significant ones without considering the block offset [52, 104, 123]. Therefore, blocks are stored in a distributed shared cache according to a set field and a home field



(a) Private L2 organization. Block i is stored in the local L2 cache bank of each requesting core.

(b) Shared L2 organization. Block i is only stored in the home L2 cache according to the home offset.

Figure 2.1: Private and shared L2 cache organizations. Core 1 and Core 2 are accessing the same memory block (Block i).

within the address of the block (see Figure 2.1(b)). Now, cache coherence is maintained at the L1 cache level. This organization is implemented by several CMPs, such as Piranha [16], Hydra [46], Sun UltraSPARC T2 [104] and Intel Merom [103]. The latest developments of Intel and AMD also use the shared organization for their last-level L3 caches (e.g., Intel Nehalem [53]).

Both organizations have perks and drawbacks. Specifically, the main advantages usually claimed for the private cache organization are the following:

- Short L2 cache access latency: L2 cache hits usually take place in the local cache bank of the requesting processor, thus reducing the latency in the common case.

- Small amount of network traffic generated: Since the local L2 cache bank can filter most of the memory requests, the number of coherence messages injected into the interconnection network is limited.

However, this organization has some disadvantages with respect to the shared organization. First, data blocks can get duplicated across the different

L2 banks, which can lessen the on-chip storage utilization. Second, if the working set accessed by the different cores is not well-balanced, some caches can be over-utilized whilst others can be under-utilized. This second factor can also raise the number of off-chip accesses.

On the other hand, the shared organization makes better use of cache resources. In this way, it achieves higher L2 cache hit rates and, therefore, it reduces the number of costly off-chip accesses. This better cache utilization is due to the following reasons:

- Single copy of blocks: A shared organization does not duplicate blocks in the L2 cache because each block maps to a single cache bank.
- Workload balancing: Since the utilization of each cache bank does not depend on the working set accessed by each core, but they are uniformly distributed among cache banks in a round-robin fashion, the aggregate cache capacity is augmented.

Unfortunately, the disadvantage of this organization is that although some accesses to the L2 cache will be sent to the local cache bank, the rest will be serviced by remote banks (L2 NUCA architecture), and this increases L2 cache access latency. In addition, the access to a remote cache bank injects extra traffic into the on-chip network.

We choose an L2 shared organization for the evaluation carried out in this thesis because it achieves higher on-chip hit rate compared to the private configuration. Additionally, to make better use of the capacity of the on-chip caches, we assume non-inclusive L1 and L2 caches. On the other hand, in this thesis we address the long access latencies that NUCA caches introduce in some cases, which makes this environment more appropriate for our work. However, our ideas can also be employed with private L2 caches as we discuss in each chapter. Particularly, a study about direct coherence protocols, presented in Chapter 5, for cc-NUMA architectures with private L2 caches can be found in [97].

2.2 Cache coherence protocols

As discussed in the introduction of this thesis, allowing multiple processors that access the same address space to store data in their private caches results in the cache coherence problem. This problem is made transparent to software through a cache coherence protocol implemented in hardware. There are two different

policies that can be used to keep cache coherence, and based on them, we can distinguish between invalidation- and update-based cache coherence protocols [108, 109]. Upon a write request, invalidation-based protocols [43] require that all sharers (except the requester) be invalidated. On the other hand, update-based protocols [82] propagate the result of the write operation to all the sharers. The main disadvantage of update-based protocols is the amount of network traffic generated. Particularly, when a processing core writes a block multiple times before another core reads the block, all updates must be notified, each one in a different message. Although adaptive protocols can reduce this drawback [89], this is one of the main reasons why most recent systems implement invalidation-based protocols and, therefore, this thesis only considers this kind of cache coherence protocols.

Invalidation-based protocols must ensure the following invariant. At any point in logical time, a cache block can be written by a single core or it can be read by multiple cores. Therefore, if a processing core wants to modify a cache block, this block has to be previously invalidated (revoking read permission) from the other caches. Likewise, if a processing core wants to read a cache block, write permission must be previously revoked if some cache had it.

When implementing a cache coherence protocol there are other important design decisions that affect the final efficiency of the protocol. Next sections discuss the choices that are more related to the research carried out in this thesis.

2.3 Design space for cache coherence protocols

There are several alternatives for designing cache coherence protocols depending on the states of the blocks stored in the private caches. These alternatives have been commonly named according to the states that they employ: MOESI, MOSI, MESI, MSI, etc. Each state represents different permissions for a block stored in a private cache:

- **M (Modified)**: A cache block in the modified state holds the only valid copy of the data. The core holding this copy in its cache has read and write permissions over the block. The other private caches cannot hold a copy of this block. The copy in the shared L2 cache (if present) is stale. When another core requests the block, the cache with the block in the modified state must provide it.

- **O (Owned)**: A cache block in the owned state holds a valid copy of the data but, in this case, another copies in shared state (not in the owned state) can coexist. The core holding this copy in its cache have read permission but cannot modify it. When this core tries to modify it, coherence actions are required to invalidate the remaining copies. In this way, the owned state is similar to the shared state. The difference lies in the fact that the owned state is responsible for providing the copy of the block in a cache miss, since the copy in the shared L2 cache (if present) is stale. Moreover, evictions of blocks in owned state always entail writeback operations.
- **E (Exclusive)**: A cache block in the exclusive state holds a valid copy of the data. The other private caches cannot hold a copy of this block. The core holding this copy in its cache can read and write it. The shared L2 cache could also store a valid copy of the block.
- **S (Shared)**: A cache block in the shared state holds the valid copy of the data. Other cores may hold copies of the data in the shared state and one of them in the owned state. If no private cache holds the block in the owned state, the shared L2 cache has also a valid copy of the block and it is responsible for providing it.
- **I (Invalid)**: A cache block in the invalid state does not hold a valid copy of the data. Valid copies of the data might be either in the shared L2 cache or in another private cache.

Table 2.1: Properties of blocks according to their cache state.

	Property		
State	**Exclusiveness**	**Ownership**	**Validity**
M	√	√	√
O	χ	√	√
E	√	χ	√
S	χ	χ	√
I	χ	χ	χ

Table 2.1 outlines the properties of memory blocks according to their state in a private cache. Only the states *M* and *E* entail the exclusiveness of the block

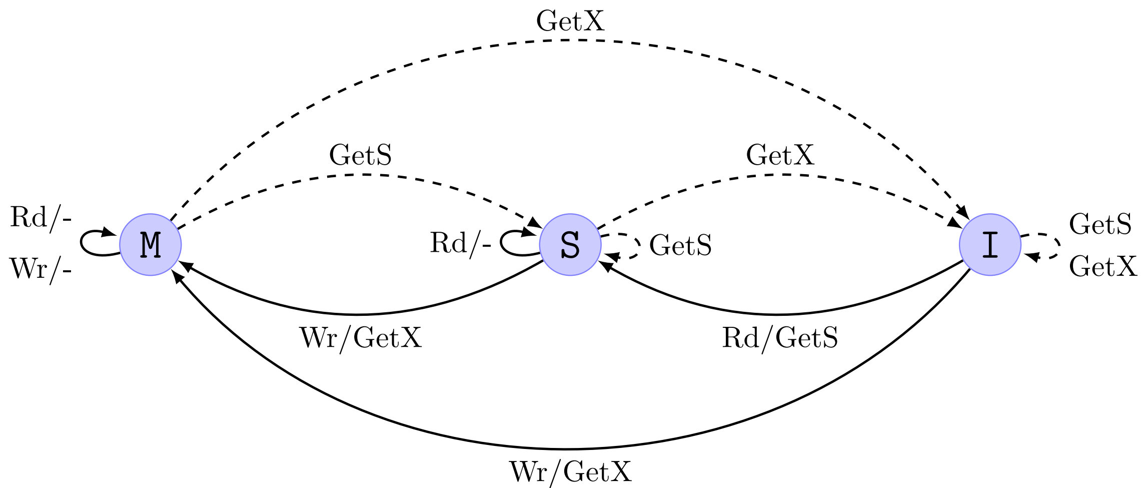


Figure 2.2: State transition diagram for a MSI protocol.

and, therefore, write permission. On the other hand, the states *M* and *O* imply ownership of the block, i.e., a cache with a block in one of these states is the owner cache for this block, and cache misses have to obtain the copy of the requested block from this cache. Finally, all states except the *I* one indicate that the copy of the block is valid, so the local core has at least read permission over it.

The MSI states represent the minimum set that allows the cache coherence protocol to ensure the invariant previously mentioned when write-back private caches are used. Either a single processing core has read/write permissions for the block, i.e., it caches the block in the state *M*, or multiple processing cores have read permission for the block, i.e., they cache the block in the state *S*. Figure 2.2 shows the state transition diagram for a MSI cache coherence protocol. Often, when a new block is stored in the cache, another block must be evicted from the cache. Since evictions of blocks always result in the cache block transitioning to the *I* state we choose not to show these transitions in the diagram. When a processing core needs read permission for a particular cache block (*Rd*) it issues a *GetS* request if the it has not read permission for that block (*Rd/GetS*). Otherwise, if the processing core has read permission for that block any request is generated (*Rd/-*). On the other hand, when the processing core requires write permission (*Wr*) it can send a *GetX* request. In the diagrams shown in Figures 2.2, 2.3 and 2.4, the normal arrows correspond to transitions caused by local re-

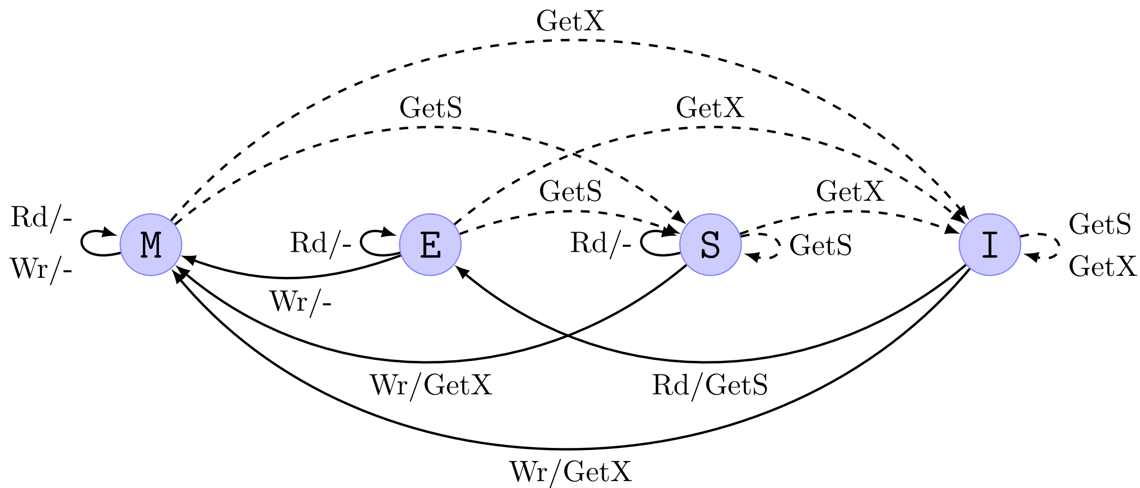


Figure 2.3: State transition diagram for a MESI protocol.

quests while the dashed arrows represent transitions due to requests generated by remote processing cores. Although the MSI protocol has a relatively simple design, it also introduces some inefficiencies that can be remedied with the addition of the states *E* and *O*.

The exclusive or *E* state optimizes the MSI protocol for non-shared data. Hence, it is essential to obtain good performance for sequential applications running on a multiprocessor. Upon a read request, the requested block is stored in cache in the exclusive state, instead of in the shared state. In this way, the requesting processor obtains write permission for the block, and a subsequent write request for this block will not fall into a cache miss (if no other processing core requested the block). In this case, the block silently transition from the exclusive state to the modified one. The main difference of the exclusive state with respect to the modified one is that the block is clean and the shared cache could store a valid copy of the block. In this way, when the block is in the exclusive state it is not necessary to writeback the data block to the shared cache neither in case of evictions nor in case of transitions to the shared state due to read requests by other processing cores. Figure 2.3 shows the state transition diagram for the MESI cache coherence protocol.

The owned state was introduced as an optimization of the MESI protocol [112]. When a remote *GetS* is received by a core that caches the block in the modified state, this state transitions to the owned one, instead of to the shared

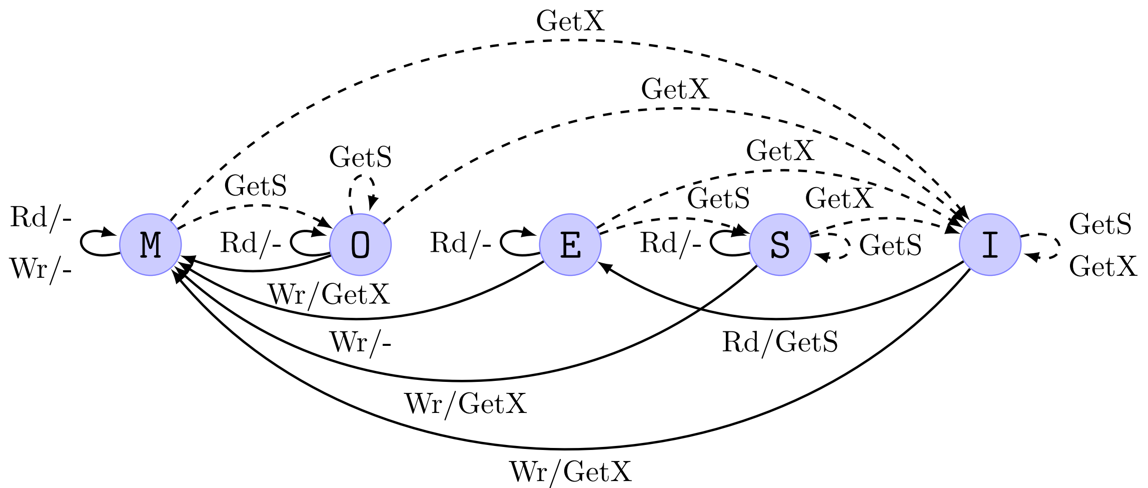


Figure 2.4: State transition diagram for a MOESI protocol.

one. The owned state is similar to the shared state with the difference that it has a dirty copy of the block and, therefore, the shared cache does not need to keep a coherent copy of the block. Thus, the addition of the owned state has the following advantages.

- First, network traffic can be reduced by not requiring a processing core to writeback the data block to the shared L2 cache when the block transitions from the modified to the owned state on a read request. If another processing core issues a write request for that block before it is evicted, the writeback message is saved.

- Second, the shared L2 cache does not need to keep a copy for blocks in the owned state, which can result in better utilization, thus reducing the miss rate of the shared cache. Note that in CMPs with a shared cache organization, the misses of the shared cache require off-chip accesses.

- Third, for some architectures cache misses can be solved more quickly by providing data from private caches than from the shared cache. This is mainly the case of the cc-NUMA machines [96, 102], where the shared cache is represented by main memory, or CMPs with a private cache organization [27]. On the other hand, in CMPs with a shared cache organization, the data block can be provided faster from the shared cache and, in this case, this advantage can be dismissed.

Figure 2.4 shows the state transition diagram for the MOESI cache coherence protocol. The owned state also can be used without the exclusive state leading to a MOSI protocol whose state transition diagram is not shown. All the cache coherence protocols considered in this thesis assume MOESI states.

2.3.1 Optimization for migratory sharing

The MOESI protocol previously described is efficient for some sharing patterns, but it is not optimized for the migratory-sharing pattern. According to Gupta and Weber [117], the migratory-sharing pattern is followed by data structures manipulated by only a single processing core at a time. Typically, such sharing occurs when a data structure is modified within a critical section, e.g., protected by locks.

Cache blocks that follow a migratory-sharing pattern are commonly read by a processing core and subsequently written by the same core. Since the read request for a modified or exclusive block does not give write permission in the described MOESI protocol, two cache misses are involved in the migration of the block. The first one obtains the data block and the second one invalidates the other copy. Since both misses are sent to the cache associated with the core that previously modified the block, the invalidation can be merged with the first miss request. This optimization is called the migratory-sharing optimization [36, 110], and it is implemented by all protocols evaluated in this thesis.

As commented, in the migratory-sharing optimization, a cache holding a modified memory block invalidates its copy when responding for a migratory read request, thus granting the requesting core read/write access to the block. This optimization has been shown to improve substantially the performance of many applications [36, 110].

2.4 Protocols for unordered networks

As introduced at the beginning of this document, traditional snooping-based protocols require an ordered interconnect to keep cache coherence, but such interconnects do not scale in terms of both area requirements and power consumption. In this section, we describe three cache coherence protocols aimed at being used over unordered networks that either can be found in the literature or have been implemented in commercial systems: *Hammer*, *Token* and *Directory*. We explain their particular implementation for tiled CMP architectures.

These protocols have been used as base protocols for the evaluation of the ideas presented in this thesis.

In the protocols implemented and evaluated in this thesis, evictions of clean blocks from private caches are performed without generating any coherence action (except for a protocol evaluated in Chapter 4 where informing about these replacements is necessary). This kind of evictions are commonly called *silent* evictions. However, upon evictions of dirty blocks the cache coherence protocol needs to perform a writeback of the evicted block to the shared cache. Both *Hammer* and *Directory* perform these evictions in three phases, i.e., it is necessary an acknowledgement from the home tile before the writeback message with a copy of the evicted data block can be sent. These three-phase evictions allow these protocols to avoid complex race conditions. However, *Token* evicts dirty blocks just by sending one message along with the tokens and the data block to the home tile of that block. Likewise, direct coherence protocols, presented in Chapter 5, can perform replacements by sending a single message without suffering complex race conditions.

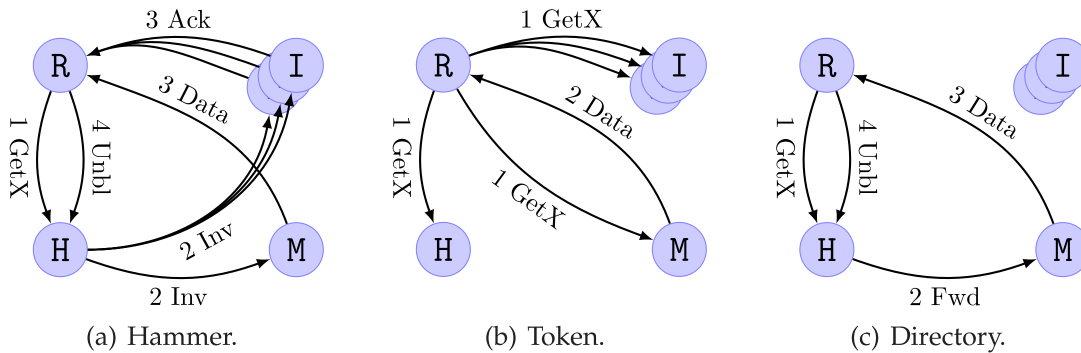
2.4.1 Hammer protocol

Hammer [92] is the cache coherence protocol used by AMD in their Opteron systems [11]. Like snooping-based protocols, *Hammer* does not store any coherence information about the blocks held in the private caches and, therefore, it relies on broadcasting requests to all tiles to solve cache misses. Its key advantage with respect to snooping-based protocols is that it targets systems that use unordered point-to-point interconnection networks. In contrast, the ordering point in this protocol is the home tile, a fact that introduces indirection on every cache miss.

We have implemented a version of the AMD's Hammer protocol for tiled CMPs. As an optimization, our implementation adds a small structure to each home tile. This structure stores a copy of the tag for the blocks that are held in the private L1 caches. In this way, cache miss latencies are reduced by avoiding off-chip accesses when the block can be obtained on-chip. Moreover, the additional structure has small size and it does not increase with the number of cores.

On every cache miss, *Hammer* sends a request to the home tile. If the memory block is present on-chip[1], the request is forwarded to the rest of tiles to obtain the requested block, and to eliminate potential copies of the block in case of a write miss. Otherwise, the block is requested to the memory controller.

[1]This information is given by the structure that we add to each home tile.



(a) Hammer. (b) Token. (c) Directory.

Figure 2.5: A cache-to-cache transfer miss in each one of the described protocols.

All tiles answer to the forwarded request by sending either an acknowledgement or the data message to the requesting core. The requesting core needs to wait until it receives the response from each other tile. When the requester receives all the responses, it sends an unblock message to the home tile. This message notifies the home tile about the fact that the miss has been satisfied. In this way, if there is another request for the same block waiting at the home tile, it can be processed. Although this unblock message can introduce more contention at the home tile, it prevents the occurrence of race conditions. This message is also used to prevent race conditions in directory-based protocols, which are described later.

Figure 2.5(a) shows an example of how *Hammer* solves a cache-to-cache transfer miss. The requesting core (*R*) sends a write request (*1 GetX*) to the home tile (*H*). Then, invalidation messages (*2 Inv*) are sent to all other tiles. The tile with the ownership of the block (*M*) responds with the data block (*3 Data*). The other tiles that do not hold a copy of the block (*I*) respond with acknowledgement messages (*3 Ack*). When the requester receives all the responses, it sends the unblock message (*4 Unbl*) to the home tile. First, we can see that this protocol requires three hops in the critical path before the requested data block is obtained. Second, broadcasting invalidation messages increases considerably the traffic injected into the interconnection network and, therefore, its power consumption.

2.4.2 Token protocol

Token coherence [75] is a framework for designing coherence protocols whose main asset is that it decouples the correctness substrate from several different

performance policies. Token coherence protocols can avoid both the need of a totally ordered network and the introduction of additional indirection caused by the access to the home tile in the common case of cache-to-cache transfers. Token coherence protocols keep cache coherence by assigning T tokens to every memory block, where one of them is the owner token. Then, a processing core can read a block only if it holds at least one token for that block and has valid data. On the other hand, a processing core can write a block only if it holds all T tokens for that block and has valid data. Token coherence avoids starvation by issuing a persistent request when a core detects potential starvation.

In this thesis, we use *Token-CMP* [78] in our simulations. *Token-CMP* is a performance policy aimed at achieving low-latency cache-to-cache transfer misses. It targets CMP systems, and uses a distributed arbitration scheme for persistent requests, which are issued after a single retry to optimize the access to contended blocks.

Particularly, on every cache miss, the requesting core broadcasts requests to all other tiles. In case of a write miss, they have to answer with all tokens that they have. The data block is sent along with the owner token. When the requester receives all tokens the block can be accessed. On the other hand, just one token is required upon a read miss. The request is broadcast to all other tiles, and only those that have more than one token (commonly the one that has the owner token) answer with a token and a copy of the requested block.

Figure 2.5(b) shows an example of how *Token* solves a cache-to-cache transfer miss. Requests are broadcast to all tiles (*1 GetX*). The only tile with tokens for that block is *M*, which responds by sending the data and all the tokens (*2 Data*). We can see that this protocol avoids indirection since only two hops are introduced in the critical path of cache misses. However, as happens in *Hammer*, this protocol also has the drawback of broadcasting requests to all tiles on every cache miss, which results in high network traffic and, consequently, power consumption at the interconnect.

2.4.3 Directory protocol

Directory-based coherence protocols [24] have been widely used in shared-memory multiprocessors. Examples of traditional multiprocessors that use directory protocols are the Standford DASH [64] and FLASH [60] multiprocessors, the SGI Origin 2000/3000 [62], and the AlphaServer GS320 [42]. Now, several Chip Multiprocessors, as Piranha [16] or Sun UltraSPARC T2 [104], also use directory protocols to keep cache coherence.

As happens in *Hammer*, the serialization point of directory-based protocols is also the home tile of each block. In contrast, they avoid broadcasting requests by keeping information about the state of each block in the private caches. This information is called *directory* information (hence the name of directory-based protocols). In order to accelerate cache misses, this directory information is not stored in main memory. Instead, it is usually stored on-chip at the home tile of each block.

The directory-based protocol that we have implemented for CMPs is similar to the intra-chip coherence protocol used in Piranha. In particular, the directory information consists of a full-map (or bit-vector) sharing code, that is employed for keeping track of the sharers. This sharing code allows the protocol to send invalidation messages just to the caches currently sharing the block, thus removing unnecessary coherence messages. In directory-based protocols that implement the *O* state (see Section 2.3), an *owner* field that identifies the owner tile is also added to the directory information of each block. The owner field allows the protocols to detect the tile that must provide the block in case of several sharers. In this way, requests are only forwarded to that tile. The use of directory information allows the protocol to reduce considerably network traffic when compared to *Hammer* and *Token*.

On every cache miss in the implemented directory protocol, the core that causes the miss sends the request only to the home tile, which is the serialization point for all requests issued for the same block. Each home tile includes an on-chip directory cache that stores the sharing and owner information for the blocks that it manages. This cache is used for the blocks that do not hold a copy in the shared cache. In addition, the tags' part of the shared cache also include a field for storing the sharing information for those blocks that have a valid entry in that cache. Once the home tile decides to process the request, it accesses to the directory information and it performs the appropriate coherence actions. These coherence actions include forwarding the request to the owner tile, and invalidating all copies of the block in case of write misses.

When a tile receives a forwarding request it provides the data to the requester if it is already available or, in other case, the request must wait until the data will be available. Like in *Hammer*, all tiles must respond to the invalidation messages with an acknowledgement message to the requester. Since acknowledgement messages are collected by the requester, it is necessary to inform the requester about the number of acknowledgements that it has to receive before accessing the requested data block. In the implementation that we use in this thesis, this information is sent from the home tile, which knows the number of

invalidation messages issued, to the requester along with the forwarding and data messages. When the requester receives all acknowledgements and the data block, it unblocks the home tile in order to allow it to process more requests for that block.

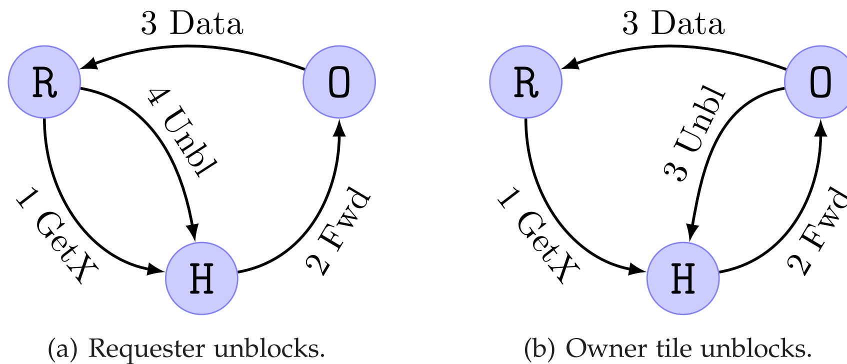
Figure 2.5(c) shows an example of how *Directory* solves a cache-to-cache transfer miss. The request is sent to the home tile, where the directory information is stored (*1 GetX*). Then, the home tile forwards the request to the provider of the block, which is obtained from the directory information (*2 Fwd*). When the data sent by the provider arrives to the requester (*3 Data*), the miss is considered solved and the home tile must be unblocked (*4 Unbl*). As we can see, although this protocol introduces indirection to solve cache misses (three hops in the critical path of the miss), few coherence messages are required to solve them, which finally translates into savings in network traffic and less power consumption. This characteristic makes the directory protocol the most scalable alternative.

2.4.4 Summary

Table 2.2: Summary of cache coherence protocols.

	Network	Requests	Indirection
Snooping	Shared interconnect	To all tiles	No
Hammer	Point-to-point	To all tiles	Yes
Token	Point-to-point	To all tiles	No
Directory	Point-to-point	Only to necessary tiles	Yes

Table 2.2 summarizes the cache coherence protocols previously described. Traditional snooping-based protocols are not suitable for scalable point-to-point networks. *Hammer* can work over scalable point-to-point networks at the cost of broadcasting requests to all tiles and introducing indirection in the critical path of cache misses. *Token* avoids the indirection but still sends requests to all tiles on every cache miss, which impacts on network traffic and power consumption. In contrast, *Directory* just send requests to the tiles that must receive them, but it introduces indirection, which impacts on applications' execution time. In Chapter 5, we present a new family of coherence protocols that avoids both broadcasting requests to all tiles and the indirection to the home tile for most cache misses.



(a) Requester unblocks.

(b) Owner tile unblocks.

Figure 2.6: Management of unblock messages for cache-to-cache transfer misses.

2.5 Sending unblock messages

In the directory-based protocol previously described, cache misses finish with the requesting tile sending an unblock message to the home tile once data and acknowledgements have been obtained. This message allows the home tile to process the subsequent requests for the same block. However, there is another alternative in which the unblock message is sent from the tile that provides the data block on a cache miss, i.e., the owner tile [93]. Next, we discuss the pros and cons of each approach:

- *Requester unblocks*: In this case, the home tile is unblocked when the cache miss is completely solved, as shown in Figure 2.6(a). The main advantage of this approach is the avoidance of race conditions due to overlapping requests. The drawback is that the home tile cannot solve other requests for the same block for longer time, which increases the waiting time of subsequent requests at the home tile. This increase in the waiting time finally translates into longer cache miss latencies.

- *Owner tile unblocks*: In this case, the tile that provides the block unblocks the home tile, as shown in Figure 2.6(b). The advantage of this approach is a slight reduction in terms of waiting time at the home tile, which can accelerate subsequent cache misses. Note that the waiting time at the home tile will be reduced from three hops to two hops in the critical path, which does not represent a significant improvement.

  The issue of this implementation is that the home tile can be unblocked before the data block arrives to the requester. Then, if there is a request

waiting at the home tile, it will be forwarded to the new owner of the block, i.e, the one waiting to receive the data block. Likewise, invalidation messages can also be sent to the new owner of the block in case of write misses. If either the request or the invalidation messages arrive before than the data block of the previous cache miss a race condition occurs, and it could be difficult to find out if the data is coming or it has been already evicted and requested again.

These race conditions are solved in the following way. Forwarded messages are always sent to the tile that has (or will have) the ownership of the requesting block. Since the evictions of owner blocks are performed in three phases requiring the confirmation of the home tile, the eviction is not acknowledged by the home tile when there is an ongoing request for that block. Therefore, if the tile that receives the forwarded message is not the owner one, the data message has not arrived yet and, then, the forwarded request can wait until it arrives.

On the other hand, invalidation messages are sent to the sharers without the ownership property, and these blocks are evicted without any confirmation from the home tile (silent evictions). If a new request for the evicted block is immediately issued, the protocol is no longer able to know whether the data message is coming or the data block has been previously evicted. If the protocol incorrectly assumes that the block has been evicted and it acknowledges the invalidation, it will lead to an incoherence. Therefore, the assumption taken is that the data message is coming. Then, when a invalidation arrives and the block is not present in the cache, the acknowledge is immediately performed, but the incoming block (if any) will not be kept in cache. This approach can lead to unnecessary (or premature) invalidations. If the core requesting the block wants to access it several times, this approach will increase the cache miss rate.

Due to the issues of the second approach and its slight advantage, the directory protocols evaluated in this thesis implement the first approach. However, for direct coherence protocols, which are described in Chapter 5, the advantages of the second approach become more acute. First, the waiting time is reduced from two hops in the critical path to none. And second, the unblock message can be removed, thus saving network traffic and power consumption. Therefore, direct coherence protocols implement the second approach.

Chapter 3

Evaluation Methodology

This chapter presents the experimental methodology used for all the evaluations carried out in this thesis. Experiments have entailed running several representative workloads on a simulation platform.

We have selected for the evaluation process the full-system simulator Virtutech Simics [72] extended with Multifacet GEMS 1.3 [77] from the University of Wisconsin. GEMS provides a detailed memory system timing model which accounts for all protocol messages and state transitions. However, the interconnection network modeled in GEMS 1.3 is not very detailed. Since, in this thesis, we compare broadcast-based protocols with directory-based ones it is very important to model precisely the interconnection network for obtaining more accurate results. Therefore, we have replaced the original network simulator offered by GEMS 1.3 with SiCoSys [94], a detailed interconnection network simulator. SiCoSys allows to take into account most of the VLSI implementation details with high precision but with much lower computational effort than hardware-level simulators. Finally, we have also used the CACTI 5.3 tool [115] to measure both the area and latencies of the different caches employed for the evaluated protocols.

We have used several parallel benchmarks from different suites to feed our simulator. These benchmarks cover a variety of computation and communication patterns. We have employed several scientific applications which are mainly from the SPLASH-2 benchmark suite [118]. We also evaluate multimedia applications from the ALPBench suite [66]. Finally, we also have analyzed the behav-

ior of some of our proposals using multi-programmed workloads, since tiled CMPs will also be employed for throughput computing.

The rest of the chapter is structured as follows. Section 3.1 details the simulation tools used for the performance evaluations carried out in this thesis. Section 3.2 describes the base system modeled by our simulator. Section 3.3 discusses the metrics and methods employed for measuring our proposals. Finally, descriptions of the benchmarks running on top of our simulation tool appear in Section 3.4.

3.1 Simulation tools

In this section, we describe the simulation tools employed through this thesis. We obtain the performance and network traffic of the workloads for each proposal in this thesis by using the Simics-GEMS simulator extended with SiCoSys. The area requirements of our proposals and the latencies of cache accesses used in our simulations have been calculated using the CACTI tool.

3.1.1 Simics-GEMS

Simics [72] is a functional full-system simulator capable of simulating several types of hardware including multiprocessor systems. Full-system simulation enables us to evaluate our ideas running realistic workloads on top of actual operating systems. In this way, we also simulate the behavior of the operating system. Differently from trace-driven simulators, Simics allows dynamic change of instructions to be executed depending on different input data.

GEMS (General Execution-driven Multiprocessor Simulator) [77] is a simulation environment which extends Virtutech Simics. GEMS is comprised of a set of modules implemented in C++ that plug into Simics and add timing abilities to the simulator. GEMS provides several modules for modeling different aspects of the architecture. For example, Ruby models memory hierarchies, Opal models the timing of an out-of-order SPARC processor, and Tourmaline is a functional transactional memory simulator. Since we assume simple in-order processing cores we only use Ruby for the evaluations carried out in this thesis.

Ruby provides an event-driven framework to simulate a memory hierarchy that is able to measure the effects of changes to the coherence protocols. Particularly, Ruby includes a domain-specific language to specify cache coherence protocols called SLICC (Specification Language for Implementing Cache Coher-

ence). SLICC allows us to easily develop different cache coherence protocols and it has been used to implement the protocols evaluated in this thesis.

The memory model provided by Ruby is made of a number of components that model the L1 caches, L2 caches, memory controllers and directory controllers. These components model the timing by calculating the delay since a request is received until a response is generated and injected into the network. All the components are connected using a simple network model that calculates the delay required to deliver a message from one component to another. Since the interconnection network model provided by GEMS 1.3 is very idealized we have replaced it with SiCoSys, a more detailed network simulator which is described next.

3.1.2 SiCoSys

SiCoSys (Simulator of Communication Systems) [94] is a general-purpose interconnection network simulator for multiprocessor systems that allows to model a wide variety of routers and network topologies in a precise way. SiCoSys is a time-driven simulator developed in C++ having in mind modularity, versatility and connectivity with other systems. The models used are intended to resemble the hardware implementations in some aspects while keeping the complexity as low as possible. In this way, the simulator mimics the hardware structure of the routers instead of just implementing their functionality. Therefore, results are very close to those obtained by using hardware simulators but at lower computational cost.

SiCoSys has a collection of hardware-inspired components like multiplexers, buffers or crossbars. Routers can be built by connecting components to each other, as they are in the hardware description. Then, the routers are connected in a certain network fashion. All this is defined in SGML (Standard Generalized Markup Language) which can be thought of as a superset of HTML. This allows SiCoSys to handle hierarchical descriptions of routers easily while keeping high readability of the configuration files.

SiCoSys also allows us to model different switching and routing techniques. Since SiCoSys does not implement multicast routing, we have extended the simulator to provide multicast support. Multicast routing allows the interconnection network to send a message to a group of destinations simultaneously in an efficient way. Messages are sent over each link of the network just once, creating copies only when the links to the multiple destinations split. This technique reduces the network traffic generated mainly by broadcast-based protocols but

also by directory-based protocols when invalidation messages are sent to several sharers.

3.1.3 CACTI

CACTI (Cache Access and Cycle Time Information) [115] provides an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. By integrating all these models together, users can have confidence that trade-offs between time, power, and area are all based on the same assumptions and, hence, are mutually consistent.

CACTI is continually being upgraded due to the incessant improvements in semiconductor technologies. Particularly, we employ the version 5.3 for the results presented in this thesis. We are mainly interested in getting the access latencies and area requirements of both cache and directory structures that are necessary for implementing our ideas. In this study, we assume that the length of the physical address is 40 bits as, for example, in the Sun UltraSPARC T2 architecture [111]. This length is used to calculate the bits required to store the tag field for each cache. Moreover, we also assume a 45nm process technology, and the other parameters shown in the following section.

3.2 Simulated system

The simulated system is a tiled CMP organized as a 4×4 array of replicated tiles, as shown in Figure 1.1. Exceptionally, when we evaluate multi-programmed workloads we consider a 4×8 tiled CMP. Our tiled CMP is comprised of two levels of cache on-chip. The first-level cache (L1) is split in both instruction and data caches, and it is private to the local core. The second-level cache (L2) is unified and logically shared among the different processing cores, but physically distributed among them[1]. Therefore, some accesses to the shared cache will be sent to the local slice whilst the rest will be serviced by remote slices (L2 NUCA architecture described in Section 2.1). Moreover, the L1 and L2 caches are non-inclusive to exploit the total cache capacity available on-chip. However, the proposals presented in this thesis could also be employed with either inclusive or exclusive cache policies. Finally, a distributed directory cache is also stored on-chip when a directory-based protocol is implemented. Each tile includes a bank

[1]Alternatively, each L2 slice could have been treated as a private L2 cache for the local core. In this case, cache coherence had to be maintained at the L2 cache level (instead of L1). In any case, our proposals would be equally applicable to this configuration.

of the directory cache to keep cache coherence for the blocks belonging to this tile. In order to avoid inefficiencies of directory-based protocols as consequence of replacements of directory entries, we assume an unlimited directory cache when simulating the base directory protocol.

Table 3.1: System parameters.

Memory Parameters (GEMS)	
Processor frequency	3GHz
Cache hierarchy	Non-inclusive
Cache block size	64 bytes
Split L1 I & D caches	128KB, 4-way
L1 cache hit time	1 (tag) + 2 (data) cycles
Shared unified L2 cache	1MB/tile, 8-way
L2 cache hit time	2 (tag) + 4 (data) cycles
Directory cache hit time	2 cycles
Memory access time	300 cycles
Page size	4KB
Network Parameters (SiCoSys)	
Network frequency	1.5GHz
Topology	2-dimensional mesh
Switching technique	Wormhole
Routing technique	Deterministic X-Y
Data and control message size	4 flits and 1 flit
Routing time	1 cycle
Switch time	1 cycle
Link latency (one hop)	2 cycles
Link bandwidth	1 flit/cycle

Since we consider tiled CMPs built from a relatively large number of cores (16 or 32), each tile contains an in-order processor core, thus offering better performance/Watt ratio than a small number of complex cores would obtain. We also assume that the interconnection network provides multicast support for all the experiments presented in this thesis. Table 3.1 shows the values of the main parameters of the architectures evaluated in this thesis. As previously mentioned, cache access latencies shown in this table have been calculated using the CACTI 5.3 tool [115]. The parameters and characteristics that are specific for

each proposal are later described in the corresponding chapter. Simulations have been performed using the sequential consistency model [61].

3.3 Metrics and methods

We evaluate the proposals presented in this thesis in terms of performance, network traffic and on-chip area required. For evaluating the performance, we measure the total number of cycles employed for each application during its parallel phase, i.e., the execution time of the parallel phase. Although the IPC (instructions per cycle) constitutes a common metric for evaluating performance improvements, it is not appropriate for multithreaded applications running on multiprocessor systems [13]. This is due to the spinning performed during the synchronization phase of the different threads. For example, a thread can be repeatedly checking the value of a lock until it becomes available, which increases the number of completed instructions (and maybe the IPC) but actually the program is not making progress.

In order to better understand the reasons why our proposals reduce the applications' execution time we also show the average latency of cache misses. Our proposals obtain improvements in terms of execution time mainly due to reductions in the average cache miss latency. When calculating this average we do not consider the overlapping of the cache misses, but it is calculated considering each miss individually. Savings in latency are partially obtained by the reduction in the number of *hops*. Depending on the context, we will talk about protocol hops (for example, in Chapter 5) or network hops (for example, in Chapter 7). Protocol hops are the number of coherence messages that are needed in the critical path of a cache miss for solving it. On the other hand, network hops represent the number of switches that a single coherence message must cross from the source tile to the destination one. Therefore, we also present results in terms of reductions in the number of hops.

We also measure the traffic injected in the interconnection network by the cache coherence protocol. The results are shown either in terms of number of messages or in terms of number of bytes transmitted through the interconnection network (the total number of bytes transmitted by all the switches of the interconnect). For example, the scheme proposed in Chapter 7 does not reduce the number of coherence messages, but it reduces the distance traveled by the message and, therefore, the number of bytes transmitted and the power consumption of the interconnection network. Savings in the number of coherence

messages usually entail reductions in the number of bytes transmitted. For all cache coherence protocols, we assume that control messages use 8 bytes while 72 bytes are employed for data messages.

Finally, we also evaluate the area required by our proposals. This area is calculated both in terms of the number of bits and mm^2. We employ the CACTI 3.5 tool to calculate the area required in terms of mm^2 by the structures added in our proposals. We employ these two metrics for the area overhead because we consider both of them important. The advantage of the first one is that it does not depend on the particular technology employed and, consequently, it is valid for any configuration. On the other hand, although the second metric depends on the technology assumed in this thesis, it offers more accurate results for our particular configuration. Moreover, we study the scalability in terms of area required by these structures by showing results with varying number of tiles. The other metrics specific to each chapter will be discussed where appropriate.

We account for the variability in multithreaded workloads by doing multiple simulation runs for each benchmark and each configuration, and injecting random perturbations in memory systems timing for each run [12]. We execute the same simulation several times using different random seeds and calculate the 95% confidence interval for our results, which is shown in our plots with error bars. Each data point is the result of at least 5 simulations, or even more in the case of applications that show more variability.

We have implemented the vast majority of the cache coherence protocols evaluated in this thesis using the SLICC language included in GEMS. Other protocols, like *Token*, are already provided by the simulator. All the implemented protocols have been exhaustively checked using a tester program provided by GEMS. The tester program stresses corner cases of cache coherence protocols to raise any incoherence by issuing requests that simulate very contended accesses to a few memory blocks.

All the experimental results reported in this thesis correspond to the parallel phase of each program. We have created benchmark checkpoints in which each application has been previously executed to ensure that memory is warmed up and, hence, avoiding the effects of page faults. Then, we run each application again up to the parallel phase, where each thread is bound to a particular core. The application is then run with full detail during the initialization of each thread before starting the actual measurements. In this way, we warm up caches to avoid cold misses.

3.4 Benchmarks

The applications the we use to evaluate the proposals presented in this thesis cover a wide variety of computation and communication patterns. We simulate both scientific and multimedia applications and we also feed our simulator with multi-programmed workloads. In particular, *Barnes*, *FFT*, *Ocean*, *Radix*, *Raytrace*, *Volrend* and *Water-Nsq* are scientific applications from the SPLASH-2 benchmark suite [118]. *Unstructured* is a computational fluid dynamics application [86]. *MPGdec* and *MPGenc* are multimedia applications from the ALPBench suite [66]. And finally, *Mix4*, *Mix8*, *Ocean4* and *Radix4* constitute the multi-programmed workloads employed in our simulations.

Table 3.2: Benchmarks and input sizes used in the simulations.

Category	Benchmark	Input Size
Scientific	Barnes	8192 bodies, 4 time steps
	FFT	64K complex doubles
	Ocean	130×130 ocean
	Radix	512K keys, 1024 radix
	Raytrace	Teapot
	Volrend	Head
	Unstructured	Mesh.2K, 5 time steps
	Water-Nsq	512 molecules, 4 time steps
Multimedia	MPGdec	525_tens_040.m2v
	MPGenc	output of *MPGdec*
Multi-programmed	Mix4	Ocean, Raytrace, Unstructured and Water-Nsq
	Mix8	Ocean×2, Raytrace×2, Unstruct.×2 and Water-Nsq×2
	Ocean4	Ocean×4
	Radix4	Radix×4

Table 3.2 shows the input sizes for each application employed in this thesis. Since full-system simulation incurs in slowdowns of several orders of magnitude, we are constrained to scaled down problem sizes for many scientific applications. For multimedia applications, the execution has been split into units

of work that we call transactions, and we measure the execution of a certain number of such transactions. Descriptions of each application are given in the following sections.

3.4.1 Scientific workloads

3.4.1.1 Barnes

The *Barnes* 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 modeled 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 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.

3.4.1.2 FFT

The *FFT* kernel is a complex one-dimensional 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.

3.4.1.3 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. Synchronization is performed by using both locks and barriers.

3.4.1.4 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.

3.4.1.5 Raytrace

This application renders a three-dimensional scene using ray tracing. A hierarchical uniform grid is used to represent the scene, and early ray termination is implemented. A ray is traced through each pixel in the image plane and it produces other rays as it strikes the objects of the scene, resulting in a tree of rays per pixel. The image is partitioned among processors in contiguous blocks of pixel groups, and distributed task queues are used with task stealing. The data accesses are highly unpredictable in this application. Synchronization in Raytrace is done by using locks. This benchmark is characterised for having

very short critical sections and very high contention. Barriers are not used for the *Raytrace* application.

3.4.1.6 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. Synchronization in this application is accomplished by using barriers and an array of locks.

3.4.1.7 Volrend

The *Volrend* application renders a three-dimensional volume using a ray casting technique. The volume is represented as a cube of voxels (volume elements), and an octree data structure is used to traverse the volume quickly. The program renders several frames from changing viewpoints, and early ray termination is implemented. A ray is shot through each pixel in every frame, but rays do not reflect. Instead, rays are sampled along their linear paths using interpolation to compute a color for the corresponding pixel. The partitioning and task queues are similar to those in Raytrace. Data accesses are input-dependent and irregular, and no attempt is made at intelligent data distribution. Synchronization in this application is mainly accomplished by using locks, but some barriers are also included.

3.4.1.8 Water-Nsq

The *Water-Nsq* application 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. Most synchronization is done using barriers, although there are also several variables holding global properties that are updated continuously and are protected using locks.

3.4.2 Multimedia workloads

3.4.2.1 MPGdec

The *MPGdec* benchmark is based on the MSSG MPEG-2 decoder. It decompresses a compressed MPEG-2 bit-stream. The original image is divided in frames. Each frame is subdivided into 16×16 pixel macroblocks. Contiguous rows of these macroblocks are called a slice. These macroblocks are then encoded independently. The main thread identifies a slice (contiguous rows of blocks) in the input stream and assigns it to another thread for decoding. The problem here is that the input stream is also variable length encoded. Thus, the main thread has to at least partly decode the input stream, in order to identify slices. This results in a staggered assignment of slices to threads and limits the scalability of extracting parallelism.

We have divided this benchmark in transactions, where each transaction is the decoding of one video frame. In Particular, the execution of a transaction comprises four phases. First, it performs variable-length Huffman decoding. Second, it inverse quantizes the resulting data. Third, the frequency-domain data is transformed with IDCT (inverse discrete cosine transform) to obtain spatial-domain data. Finally, the resulting blocks are motion-compensated to produce the original pictures.

3.4.2.2 MPGenc

The *MPGenc* benchmark is based on the MSSG MPEG-2 encoder. It converts video frames into a compressed bit-stream. The encoder uses in principle the same data structures as the decoder. The encoding process is parallelized by assigning different slices to each thread. However, since these slices can be determined very easily in uncompressed data, the encoding process can be parallelised without much effort by assigning different slices to different threads. The ALPBench version has been modified to use an intelligent three-step motion

Figure 3.1: Multi-programmed workloads evaluated in this thesis.

search algorithm instead of the original exhaustive search algorithm and to use a fast integer discrete cosine transform (DCT) butterfly algorithm instead of the original floating point matrix based DCT. Also, the rate control logic has been removed to avoid a serial bottleneck.

We have divided this benchmark in transactions, where each transaction is the encoding of one video frame. Again, the execution of a transaction comprises several phases: motion estimation, form prediction, quantization, discrete cosine transform (DCT), variable length coding (VLC), inverse quantization, and inverse discrete cosine transform (IDCT).

3.4.3 Multi-programmed workloads

We also evaluate some of our proposals with multi-programmed workloads, which consist of several program instances running at the same time using different subsets of the cores available on chip. Since it is expected that many-core architectures will also be employed for throughput computing [29] and multi-programmed workloads have different protocol requirements than parallel ap-

plications, they also constitute an interesting scenario for the evaluation carried out in this thesis, particularly in Chapter 7.

We have simulated the configurations shown in Figure 3.1, where the threads of each instance are bound to neighbouring cores. We classify workloads as either *homogeneous* or *heterogeneous*. We have created two homogeneous and two heterogeneous multi-programmed workloads. *Ocean4* and *Radix4* consist of four instances of the *Ocean* and *Radix* applications, respectively, with eight threads each one, representing homogeneous workloads. *Mix4* and *Mix8* run *Ocean*, *Raytrace* (teapot), *Water-Nsq* (512 molecules, 4 time steps) and *Unstructured*. In *Mix4* each application has eight threads. In *Mix8* two instances of each application are run with four threads each. These two workloads represent the heterogeneous and more common workloads.

Chapter 4

A Scalable Organization for Distributed Directories

4.1 Introduction

In CMP architectures, the cache coherence protocol is a key component since it can add requirements of area and power consumption to the final design and, therefore, it could restrict severely its scalability. When the CMP is comprised of a large number of cores, the best way of keeping cache coherence is by implementing a directory-based protocol, since protocols based on broadcasting requests are not power-efficient due to the tremendous number of messages that they would generate, as discussed in Chapter 2.

Directory-based protocols reduce power consumption compared to broadcast-based protocols because they keep track of the sharers of each block in a directory structure. In a traditional directory organization, each directory entry stores the sharers for each memory block through a simple full-map or bit-vector sharing code, i.e., one bit per private cache. Since the area requirements of this structure grow linearly with the number of cores in the CMP, many approaches aimed at improving its scalability have been proposed [1, 8, 26, 44, 85]. However, they do not bring perfect scalability and usually reduce the directory memory overhead by compressing coherence information, which in turn results in extra unnecessary coherence messages, wasted energy, and some performance degradation. Another alternative to the full-map scheme that also keeps precise sharing information is to have a directory structure that stores duplicate tags

of the blocks held in the private caches. This scheme has been recently used both in cc-NUMA machines as Everest [87] and in CMPs as Piranha [16] or Sun UltraSPARC T2 [111].

In tiled CMPs, the directory structure is split into banks which are distributed across the tiles. Each directory bank tracks a particular range of memory blocks. Up to now, most tiled CMP proposals assume a straightforward implementation for the directory structure based on the use of a full-map sharing code. As previously commented, this directory organization does not scale, since its size grows linearly with the number of tiles of the system. Moreover, since the directory must be stored on-chip to allow for short cache miss latencies and CMP designs are constrained by area, the directory area should represent a small fraction of the total chip.

In this chapter, we show that a directory organization based on duplicate tags, which are distributed among the tiles of a tiled CMP by following a particular granularity of interleaving can scale up to a certain number of cores, while still storing precise coherence information [99]. In particular, we show that the size of each directory bank does not depend on the number of tiles. In the proposed directory organization, each directory entry has associated a unique entry of a private cache in the system. A directory entry stores the tag of the block allocated in its corresponding entry of the private cache, a valid bit and an ownership bit. If the ownership bit is enabled the cache is known to be the owner of the block.

The size of each directory bank in the proposed organization is $c \times (l_t + 2)$, where c is the number of entries of the last-level private cache if the private caches are inclusive or the aggregate number of entries of all private caches if they are non-inclusive, and l_t is the size of the tag field. To ensure that each directory entry is associated with just one entry of some private cache, and vice versa, the directory interleaving must be defined by taking some bits of the memory address that fulfill the following condition: $bits_home \subseteq bits_private_cache_set$, i.e., the bits that define the home tile must be a subset of the bits used for indexing private caches. We also have measured the area overhead of the proposed directory organization using the CACTI tool [115], obtaining an overhead of just 0.53% when compared to the on-chip data caches considered in this thesis.

In addition, this directory organization allows us to modify the coherence protocol in order to remove extra messages caused by replacements. We have named this technique as *implicit replacements* [99]. Since each cache entry is associated to a directory entry (the same way too in case of associative caches) the requesting tile does not have to inform the directory about replacements,

because the directory knows which block is being replaced when the request for a new block arrives to it. We have found that this mechanism leads to average reductions of 13% (up to 32%) compared to a non-scalable traditional directory-based protocol that employs unlimited directory caches and informs the directory about replacements only in case of evictions of dirty blocks. Moreover, compared to a directory organization based on duplicate tags that also needs to inform the directory about evictions of clean blocks, the implicit replacement mechanism saves 33% of coherence messages on average. These reductions in network traffic are expected to result in significant savings in power consumption.

On the other hand, designing large-scale CMPs is not straightforward, and tiled CMPs are aimed at simplifying the development of these multiprocessors by duplicating identical or close-to-identical building blocks. This allows processor vendors to support families of products with varying computational power, and thus, cost. The proposed scalable distributed directory organization will allow vendors to use the same building block for designing tiled CMPs with different number of tiles.

The rest of the chapter is organized as follows. A background on directory organizations for both cc-NUMA and CMP systems is given in Section 4.2. Section 4.3 describes the scalable distributed directory organization. The implicit replacements mechanism is presented in Section 4.4. Section 4.5 shows the area requirements of the directory organization, the savings in network traffic obtained with the implicit replacements mechanism, and the small variations in applications' execution time. Section 4.6 studies the limitations of the presented directory organization and how to avoid them. And finally, Section 4.7 concludes the chapter.

4.2 Background on directory organizations

Directory-based cache coherence protocols have been used for long in shared-memory multiprocessors. These protocols introduce directory memory overhead due to the need of keeping the sharing status of a memory block in a directory structure. Traditionally, this structure has been associated with main memory. Moreover, the straightforward way of tracking sharers of a block is by using a full-map sharing code where each bit represents a core in the system and a bit is set when its core holds a copy of the block. The size of this directory structure does not scale with the number of cores in the system. In particular,

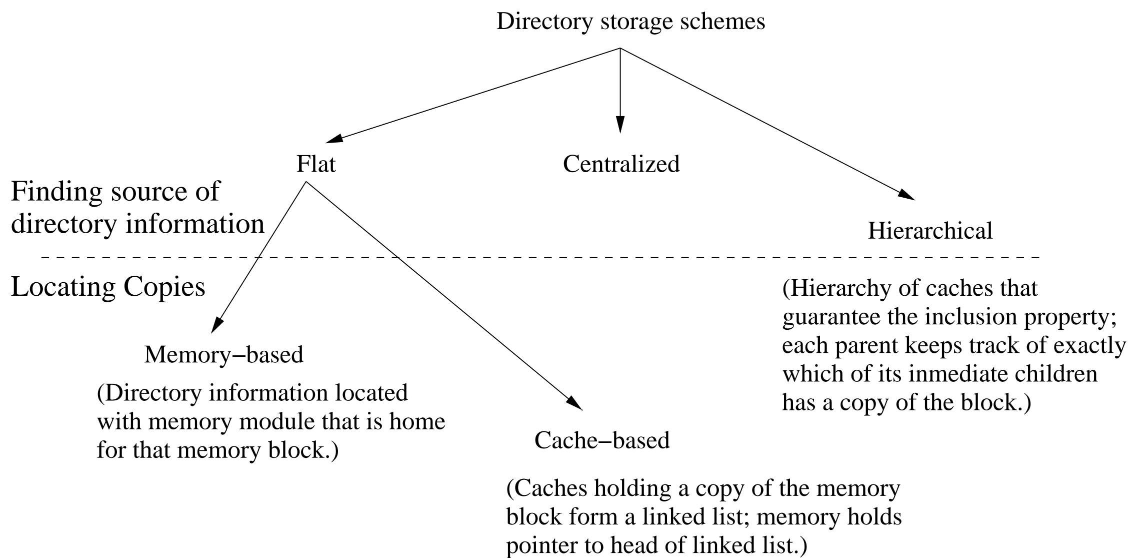


Figure 4.1: Alternatives for storing directory information.

its size is $n \times m$, where m is the number of memory entries and n is the number of cores in the system.

The directory information can represent an overhead in extra memory from the 3% as, for example, in the SGI Altix 3000 [119] to 12% as happens in other systems. However, this overhead could be much higher [5] depending on both the sharing code and the number of cores that comprise the multiprocessor system, which is definitely prohibitive.

In this chapter, we study a directory organization for tiled CMPs that addresses this problem. Next, we review the main alternatives for storing the directory information and some proposals aimed to reduce the directory memory overhead. The following subsection discusses the different directory organizations implemented in current CMP proposals and architectures.

As shown in Figure 4.1, the three main alternatives for finding the source of the directory information for a block are known as *flat directory* schemes, *centralized directory* schemes and *hierarchical directory* schemes [38]. Flat schemes are more popular than the others and they can be classified into two categories: *memory-based* schemes and *cache-based* schemes. Memory-based schemes store the directory information about all cached copies at the home node of the block. The conventional architecture which uses the full-map sharing code previously described is memory based. In cache-based schemes (also known as chained

directory schemes), such as the IEEE Standard Scalable Coherent Interface (SCI) [45], the information about cached copies is not all contained at the home but is distributed among the copies themselves. The home node contains only a pointer to the first sharer in a distributed double linked-list organization with forward and backward pointers. The locations of the copies are therefore determined by traversing the list via network transactions.

The most important advantage of cache-based directory schemes is their ability to significantly reduce directory memory overhead, since the number of forward and backward pointers is proportional to the number of cache entries, which is much smaller than the number of memory entries. Several improvements have been proposed for chained directory protocols [28, 55, 88] and commercial multiprocessors have been designed according to these schemes, such as Sequent NUMA-Q [70], which has been designed for commercial workloads like data bases or online transaction processing (OLTP), and Convex Exemplar [114] multiprocessors, destined to scientific computing. Nevertheless, these schemes increase the latency of coherence transactions as well as overload the coherence controllers and lead to complex protocols implementations [38]. In addition, they need larger cache states and extra bits for forward and backward pointers, which implies changing processor caches. These factors make more popular memory-based schemes than cache-based ones.

On the other hand, the problem of the directory memory overhead in memory-based schemes is usually managed from two orthogonal points of view: reducing directory width and reducing directory height. The width of the directory structure is given by the directory entries and it mainly depends on the number of bits used by the sharing code. The height of the directory structure is given by the number of entries that comprise the directory.

A way to reduce the size of the directory entries is to use compressed sharing codes instead of full-map. These sharing codes compress the full coherence information in order to represent it using fewer number of bits than a full-map. Compression introduces a loss of precision, i.e., when the coherence information is reconstructed, sharers that do not cache the block can appear. For example, *coarse vector* [44], which was employed in the SGI Origin 2000/3000 multiprocessor [62], is based on using each bit of the sharing code for a group of K processors. A bit is set if at least one of the processors in the group holds the memory block. Another compressed sharing code is *tristate* [8], also called superset scheme, which stores a word of d digits where each digit takes one of three values: 0, 1 and both, denoting all the sharers whose identifiers agree for both values of *both*. *Gray-tristate* [85] improves tristate in some cases using Gray

code to number the nodes. Finally, a codification based on the multi-layer clustering concept was proposed in [1], and the most elaborated proposal is *binary tree with subtrees* that uses two binary trees for including all sharers. One of the trees is computed from the home node. For the other one a symmetric node is employed..

Other authors propose to reduce the size of the directory entries by having a limited number of pointers per entry, which are chosen for covering the common case [26, 106]. The differences between these proposals are found in how the overflow situations are handled, i.e., when the number of sharers of the block exceeds the number of available pointers. The two main alternatives are to broadcast invalidation messages or to eliminate one of the existing copies. Examples of these proposals are FLASH [60] and Alewife [7].

More recently, the segment directory [35] has been proposed as an alternative to limited pointer schemes. This technique is a hybrid of the full-map and limited pointers schemes. Each entry of a segment directory consist in 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 the bits of the directory structure in this way results in a reduction of the number of directory overflows suffered by the limited pointer schemes.

On the other hand, other proposals try to reduce directory height, i.e., the total number of directory entries that are available. A way to achieve this reduction can be by combining several entries into a single one (*directory entry combining*) [105]. An alternative way is to organize the directory structure as a cache (*sparse directory*) [90, 44], or even include this information in the tags of private caches [95], thus reducing the height of the directory down to the height of the private caches. These proposals are based on the observation that only a small fraction of the memory blocks can be stored in the private caches at a particular moment of time. Unfortunately, these techniques introduce directory misses, i.e., the directory information for a memory block missing in cache is not found. This situation can be managed by broadcasting invalidation messages to all processors, which can impact on coherence traffic and applications' performance.

In general, all the described techniques result in extra coherence messages being sent or in increased cache miss rates, reducing the directory memory overhead at the expense of performance and/or power (as a consequence of an increase in network traffic).

The idea of having duplicate tags has also been used for distributed shared-memory multiprocessors as, for example, in Everest [87]. In Everest, the directory structure or complete and concise remote (CCR) directory keeps the state information (tag and state) of the memory blocks belonging to the local home that are cached in the remote nodes. In this way, CCR directory contains the same amount than memory as a sparse directory and keeps the same information than a full-map directory. However, the number of entries in the CCR directory grows linearly with the number of cores in the system.

On the other hand, other authors studied the directory interleaving to reduce the size of a distributed directory that stores a linked list of pointers to the sharers of each cache block [58]. An interleaving consisting in taking the less significant bits of the memory address allows each directory bank to have the same number of entries than the number of entries of the last-level private cache. Unfortunately, the access to the full list of pointers requires extra latency, and the size of the pointers is not completely scalable –$O(log_2 n)$–.

4.2.1 Directory organizations for CMPs

Some current small-scale CMPs keep cache coherence by implementing a snooping-based protocol, such as the IBM POWER6 architecture [63]. However, this architecture also employs directory states to filter some unnecessary coherence messages.

Other CMPs that implement a directory-based cache coherence protocol use duplicate (or shadow) tags to keep the coherence information, such as the Piranha [16] or Sun UltraSPARC T2 [111] architectures. In this case, each directory entry has fixed size and is comprised of a tag and a state field. The number of directory entries required to keep track of all blocks stored in the private caches corresponds to the sum of the entries of all private caches. In Piranha, this directory structure is centralized and, therefore, it increases with the number of cores since each core includes a private cache. Moreover, all cache misses must access this centralized directory structure, which would mean a significant bottleneck for many-core CMPs. The Sun UltraSPARC T2 architecture distributes the directory among the L2 cache banks leading to an organization similar to the studied in this chapter, but this architecture still uses a non-scalable crossbar as the interconnection network.

On the contrary, large-scale tiled CMPs require a distributed directory organization for scalability reasons. Essentially, each tile includes at least one level of private cache and a slice of the total directory. Each memory block is mapped

to a *home* tile which is responsible for keeping coherence information for that block. The identity of the home tile of a block is commonly known from the address bits ($log_2 n$ bits, where n is the number of tiles). We consider that a scalable directory organization for these systems is achieved when the size of each directory slice does not vary with the number of tiles. Obviously, the number of directory slices increases proportionally with the number of tiles, but also the number of data caches. Therefore, under this assumption the overhead introduced by the directory information does not increase with respect to data as the number of tiles grows.

The two most popular ways of organizing a distributed directory in tiled CMPs are (1) the use of directory caches [80] or (2) the inclusion of a full-map sharing code in the first level of shared caches [52]. The first technique can result in a high directory miss rate (up to 70%, as recently reported in several studies [80, 54]). The second technique avoids directory misses by using the same number of entries as the shared cache. However, this scheme can only be used when the inclusion property between private and shared caches is enforced, i.e., the shared cache must allocate an entry for each block in a private cache. In the other case, a directory cache is also needed for those blocks not allocated in the shared cache, thus introducing scalability problems and the appearance of directory misses.

Unfortunately, the inclusion property between private and shared caches could also restrict system scalability. When the number of cores grows and, therefore, the number of private caches, more pressure could be put over a particular slice of the shared cache, resulting in a larger amount of replacements. Additionally, the inclusion property forces all copies of a block to be invalidated from the private caches when the block is replaced from the shared cache, increasing the miss rate of private caches and, consequently, degrading performance. On the other hand, the size of the directory entries either does not scale with the number of cores (e.g., full-map) or does not keep precise sharing information (e.g., coarse vector [44] or limited pointers [26]).

Recently, in [80] different directory organizations have been studied for tiled CMPs which demonstrate that the organization for the directory is a crucial aspect when designing large-scale CMPs.

4.3 Scalable directory organization

In this section we show that a distributed directory organization based on duplicate tags can scale up to a certain number of cores depending on the system parameters. Moreover, the described directory organization keeps precise information about all blocks stored in private caches, i.e., directory misses only take place when the block is not stored in any private cache and, therefore, no extra coherence actions are needed as consequence of directory misses.

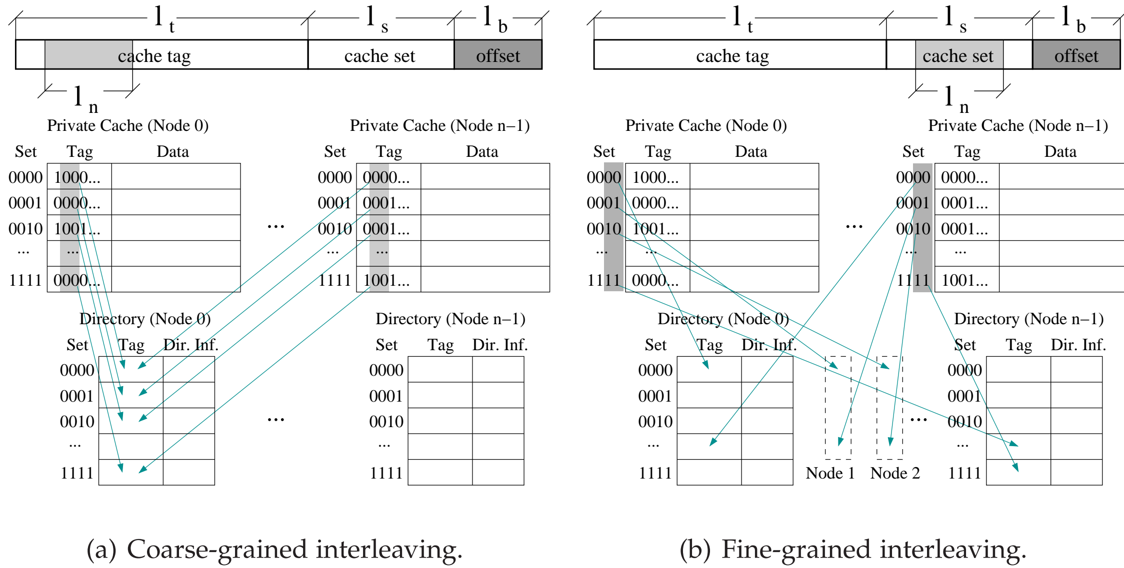
To guarantee the scalability of the directory it is necessary to keep fixed both the size of each directory entry (directory width) and the number of entries per slice (directory height). The use of duplicate tags as directory entries makes scalable the directory width, since each directory entry is comprised of a tag and a state field. On the other hand, the total number of directory entries required to track all blocks stored in the private caches should be the same as the number of entries of all private caches. This rule is always fulfilled for centralized directories, but not when the directory is distributed.

As previously discussed, tiled CMPs split the directory structure into banks which are distributed across the tiles. Each directory bank tracks a particular range of memory blocks. If all blocks stored in the private caches map to the same bank, the directory of this bank can overflow, thus requiring more entries to keep all the directory information. In this case, the minimal number of entries required to store all duplicate tags increases linearly with the number of tiles.

In the first subsection, we discuss how the granularity of the directory interleaving can affect the maximum number of entries required by each directory bank and, therefore, the scalability of the directory. Then, we define the conditions that are necessary for the directory structure to scale with the number of cores. We also describe the structure of the directory and how precise sharing information can be obtained from it. Finally, we comment on the requirements of a cache coherence protocol that implements this directory organization.

4.3.1 Granularity of directory interleaving

One important decision when designing the memory hierarchy of a tiled CMP is the granularity of the directory interleaving. Each memory block must map to a particular tile, which is the home tile for that block. This tile is responsible for keeping cache coherence for that block and, therefore, must store the directory information necessary to perform that task. On the other hand, if the tiled CMP includes an on-chip cache which is logically shared by all cores (but obviously



(a) Coarse-grained interleaving.

(b) Fine-grained interleaving.

Figure 4.2: Granularity of directory interleaving and its effect on directory size.

distributed among the tiles), it is also necessary to define an interleaving for that cache. Cache and directory interleavings may be different. However, this decision incurs in extra coherence messages between the tile where the directory information is stored and the tile where data can reside, thus making the cache coherence protocol less efficient and more complex. Therefore, it is desirable that both the shared cache and directory have the same interleaving.

The directory can be easily distributed among the tiles of the CMP by taking $log_2 n$ (l_n) bits of the block address, where n is the number of tiles of the system (physical address mapping). These bits denote the tile where the directory information for each block can be found. The position of these bits defines the granularity of the interleaving, and as shown in Figure 4.2, the number of entries required by each directory bank to be able to keep the sharing information of all cached blocks belonging to it.

In Figure 4.2, we can observe two alternative ways of distributing the blocks among the tiles of the CMP and their consequences. Looking at the address of a memory block we can distinguish three main fields from the point of view of a private cache: the block *offset* (l_b) which depends on the size of blocks stored in cache, the *cache set* (l_s) in which the block must be stored and the *cache tag* (l_t) used to identify a block stored in a cache.

If the l_n bits chosen to define the home tile belong to the cache tag field, huge

continuous memory regions map to the same tile (coarse-grained interleaving). Under this assumption, all the blocks stored in the private caches could map to the same directory bank. This situation is shown in Figure 4.2(a). Assuming that the number of sets and the associativity of the private caches are s and a, respectively, the number of entries required by each directory to keep the information of the cached blocks mapped to it is $n \times s \times a$ (or $n \times c$, where c is the number of entries of each last level private cache). In particular, each directory must have s sets of $n \times a$ ways each.

Otherwise, if the l_n bits belong to the cache set field, memory is split in a great amount of small regions that map to different tiles in a round-robin fashion (fine-grained interleaving), as shown in Figure 4.2(b). Under this assumption, each entry of each L1 cache maps to only one entry of the directory. Therefore, the number of entries required by each directory bank will be $s \times a$. In particular, each directory bank must have s/n sets of $n \times a$ ways each one. The size required by this structure is c, which scales with the number of tiles of the system. The proposed directory organization uses an interleaving where the l_n bits belong to the cache set fields.

4.3.2 Conditions required for ensuring directory scalability

A distributed directory organization with the same number of entries as the private caches needs a function that maps private cache entries to directory entries so that (1) a particular memory block always has its duplicate tag in the same directory bank (the home one) regardless of the cache wherein the block is stored and (2) the function is injective, i.e., one-to-one. The first rule guaranties that sharing information for a particular block can always be found in its home bank. The second one ensures that the number of directory entries corresponds to the number of cache entries, thus achieving the scalability of the directory.

To describe the aforementioned function, let's first consider systems with just one level of direct mapped private caches. Each cache entry can be uniquely defined by a tuple of $(core, set)$ and, in the same way, each directory entry can be defined by the tuple $(home, set)$. Due to the first rule, the bits used to select the home cannot be taken from the bits used to identify the core since, in that case, a block can map to any tile depending on the cache where it is stored. Therefore, the bits used to select the home must be a subset of the bits used to select the cache set, i.e., $bits_home \subseteq bits_private_cache_set$. This mapping rule guaranties a scalable distributed directory organization. However it also has a restriction. More specifically, it can only be used when the number of

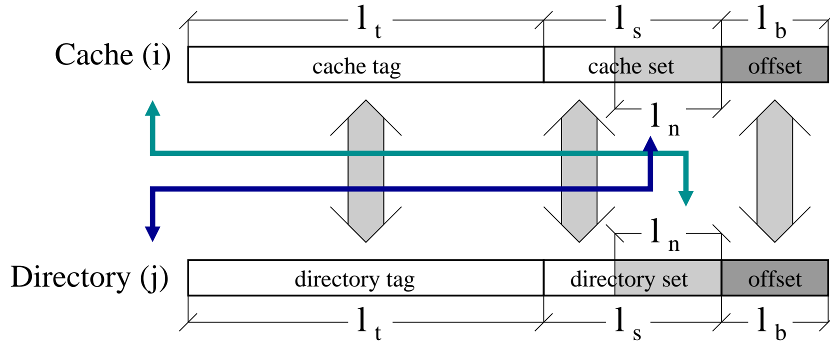


Figure 4.3: Mapping between cache entries and directory entries.

sets of the private cache is greater or equal than the number of tiles of the system $(num_tiles \leq num_private_cache_sets)$. Later on, this restriction will be discussed more thoroughly.

When we consider associative caches, each cache entry can be defined by the tuple $(core, set, way)$, and each directory entry by the $(home, set, way)$ one, assuming that both structures have the same associativity. Again, due to the first rule, the bits used to select the home cannot be taken neither from the bits identifying the core nor from the bits identifying the way, so that the rule that guaranties scalability is respected.

Finally, regarding private caches organized in several cache levels, if the cache levels are inclusive the scalability is achieved in the same way as with just the last (larger) cache level. However, when the inclusion policy is not ensured, each home tile should have as many directory banks as caches in the private hierarchy, each one with the same number of entries as each cache. In this case, the scalability can be achieved when the number of tiles is not greater than the number of sets of the small cache in the hierarchy.

By using this mapping function we can see that the associativity required for each directory bank is the same as the one used for the private caches. Therefore, the associativity of the directory can be reduced from $n \times a$, as discussed in the previous section, to a by taking the number of the tile in which the block is cached as part of the set bits. In this way, the number of sets grows from s/n to s. In conclusion, each directory bank requires the same number of sets and associativity as a private cache.

Figure 4.3 shows the mapping function for scalable distributed directory organizations. Home tiles are chosen by taking l_n bits of the ones used to select

the set in the private cache (e.g., the less significant ones). Moreover, the set in a directory bank is obtained from both the remaining bits of the cache set and the number of the tile where the block is stored. Likewise, l_n bits of the directory set are used to identify the tile that holds the copy in its private cache. Although in the scheme the l_n bits are the less significant ones of the set field, they can be any set of bits of that field.

When the number of tiles n is greater than the number of sets of the L1 cache s, the number of entries required by the directory is $n \times a$, but this is not the common case. In any case, the number of entries required by this directory organization is $max\{s, n\} \times a$, that is to say, the number of entries completely scales for values of s greater than n.

4.3.3 Directory structure

In the previous sections we have described how the number of entries of the directory can scale with the number of tiles. However, the size of the entries commonly used to keep the directory information does not scale with the number of tiles (e.g., n for a full-map sharing code, or $p \times log_2 n$ when p pointers are used to locate the cached copies). One way to keep constant the size of the directory entries is storing duplicate tags. Particularly, our proposal for a scalable directory stores in each entry the tag of the block plus two extra bits. The first bit is the valid or presence bit. If this bit is set the block is known to be stored in the cache entry associated with this directory entry. Remember that each directory entry is associated with only one cache entry (injective function). This bit is used to locate all the copies of the block on a write miss. The second bit is the ownership bit and when it is set the cache entry is known to have the ownership of the block. This bit is used to enable the implementation of a MOESI protocol, and it identifies the cache that must provide the data block on a cache miss.

Since we only store the tag of the block and two more bits in each directory entry, and the tag bits keep invariant with the number of tiles, the size of the directory keeps constant as the number of cores of the CMP increases. The total size of each directory bank is $c \times (l_t + 2)$.

Considering the mapping function presented in the previous section, the coherence information for a particular block can be obtained from the home directory bank as shown in Figure 4.4. A block is stored in a particular private cache whether there is a hit in the directory bank for the corresponding set and the valid bit is enabled. The corresponding set is calculated by replacing the l_n bits that identify the home directory with the l_n bits that identify the tile which

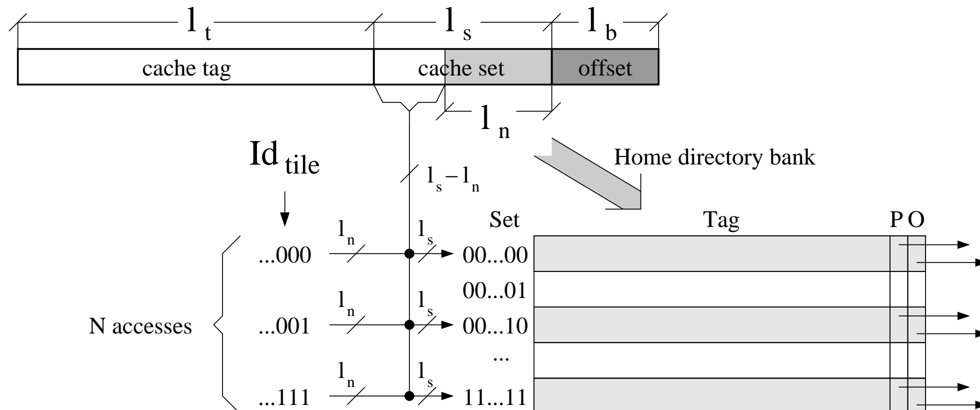


Figure 4.4: Finding coherence information (P=presence bit; O=ownership bit).

contains that cache. If the ownership bit is also enabled, that cache is the owner of the block. Therefore, the identifier of the set for a block depends on the cache where it is stored. By searching this information in the corresponding *n* directory entries (one per each private cache) the complete directory information is obtained. This search can be performed in parallel to accelerate the access to the directory information.

Updating the directory information only requires to modify few bits. On a write miss, invalidation messages are sent to the sharers and their corresponding presence bits are disabled. The directory entry for the new owner of the block is with the tag of the block and both the valid and the ownership bits are enabled. Adding a new sharer only requires writing the tag of the block in the corresponding directory entry and setting the valid bit.

4.3.4 Changes in the cache coherence protocol

The cache coherence protocol required by this directory organization is similar to the required by a directory-based protocol that uses directory caches with a non-scalable full-map sharing code in each entry. However, the use of a limited number of duplicate tags requires some extra modifications.

As happens in Piranha [16] and Everest [87], replacements of shared blocks must be notified to the home tile. These notifications are necessary to deallocate an old directory entry, in order to allow the new block to use that entry. Notifications of these evictions make the cache coherence protocol more complex.

Moreover, to avoid race conditions replacements are usually performed in three phases, a fact that entails extra network traffic.

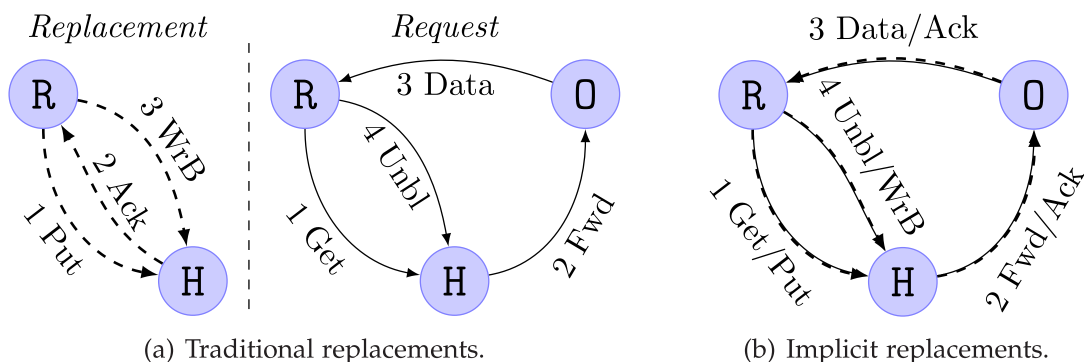
In systems with unordered networks, as tiled CMPs are, the request for a block can reach the home tile before the replacement caused by that block. If the directory set for that block has valid information in all the ways, the request must wait until the replacement deallocates one of the entries. This can result in extra cache miss latency. To avoid these issues and to remove the network traffic generated by replacements, we propose the implicit replacements mechanism described in the following section.

4.4 Implicit replacements

The proposed directory organization allows us to slightly modify the coherence protocol in order to remove the messages caused by replacements. This is achieved by performing the replacements in an implicit way along with the requests which cause them. Note that we use as base protocol a directory protocol that unblocks the home tile from the requester (see Section 2.5 for a detailed discussion of this issue). In this way, we can merge all messages generated by evictions with messages generated by requests.

There are two main observations that allow our proposal for scalable directory organization to support implicit replacements. Firstly, since we employ the bits used to select the set of private caches to associate cache entries to directory entries we ensure that the evicted and the requested blocks map to the same home tile and, therefore, to the same directory bank. Note that if a coarse-grained interleaving was chosen, these blocks could map to different directory banks (depending on the value of the tag). Secondly, each cache entry is associated with only one directory entry (the same way too), and vice versa. In this way, both the directory and the requesting cache know the address of the evicted block and it is not necessary to attach it to the request messages. Therefore, the size of coherence messages does not change considerably. It is only necessary the addition of a field informing about the way within the set where the requested block is going to be stored (2 bits in our case), which is the same way where its duplicate tag is stored in the directory.

In Figure 4.5(a), we can see how a replacement is usually performed. When a block must be stored in cache and the corresponding set is full, the less recently used block must be evicted (we assume a pseudo LRU eviction policy). In current directory protocols evictions of shared blocks are usually performed trans-



(a) Traditional replacements. (b) Implicit replacements.

Figure 4.5: Differences between the proposed coherence protocol and a traditional coherence protocol.

parently without informing the directory. However, as previously discussed, when the directory is organized with duplicate tags, replacements must be notified even for blocks in shared state. Moreover, evictions of dirty blocks must writeback data to the next cache level. For simplicity, these writebacks are performed in a three-phase transaction as illustrated in Figure 4.5(a) (Replacement). First, the cache asks the home tile permission to *writeback* the block (*1 Put*). Then the home tile confirms the transaction ((*2 Ack*), and finally the block is sent to the next cache level ((*3 WrB*). Figure 4.5(a) (Request) shows how a cache-to-cache transfer miss is solved. We consider that the unblock message is sent by the requesting tile, as described in Section 2.5. Requests are sent to the home tile to get the directory information (*1 Get*), and then are forwarded to the owner cache (*2 Fwd*) where the data is provided (*3 Data*), or the data is directly provided from the L2 cache in the home tile. Finally, the requesting cache informs the home tile that another cache miss for this memory block can be processed (*4 Unbl*).

Figure 4.5(b) shows how implicit replacements are performed along with the requests that cause them. On each cache miss an MSHR (Miss Status Hold Register) entry is allocated with the information about the request. However, in our proposal we also need to store the address of the evicted block (if any), so that our MSHR has two address fields instead of one. If there is an evicted block we also allocate a new entry for it in the MSHR indicating the state of that block. Moreover, the way where the new block will be stored is specified in the request message (*1 Get/Put*). When this message reaches the home tile, another two MSHR entries must be allocated, as usual. One of the entries stores both addresses. Note that storing the address of the evicted block can be replaced

with a pointer to the MSHR entry where this address is stored, thus reducing its size. The acknowledge of the replacement is forwarded along with the request (*2 Fwd/Ack*). When the data arrives to the requesting cache (*3 Data/Ack*) both MSHRs are deallocated and the writeback is performed along with the unblock message (*4 Unbl/WrB*), thus allowing the directory to process the subsequent requests for both blocks. Another advantage of this protocol is that it avoids the race conditions caused by replacements that were discussed earlier, because now the replacement is implicit into the request and, therefore, both messages reach the home tile at the same time.

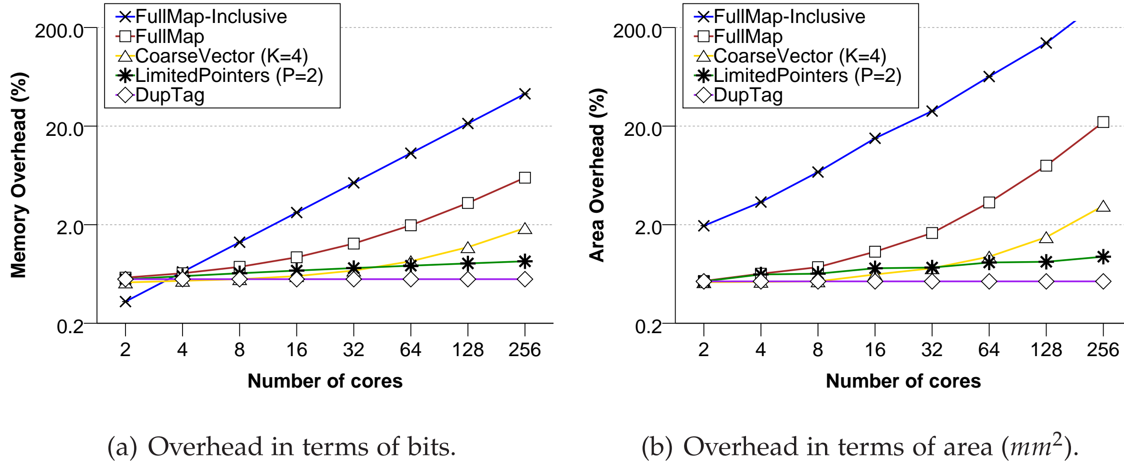
While the described mechanism is employed for evictions of dirty blocks, evictions of shared blocks are avoided in an easier way since we know the way within the cache set where the block is going to be stored. When the request arrives to the home tile, the directory information for the new block will be stored in the same way as in the cache. Therefore, the tag of the new block replaces the tag of the evicted block, performing the notification without requiring extra coherence messages.

4.5 Evaluation results and analysis

In this section, we analyze the area requirements of the proposed directory organization and the network traffic that can be saved with the implicit replacements mechanism. We also show that the execution time obtained when an unlimited directory cache is employed is comparable to the execution time reached with the proposed directory organization. However, a slight performance degradation can appear when the implicit replacements mechanism is implemented. Area requirements have been calculated using the CACTI tool, while both network traffic and execution time have been measured using the GEMS simulator enhanced with SiCoSys. The cache coherence protocols evaluated in this section implement MOESI states and the unblock message is issued by the requesting tile, as discussed in Section 2.5. However, this proposal can also be used with a MESI protocol. As previously commented, since the unblock message is sent by the requesting tile, the writeback message can be merged with the unblock one.

4.5.1 Directory memory overhead

In this section we study the directory memory overhead of our proposed organization compared to some of the schemes described in Section 4.2. Figure 4.6



(a) Overhead in terms of bits.

(b) Overhead in terms of area (mm^2).

Figure 4.6: Directory memory overhead as a function of the number of tiles.

shows this overhead as a function of the number of tiles in the system. The directory organizations shown in the graphs are *FullMap-Inclusive*, *FullMap*, *CoarseVector (K=4)*, *Limited pointers (P=2)*, and finally the organization presented in this chapter (*DupTag*). The characteristics of all these schemes are described below. The overhead of the directory structure has been calculated with respect to both the L1 and the L2 caches taking into account the size of the tag field. Figure 4.6(a) shows the directory memory overhead in terms of number of bits added by the coherence information (and the tags that the storage of the coherence information entails). Figure 4.6(b) shows the directory area overhead in terms of mm^2. For both graphs, we assume a tiled CMP with the parameters described in Chapter 3.

FullMap-Inclusive is currently used in some proposed tiled CMPs in which the L1 and the L2 are inclusive (the L2 contains all blocks held in the private L1 caches). The directory is stored in the tags' part of the L2 caches, thus removing the need of an extra directory cache. As discussed in Section 4.2.1, enforcing the inclusion property between private and shared caches may not be scalable. Moreover, the use of a full-map sharing code and the fact that it is introduced one sharing code field per L2 cache entry makes the overhead of this scheme increase linearly with the number of cores.

Another approach aimed at reducing the directory storage is to employ on-chip directory caches. For this evaluation, we assume directory caches with the same number of entries (same number of sets and associativity) as a private L1

cache. Note that due to the limited number of entries used in each directory cache, it could be needed to re-use an existing entry to store directory information for a new block. This implies invalidating all copies of a block when its directory information is evicted from the directory cache. However, this option can increase the miss rate of private caches. Another option is keeping an off-chip directory with the evicted, but this scheme results in extra storage, or broadcasting requests to all cores when a directory entry for a particular block is not found in the directory cache, but this approach results in extra network usage. As discussed in Section 4.3.1, for a fine-grained interleaving and an associativity of $a \times n$, all necessary directory information can be stored, but in this case the associativity of the directory cache is also not scalable.

In any case, when these directory caches store a full-map sharing code (*FullMap* case), the area overhead grows up to 20% for 256 tiles, which is prohibitive. Compressed sharing codes can reduce this overhead by losing accuracy. In *CoarseVector (K=4)* the sharing code is compressed by using one bit per each group of four tiles. The bit is set if at least one of the four tiles holds a copy of the block. Although the area of the directory structure is reduced, it still increases with the number of tiles. In *Limited pointers (P=2)* only two pointers are used to identify the caches that share each memory block. When the number of sharers is greater than two, writes are performed by broadcasting invalidation messages (a broadcast bit is also required per entry). The area required by this organization is $2 \times log_2 n$, which scales better than the former sharing codes. However, differently from the proposed organization, compressed sharing codes fall into extra coherence messages since they do not store precise information about all the caches that hold the blocks.

Finally, we can see that by combining a fine-grained interleaving and duplicate tags (*DupTag* case) we can achieve a completely scalable directory organization which keeps on-chip all the information necessary to keep cache coherence and, therefore, neither extra invalidation messages nor off-chip directory structures are required. The area overhead of this directory organization is 0.53% when compared to the area taken by L1 and L2 caches.

4.5.2 Reductions in number of coherence messages

In this section, we evaluate the results in terms of number of messages generated by the cache coherence protocol. Figure 4.7 shows the number of coherence messages generated by several directory organizations. This number has been normalized with respect to a directory-based protocol that uses unlimited on-

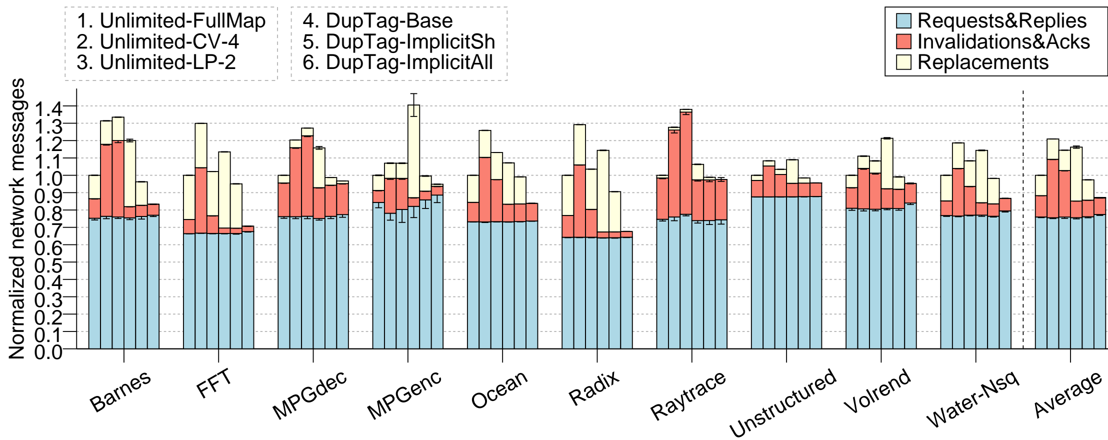


Figure 4.7: Reductions in number of coherence messages.

chip directory caches with a full-map sharing code (*Unlimited-FullMap* case). *Unlimited-CV-4* and *Unlimited-LP-2* represent configurations with unlimited directory caches that store a coarse vector with groups of four tiles and a limited pointer scheme with two pointers, respectively. Note that by simulating unlimited directory caches we do not account for the extra invalidation messages that are necessary when a directory entry has to be evicted. Finally, we show the directory organization based on duplicate tags (*DupTag-Base*), and the optimizations entailed by the implicit replacement mechanism. *DupTag-ImplicitSh* only removes evictions of shared blocks. In *DupTag-ImplicitAll*, all evictions are performed along with request messages.

First, we can observe that the number of invalidation messages generated by the protocols with compressed sharing codes is greater than the number of invalidation messages generated by a protocol with a precise sharing code. All *DupTag* configurations slightly reduce the number of invalidation messages compared to *Unlimited-FullMap*. This is because replacements of shared blocks are not notified for the *Unlimited-FullMap* scheme and, therefore, unnecessary invalidation messages, which find the block evicted from cache, are issued for some misses.

On the other hand, the implicit replacements mechanism removes the coherence messages caused by evictions. In the three first configurations shown in Figure 4.7, evictions of shared blocks are performed without informing the directory structure. However, as discussed in Section 4.3.4, an organization based on duplicate tags requires informing the directory in case of these evictions,

which increases network traffic. *DupTag-ImplicitSh* performs evictions of shared blocks in an implicit way, which reduces significantly the traffic caused by replacements. *DupTag-ImplicitAll* is more aggressive and removes all replacement messages by also merging evictions of private or owned blocks with the request that causes the eviction.

In general, the directory organization based on duplicate tags is the one with lowest invalidation messages and the implicit replacements mechanism is able to remove the coherence messages caused by L1 replacements. Average reductions of 13% and up to 32% for *Radix* are obtained compared to *Unlimited-FullMap*. However, if we compare the implicit replacements mechanism with the *DupTag-Base* configuration we can save 33% of coherence messages on average. Moreover, if we consider smaller caches, reductions in terms of network messages are higher.

4.5.3 Impact on execution time

In this section we analyze the performance of the evaluated protocols. Although our directory organization requires significantly less area than the other schemes (even when they employ directory caches with the same size and associativity as the L1 caches), we consider the same access latency for all the directory structures. Moreover, it is important to note that by simulating unlimited directory caches we do not account neither for the extra latency caused by directory misses nor for the increase in cache miss rate caused by the invalidation messages issued when a directory entry has to be evicted from the directory cache. Therefore, despite of the area requirements, our base configuration is an ideal directory-based protocol.

In general, we can see in Figure 4.8 that all the protocols achieve similar results under the described conditions. Particularly, *DupTag-Base* obtains the same performance as *Unlimited-FullMap*, but the first one only requires a small amount of area. When the implicit replacements mechanism is implemented, the execution time slightly increases. The more aggressive is the mechanism, the more execution time is degraded. However, for the most aggressive mechanism *DupTag-ImplicitAll*, the average performance degradation is just 2%. This degradation comes mainly as consequence of the fact that replacements take more time to be completed in the implicit replacements mechanism than in the traditional replacements mechanism, thus blocking some entries at the home tile for longer.

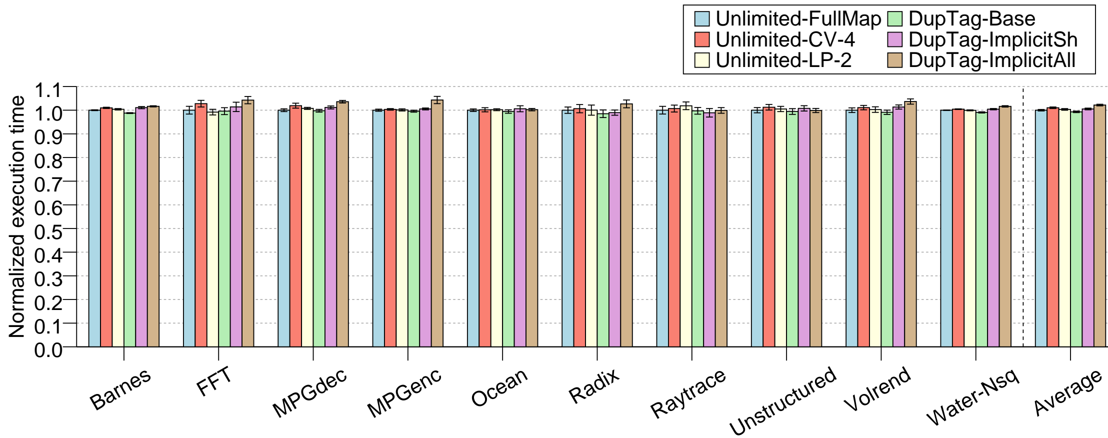


Figure 4.8: Impact on execution time.

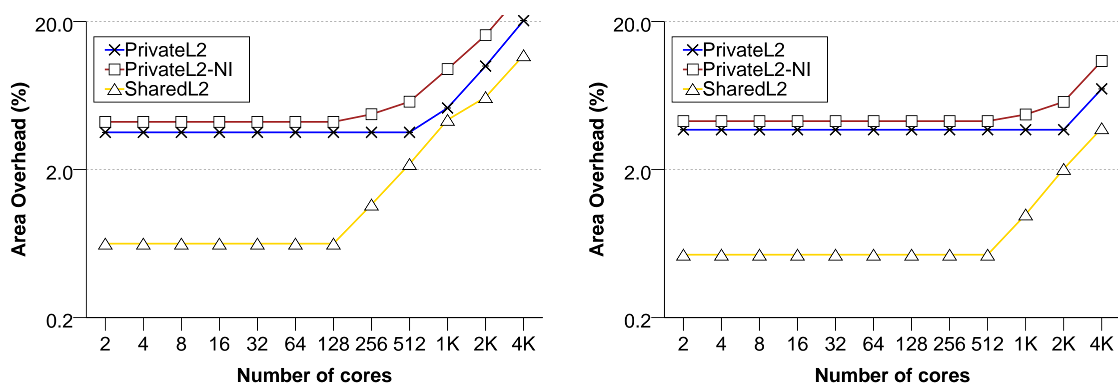
4.6 Managing scalability limits and locality issues

The directory organization presented in this chapter has two main limitations, which are discussed in this section. First, the scalability is limited to configurations for which the number of private cache sets is greater than the number of tiles. Second, we assume that memory blocks are distributed among the tiles in a round-robin fashion with a fine-grained interleaving. This distribution does not consider the locality of the accesses to the shared cache, and may result in longer latencies for L2 cache accesses.

4.6.1 Scalability limits

We first focus on the scalability limitations. When the number of cores grows up to the point where there are more cores than cache sets, the number of sets required by each directory bank to allow it to store all duplicate tags must be equal to the number of cores, instead of the number of cache sets. More specifically, the number of entries required by each directory bank is $max(c, n \times a)$, where c is the number of entries of a private cache, n is the number of tiles in the system, and a is the associativity of private caches.

Nowadays tiled CMPs could be designed with this scalable directory. One example is the Tilera tile64 [19], which is a 64-tile CMP with 8KB direct mapped L1 instruction and data caches and 64KB 2-way associative L2 caches per tile. Both caches store blocks of 64 bytes. As we can see in Figure 4.9(a), cache



(a) Tilera Tile64: 8KB direct mapped L1 cache and 64KB 2-way associative L2 cache.

(b) Simulated tiled CMP: 64KB 4-way associative L1 cache and 512KB 8-way associative L2 cache.

Figure 4.9: Directory memory overhead for two different systems and several configurations.

coherence could be kept by using this directory organization, while still being scalable for up to 128 tiles if L2 caches are shared. If we consider private L2 caches (as Tilera tile64 does) the scalability limit is 512 tiles. Obviously, with private L2 caches, the directory structure requires more area, since the amount of blocks that can be allocated on private caches is higher. Considering the system that we have simulated in this chapter (Figure 4.9(b)), the directory organization can scale up to 512 tiles (or 2048 in case of implementing private L2 caches). The *PrivateL2-NI* line in Figure 4.9(a) represents the scalability of the directory when L1 and L2 private caches do not enforce inclusion.

Because it is expected that the number of tiles will increase while the size of the L1 caches will keep more or less constant, this directory organization could have only limited scalability. This may be partly remedied for future systems if they include several levels of inclusive private caches. Fortunately, it is expected that the next generation of commodity multiprocessors include two or more levels of inclusive private caches on-chip and a shared last-level on-chip cache, as happens in the new Intel Nehalem [53] and AMD Barcelona CMP architectures. In this way, the number of sets of the last-level private cache can increase as the integration scale becomes higher. As we discussed in Section 4.3, under this scenario the directory structure scales up to a number of tiles less

or equal than the number of sets of the last level of private caches and, therefore, it will be able to scale for a larger number of tiles.

4.6.2 Locality of the accesses to the shared cache

Regarding the locality problem, there are two main ways of reducing the latency of the accesses to a shared L2 cache. One of them is to map memory regions to the tile whose processor is more frequently requesting them (e.g., a first-touch policy [34]). Another approach is to use the local L2 cache bank as a victim cache, to avoid accessing the home tile for some L1 misses (i.e., victim replication [123]).

A first-touch policy maps a memory page to a tile according to the first reference to that page. On a page fault, the OS looks for a free physical page address that maps to the tile whose processor is requesting the block. Since address translation is performed at page size granularity, the granularity of the interleaving must be at least the size of a memory page. Under these assumptions, the bits that identify the home tile cannot be the less significant ones, i.e., they cannot be chosen from the page offset. As discussed in Section 4.3.1 a coarser granularity for the interleaving restricts even more the scalability of the directory.

A solution to this scalability problem is to change the private cache indexing, i.e., the address bits used to define the cache set. If these bits are chosen from the bits that identify the home tile, the scalability will be the same as if block-grained interleaving were used. Remember that the rule to achieve a scalable directory is $bits_home \subseteq bits_private_cache_set$. Unfortunately, this private cache indexing can increase the cache miss rate. This happens because the same bits used for identifying the home tile are used for indexing the block in the private cache, and we are trying to assign the same local home to the blocks requested by the local cache (first touch policy). Therefore, there may be some sets that are almost unused in the private cache, thus impacting in cache hit rate.

On the other hand, victim replication is an approach that improves locality without changing the home directory, but instead, replicating blocks in the local shared slice when they are evicted from a private cache. This approach allows to use the described scalable directory organization. However, since a block can be either in any private cache or in any slice of the shared cache, the number of entries of each directory slice must be the same as the number of entries of the private and shared caches, as previously described for a non-inclusive private cache hierarchy (*PrivateL2-NI* label in Figure 4.9).

4.7 Conclusions

In CMP architectures, the cache coherence protocol is a key component since it can add requirements of area or power consumption to the final design and, therefore, could restrict severely its scalability. Although directory-based cache coherence protocols are the best choice when designing many-core CMPs, the memory overhead introduced by the directory structure may not scale gracefully with the number of cores, specially when the coherence information is kept by using a full-map sharing code.

In this chapter, we show that a directory organization based on duplicate tags, which are distributed among the tiles of a tiled CMP by following a particular granularity for the directory interleaving, can scale up to a certain number of cores. The rule to achieve this scalability is $bits_home \subseteq bits_private_cache_set$. Therefore, a directory can scale meanwhile the number of cores of the CMP is less than the number of sets of the private L1 cache. Since the bits used to index private caches are commonly the less significant ones (without considering the block offset), it is preferable for the interleaving to have block granularity in order to achieve maximum scalability.

We show that, under these conditions, the size of each directory bank does not depend on the number of tiles in the system. The total size of each directory bank in the studied organization is $c \times (l_t + 2)$, where c is the number of entries of the private L1 cache and l_t is the size of the tag field. We also have measured the area overhead of the proposed directory organization with the CACTI tool obtaining an overhead of just 0.53% with respect to the area taken by the on-chip data caches. Moreover, since the structure of each directory bank does not change with the number of tiles, the same building block could be used for systems with different number of tiles, thus making easier the design of tiled CMPs with varying number of tiles.

We have also redesigned the cache coherence protocol to take full advantage of this directory organization. In particular, since each directory entry is mapped to only one cache entry, we can perform the replacements in an implicit way along with the requests which cause them, thus saving the network traffic introduced by replacements. This technique called *implicit replacements* leads to average reductions of 13% (up to 32%) compared to a traditional full-map directory with unlimited caches. Moreover, compared to a directory organization based on duplicate tags that needs to inform the directory about evictions of shared blocks, the implicit replacements mechanism saves 33% of coherence messages on average. These reductions in network traffic finally will result in

significant savings in power consumption. We also have shown that the impact in terms of execution time of the implicit replacement mechanism is negligible.

Finally, we also study the constrains of the proposed scalable directory organization and discuss how future chip multiprocessors can deal with them.

Chapter 5

Direct Coherence Protocols

5.1 Introduction

Directory-based cache coherence protocols have been typically employed in large-scale systems with point-to-point interconnection networks (as tiled CMPs are). Since these interconnects do not guarantee the total order of the coherence messages traveling across them, the home tile of each block is responsible for serializing the different requests issued by several cores. The home tile also keeps the directory information for the memory blocks that map to it. In this way, when a cache miss takes place, the request is sent to the corresponding home tile, which determines when the request must be processed and, then, performs the coherence actions that are necessary to satisfy the cache miss. These coherence actions mainly consist in forwarding the request to the cache that must provide the data block, and sending invalidation messages in case of a write miss.

Unfortunately, these protocols introduce indirection in the critical path of cache misses, because every cache miss must reach the home tile before any coherence action can be performed. This indirection in the access to the home tile adds unnecessary hops into the critical path of the cache misses, finally resulting in longer cache miss latencies compared to snooping-based protocols, which directly send requests to sharers.

In addition, the number of cache misses suffering from indirection increases with tiled CMPs that distribute memory blocks among home tiles through a physical address mapping, as described in several proposals [52, 123] and CMP

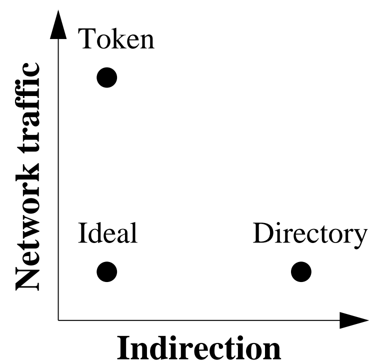


Figure 5.1: Trade-off between *Token* and directory protocols.

systems [63, 111]. In a physical address mapping, the home tile of a memory block is calculated by taking $log_2 n$ bits from the block address, where n is the number of tiles in the system. Typically, these $log_2 n$ bits are taken from the less significant bits of the block address. Since this mapping distributes memory blocks among tiles in a round-robin fashion without considering the cores requesting each block, the probability of accessing a remote home tile increases and, as consequence, the number of misses with indirection.

As discussed in the introduction and background chapters of this thesis, an alternative approach that avoids indirection is *Token* [78]. *Token* broadcasts requests directly to all cores. In this way, caches can provide data when they receive a request (no indirection occurs). Unfortunately, the use of broadcast increases network traffic and, therefore, power consumption in the interconnection network, which has been previously reported to constitute a significant fraction (approaching 50% in some cases) of the overall chip power [71, 116].

Since directory protocols entail indirection, which increases cache miss latency, and *Token* significantly increases network traffic, it is necessary to redesign cache coherence protocols in order to obtain the best of both protocols and get rid of their drawbacks. Figure 5.1 shows the trade-off between *Token* and directory protocols [74]. An ideal protocol for tiled CMPs would avoid the indirection of the directory protocols without relying on broadcasting requests.

In this chapter, we present direct coherence protocols [97, 98], a family of cache coherence protocols that meets the advantages of *Token* and directory protocols and avoids their problems. In direct coherence protocols, the task of storing up-to-date sharing information and ensuring ordered accesses for every memory block is assigned to one of the caches that shares the block, particularly

the one that provides the block on a cache miss. As we discussed in Chapter 2, the cache that provides the copy of a block is the one that has the ownership property over the requested block, i.e., the *owner* cache. Indirection is avoided by directly sending the requests to the owner cache instead of to the home tile, where coherence information resides in a directory protocol.

In this way, direct coherence protocols reduce the latency of cache misses compared to a directory protocol by sending coherence messages directly from the requesting caches to those that must observe them, as it would be done in *Token*, and reduce network traffic compared to *Token* by sending just one request message on every cache miss, which also translates into improvements in execution time.

Although direct coherence protocols can obtain significant improvements over a directory protocol for cc-NUMA architectures, as we showed in [97], in this chapter we focus on their implementation for tiled CMPs with a shared L2 cache level [98]. We call this implementation *DiCo-CMP*. In *DiCo-CMP*, the identity of the owner caches is recorded in a small structure called *L1 coherence cache* associated to every core. To achieve accurate owner predictions, this structure can be updated whenever the owner tile changes through control messages called *hints*. Additionally, since the owner cache can change on write misses, another structure called *L2 coherence cache* keeps up-to-date information about the identity of the owner cache. This L2 coherence cache replaces the directory cache required by directory protocols and is accessed each time a request fails to locate the owner cache.

The evaluation carried out in this chapter shows that *DiCo-CMP* achieves improvements in total execution time of 9% on average over a directory protocol and of 8% on average over *Token*. Moreover, our proposal reduces network traffic up to 37% on average compared to *Token* and, consequently, the total power consumed in the interconnection network. Compared to a directory protocol, our proposal obtains similar traffic requirements.

The rest of the chapter is organized as follows. In Section 5.2, we provide a review of the related work and introduce the lightweight directory architecture (an architecture for the directory that stores the directory information in the private caches). Section 5.3 describes direct coherence protocols, the modifications required in the structure of the tiles of a tiled CMP, and the different ways of updating the L1 coherence cache. Section 5.5 discusses the area and power requirements of the different mechanisms used for updating the L1 coherence cache in direct coherence protocols. Section 5.6 shows the performance results obtained by our proposal. And finally, Section 5.7 concludes the chapter.

5.2 Related work and background

In this chapter, we compare *DiCo-CMP* against two cache coherence protocols aimed at being used in CMPs from both the embedded and the desktop domains: *Token* and a directory protocol for CMPs. Both protocols have been discussed in Chapter 2. First, this section comments on some of the works related to direct coherence protocols. Then, it describes the lightweight directory architecture [95, 102], which is a proposal previously developed by us, as part of the background for direct coherence protocols, since it also keeps the directory information along with the blocks stored in the private caches.

In the shared-memory multiprocessors domain, Acacio *et al.* propose to avoid the indirection for cache-to-cache transfer misses [2] and upgrade misses [3] separately by predicting the current holders of every cache block. Predictions must be verified by the corresponding directory controller, thus increasing the complexity of the protocol on mis-predictions. Hossain *et al.* propose different optimizations for each sharing pattern considering a chip multiprocessor architecture [50]. Particularly, they accelerate the producer-consumer pattern by converting 3-hop read misses into 2-hop read misses. Again, communication between the cache providing the data block and the directory is necessary, thus introducing more complexity in the protocol. In contrast, direct coherence is applicable to all types of misses (reads, writes and upgrades) and just the identity of the owner tile is predicted. Moreover, the fact that the directory information is stored along with the owner of the block simplifies the protocol. Finally, differently from the techniques proposed by Acacio *et al.*, we avoid predicting the current holders of a block by storing the up-to-date directory information in the owner tile.

Also in the context of shared-memory multiprocessors, Cheng *et al.* [31] have proposed converting 3-hop read misses into 2-hop read misses for memory blocks that exhibit the producer-consumer sharing pattern by using extra hardware to detect when a block is being accessed according to this pattern. Now the directory is delegated to the tile that updates the block in a producer-consumer sharing pattern in order to reduce the complexity of the cache coherence protocol. Differently from this proposal, direct coherence protocols obtain 2-hop misses for read, write and upgrade misses without taking into account sharing patterns.

Enright *et al.* propose Virtual Tree Coherence (VTC) [54]. This mechanism uses coarse-grain coherence tracking [23] and the sharers of a memory region are connected by means of a virtual tree. Since the root of the virtual tree serves as

the ordering point in place of the home tile, and the root tile is one of the sharers of the region, the indirection can be avoided for some misses. In contrast, direct coherence protocols keep the coherence information at block granularity and the ordering point always has the valid copy of the block, which leads to less network traffic and lower levels of indirection.

Huh *et al.* propose to allow replication in a NUCA cache to reduce the access time to a shared multibanked cache [52]. In the same vein, Zhang *et al.* propose victim replication [123], a technique that allows some blocks evicted from an L1 cache to be stored in the local L2 bank. In this way, the next cache miss for this block will find it at the local tile, thus reducing miss latency. More recently, Beckmann *et al.* [17] present ASR (Adaptive Selective Replication) that replicates cache blocks only when it is estimated that the benefits of replication (lower L2 hit latency) exceeds its costs (more L2 misses). In contrast, our protocol reduces miss latencies by avoiding the access to the L2 cache when it is not necessary, and no replication at the L2 cache is performed. These techniques could also be implemented along with direct coherence protocols.

Chang and Sohi propose cooperative caching [27], a set of techniques that reduce the number of off-chip accesses in a CMP with a private cache organization for the last-level caches (the L2 caches in this case). Differently from previous works, they assume a private organization, in which blocks are inherently replicated in the L2 caches allowing fast L2 accesses, and they try to remove copies of replicated blocks in order to improve the L2 cache hit rate. Again, these techniques can be implemented along with direct coherence protocols, since they can be used for shared and private organizations.

Martin *et al.* present a technique that allows snooping-based protocols to utilize unordered networks by adding logical timing to coherence requests and reordering them on destiny to establish a total order [76]. Likewise, Agarwal *et al.* propose In-Network Snoop Ordering (INSO) [9] to allow snooping over unordered networks. Since direct coherence protocols do not rely on broadcasting requests, they generate less traffic and, therefore, less power consumption when compared to snooping-based protocols.

Martin *et al.* propose to use destination-set prediction to reduce the bandwidth required by a snoopy protocol [74]. Differently from *DiCo-CMP*, this proposal is based on a totally-ordered interconnect (a crossbar switch), which does not scale with the number of cores. Destination-set prediction is also used by *Token-M* in shared-memory multiprocessors with unordered networks [73]. However, on mis-predictions, requests are solved by resorting on broadcasting

after a time-out period. Differently, in *DiCo-CMP* mis-predictions are re-sent immediately to the owner cache, thus reducing latency and network traffic.

Some authors evaluated the use of hints with different objectives [20, 51]. In these works the authors try to keep updated directory information to find out where a fresh copy of the block can be obtained in case of a read miss. In contrast, we use the hints as a policy to update the location of the owner cache which servers as ordering point and stores up-to-date directory information. The use of signatures has been recently proposed for disambiguating addresses across threads in transactional memory [25]. In contrast, we use signatures to keep information that improves the efficiency of the hints mechanism.

Cheng *et al.* [32] adapt already existing coherence protocols for reducing energy consumption and execution time in CMPs with heterogeneous networks. In particular, they assume a heterogeneous network comprised of several sets of wires, each one with different latency, bandwidth, and energy characteristics, and propose to send each coherence message through a particular set of wires depending on its latency and bandwidth requirements. Our proposal is orthogonal to this work and the ideas presented by Cheng could also be applied to *DiCo-CMP*.

Finally, the idea of storing the directory information along with the tag field in the private cache was previously proposed by us in the lightweight directory architecture [95, 102] in the context of cc-NUMA systems. Since the improvements obtained by this proposal are not significant for CMPs with a shared cache organization, we do not evaluate it in this thesis. However, next section summarizes the concepts behind the lightweight directory architecture.

5.2.1 The lightweight directory architecture

Traditional cc-NUMA architectures store the directory information in main memory. Since each cache miss requires the access to the directory information in a directory protocol, this access incurs in long L2 miss latencies [4]. Directory caches help to avoid memory accesses when the only valid copy of the requested block holds in a private cache. If the requested block is shared by several cores, the valid copy of the block is obtained from main memory. MOESI protocols can avoid these expensive accesses but at the cost of increasing the number of misses with indirection [96].

On the other hand, the amount of extra memory required for storing directory information (directory memory overhead) could become prohibitive for a large-scale configuration of the multiprocessor if care is not taken [5]. Even

when a directory cache is employed, a backup directory stored in main memory is required to prevent unnecessary invalidations.

The lightweight directory architecture cope with the long cache miss latencies and the directory memory overhead. Unlike conventional directories, which associate directory entries to memory blocks, this proposal moves directory information to the cache level where the coherence of the memory block is managed (the L2 cache in the cc-NUMA system that we consider). In this way, directory information is removed from main memory, and it is only necessary to add a sharing core and a state field to the private L2 caches. Since we assume that L1 and L2 caches are inclusive, the L1 caches do not have to be modified.

As in a conventional directory protocol, L2 cache misses are sent to the corresponding home node which is in charge of satisfying the miss (for example, by providing the memory block in case of a load miss). However, on the first reference to a memory block in the lightweight directory architecture, the home node books an entry in the local L2 cache which is used to store directory information for the block and occasionally the own block. Subsequent L2 cache misses to the same block will find directory information and in some cases data in the L2 cache of the home node. Note that when both the directory information and the data block are found in the home L2 cache, read misses are solved in just two hops (without entailing indirection), instead of accessing main memory.

This proposal is motivated by the observation that only a small fraction of the memory blocks are stored in the L2 caches at a particular time (temporal locality), and that in most cases, when a request for a memory block from a remote node arrives at the corresponding home node either the home node has recently accessed the block and it resides in its L2 cache, or the home node will request the block in a near future.

However, storing directory information in the L2 cache for each block requested by any remote node could result in a significant increase in the number of blocks being stored in the L2 cache of the corresponding home directory and, consequently, in its total number of replacements. Fortunately, the observation that motivates our proposal points out that it is not the common case, and the improvements obtained for avoiding the memory accesses compensate the small increase in L2 cache replacements [95].

This proposal, therefore, brings two important benefits. First of all, since the total number of memory blocks is much larger than the total number of L2 cache entries, directory memory overhead is drastically reduced by a ratio of 1024 (or more) compared to conventional directory architectures. Second, since directory entries are stored in the L2 cache of the home node, the time needed

to access the directory is significantly reduced, which translates into important reductions in the latency of L2 cache misses.

Next section provides more detail about the implementation of the lightweight directory architecture for cc-NUMA machines. This architecture constitutes a simple cache design that only adds two fields to the tags' portion of the L2 cache for storing directory information. In this way, this design does not need extra hardware structures (in contrast with the inclusion of directory caches) to avoid the accesses to main memory when only directory information is needed. On the other hand, this design also ensures that an up-to-date copy of data will always be in the cache of the home node for those blocks in shared state, avoiding thus the long access to main memory to get the block in these cases. Its main drawback is, however, that the total number of replacements could increase for applications with low temporal locality in the accesses to memory that several nodes are performing, but fortunately this is not the common case.

5.2.1.1 Cache Design

Figure 5.2 shows the cache design assumed in the lightweight directory architecture. The cache is split into tags and data structures, as is commonly found in current designs. The access to both structures is performed in parallel. Each cache block contains four main fields in the tags' portion: the *tag* itself, used to identify the block, the cache state (*L2*), the directory state (*Dir*), and the *sharing code*. The latter two fields are added by the lightweight directory architecture. If the cache state is invalid the node does not keep an up-to-data copy of the cache block. However, if any of the presence bits in the sharing code is set the entry has valid directory information. The directory state field can take two values (one bit):

- S (Shared): The memory block is shared by several cores, each one of them with an up-to-date copy. When needed, the cache of the home node will provide the block to the requester, since this cache has always a valid copy even when the local processor has not referenced the block.

- P (Private): The memory block is in just one private cache and could have been modified. The single valid copy of the block is held in the cache of the home node when its cache state is modified or exclusive, or alternatively, in one of the caches of the remote nodes. In the latter case, the cache state

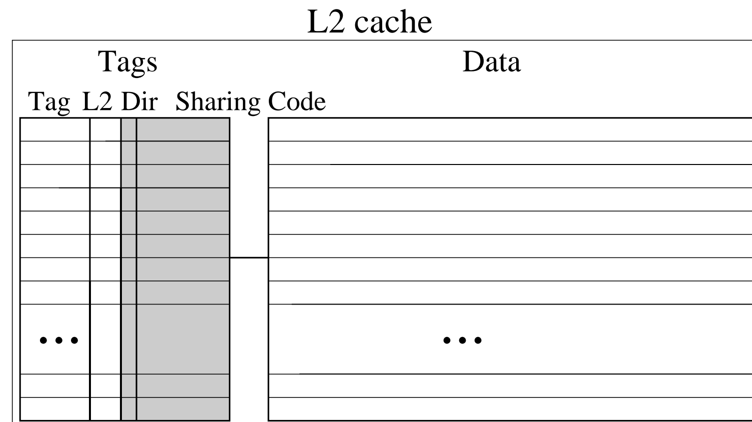


Figure 5.2: Cache design for the lightweight directory architecture. The grey zone represents the overhead in cache memory introduced by the directory.

for the memory block in the home node is invalid, and the identity of the owner cache is stored in the sharing code field.

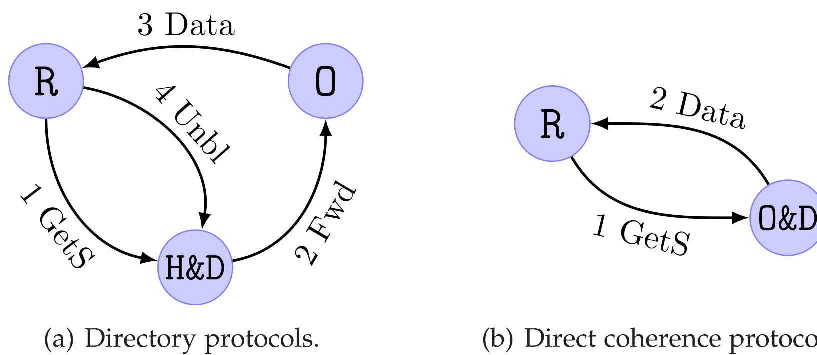
Note that an additional directory state is implicit. The *U* state (Uncached) takes place when the memory block is not held by any cache and its only copy resides in main memory. This is the case of those memory blocks that have not been accessed by any node yet, or those that were evicted from all the caches.

5.3 Direct coherence protocols

In this section, we describe direct coherence protocols and its implementation for tiled CMPs in detail. First, we explain how direct coherence avoids indirection for most cache misses by changing the distribution of the roles involved in cache coherence maintenance. We also study the changes in the structure of the tiles necessary to implement *DiCo-CMP*. Then, we describe the cache coherence protocol for tiled CMPs and, finally, we study how to avoid the starvation issues that could arise in direct coherence protocols.

5.3.1 Direct coherence basis

As already discussed, directory protocols introduce indirection in the critical path of cache misses. Figure 5.3(a) shows a cache miss suffering indirection in



(a) Directory protocols.

(b) Direct coherence protocols.

Figure 5.3: How cache-to-cache transfer misses are solved in directory and direct coherence protocols. R=Requester; H=Home; D=Directory; O=Owner.

a directory protocol, a cache-to-cache transfer for a read miss. When a cache miss takes place it is necessary to access the home tile to obtain the directory information and serialize the requests before performing any coherence action (*1 GetS*). In case of a cache-to-cache transfer miss, the request is subsequently forwarded to the owner tile (*2 Fwd*), where the block is provided (*3 Data*). As it can be observed, the miss is solved in three hops. Moreover, requests for the same block cannot be processed by the directory until it receives the unblock message (*4 Unbl*).

To avoid this indirection problem, we propose to directly send the request to the provider of the block, i.e., the owner tile. This is the main motivation behind direct coherence. To allow the owner tile to process the request, direct coherence stores the sharing information along with the owner block, and it also assigns the task of keeping cache coherence and ensuring ordered accesses for every memory block to the tile that stores that block. As shown in Figure 5.3(b), *DiCo-CMP* sends the request directly to the owner tile (*1 GetS*), instead of to the home tile. In this way, data can be provided by the owner tile (*2 Data*), just requiring two hops to solve the cache miss. As we can see, the unblock message is not necessary in direct coherence protocols as discussed later at the end of this section.

Therefore, direct coherence requires a re-distribution of the roles involved in solving a cache miss. Next, we describe the tasks performed in cache coherence protocols and the component responsible for each task in both directory and direct coherence protocols, which are illustrated in Figure 5.4:

- *Order requests*: Cache coherence maintenance requires to serialize the re-

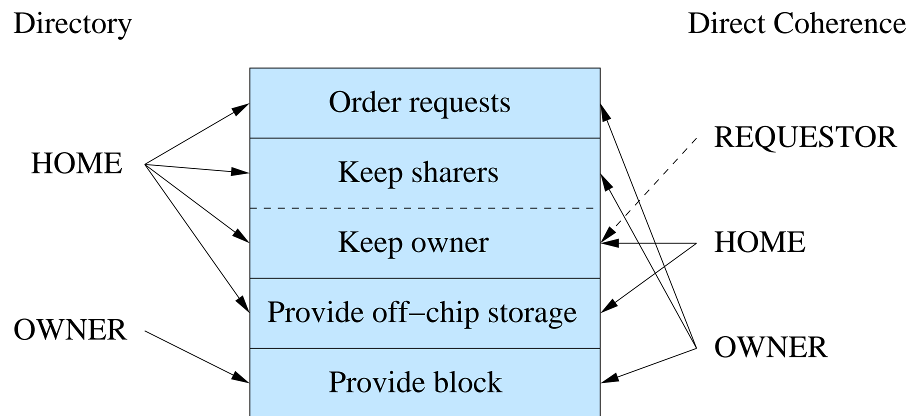
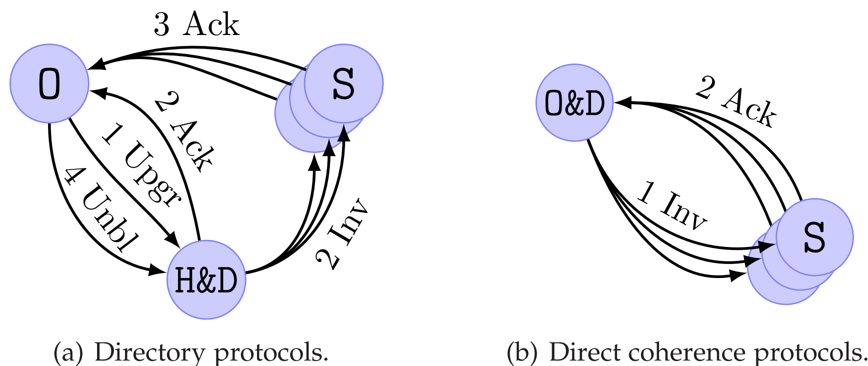


Figure 5.4: Tasks performed in cache coherence protocols.

quests issued by different cores for the same block. In snooping-based cache coherence protocols, the requests are ordered by the shared interconnection network (usually, a bus or a crossbar). However, since tiled CMP architectures implement an unordered network, this serialization of the requests must be carried out by another component. Directory protocols assign this task to the home tile of each memory block. On the other hand, this task is performed by the owner tile in direct coherence protocols.

- *Keep coherence information*: Coherence information is used to track blocks stored in private caches. In protocols that include the *O* state, like MOESI protocols, coherence information also identifies the owner tile. In particular, *sharing information* is used to invalidate all cached blocks on write misses, while *owner information* is used to know the identity of the provider of the block on every miss. Directory protocols store coherence information at the home tile, where cache coherence is maintained. Instead, direct coherence requires that sharing information be stored in the owner tile for keeping coherence there, while owner information is stored in two different components. First, the requesting cores need to know the tile cache to send the requests to it. Processors can easily keep the identity of the owner tile, e.g., by recording the last core that invalidated their copy. However, this information can become stale and, therefore, it is only used for avoiding indirection (dashed arrow in Figure 5.4). Then, the responsible for tracking the up-to-date identity of the tile cache is the home tile which must be notified on every ownership change.



(a) Directory protocols.

(b) Direct coherence protocols.

Figure 5.5: How upgrades are solved in directory and direct coherence protocols. O=Owner; H=Home; D=Directory; S=Sharers.

- *Provide the data block*: If the valid copy of the block resides on chip, data is always provided by the owner tile, since it always holds a valid copy. In a MOESI protocol, the owner of a block is either a cache holding the block in the exclusive or the modified state, the last cache that wrote the block when there are multiple sharers, or the L2 cache slice inside the home tile in case of the eviction of the owner block from some L1 cache. On the other hand, in MESI protocols data is always provided by the home tile when it is shared by several cores (state *S*).

- *Provide off-chip storage*: When the valid copy of a requested block is not stored on chip, an off-chip access is required to obtain the block. Both in directory and direct coherence protocols the home tile is responsible for detecting that the owner copy of the block is not stored on chip. It is also responsible for sending the off-chip request and receiving the data block.

Another example of the advantages of direct coherence is shown in Figure 5.5. This diagram represents an upgrade that takes place in a tile whose L1 cache holds the block in the owned state, which happens frequently in common applications (e.g., for the producer-consumer pattern). In a directory protocol, upgrades are solved by sending the request to the home tile (*1 Upgr*), which replies with the number of acknowledgements that must be received before the block can be modified[1] (*2 Ack*), and sends invalidation messages to all sharers

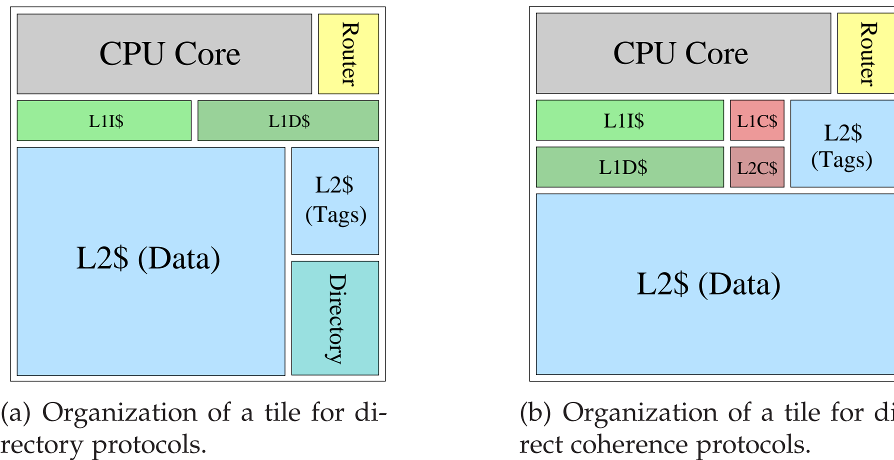
[1]As described in Section 2.4, the directory protocol that we assume sends the number of acknowledgement messages that are expected to be received by the requester along with the forwarded and data messages. Obviously, this happens for all the cases except when the data block is not requested since it is already stored in the L1 cache of the requesting core.

(*2 Inv*). Sharers confirm their invalidation to the requester (*3 Ack*). Once all the acknowledgements have been received by the requester, the block can be modified and the directory is unblocked (*4 Unbl*). In contrast, in *DiCo-CMP* only invalidation messages (*1 Inv*) and acknowledgements (*2 Ack*) are required because the directory information is stored along with the data block, thereby solving the miss with just two hops in the critical path.

Additionally, by keeping together the owner block and the directory information, the control messages between them are not necessary, thus saving some network traffic (two messages in Figure 5.3 and three in Figure 5.5). As previously commented, direct coherence does not need the unblock message required by directory protocols to serialize the requests. In directory protocols the unblock message can be sent either by the owner or by the requesting tile (see Section 2.5). Although in directory protocols this decision does not affect performance significantly, in direct coherence protocols it is preferable to let the owner tile send the unblock message. Since the owner tile and the ordering tile are the same, this message is not necessary, thus saving coherence traffic. Moreover, it also reduces the waiting time for the subsequent requests and, consequently, the average miss latency. Note that, in direct coherence protocols, sending the unblock message from the requester makes the owner wait for two hops (data and unblock messages), while sending the message from the owner, i.e., not sending any message, results in no waiting time at the owner tile. Finally, this also allows the O&D tile to solve misses without using transient states, thus reducing the number of states and making the implementation simpler than a directory protocol.

5.3.2 Changes to the structure of the tiles of a CMP

The new distribution of roles that characterizes direct coherence protocols requires some modifications in the structure of the tiles that build the CMP. Firstly, the identity of the sharers for every block is stored in the corresponding owner tile instead of the home one to allow caches to keep coherence for the memory blocks that they hold in the owned state. Therefore, *DiCo-CMP* extends the tags' part of the L1 caches with a sharing code field, e.g., a full-map (L2 caches already include this field in directory protocols). In this way, the design of the caches in *DiCo-CMP* is similar to that previously described for the lightweight directory architecture (see Figure 5.2). In contrast, *DiCo-CMP* does not need to store a directory structure at the home tile, as happens in directory protocols.



(a) Organization of a tile for directory protocols.

(b) Organization of a tile for direct coherence protocols.

Figure 5.6: Modifications to the structure of a tile required by direct coherence protocols.

Additionally, *DiCo-CMP* adds two extra hardware structures that are used to record the identity of the owner tile of the memory blocks stored on chip:

- *L1 coherence cache* (L1C$): The pointers stored in this structure are used by the requesting core to avoid indirection by directly sending local requests to the corresponding owner tile. Therefore, this structure is located close to each processor's core. *DiCo-CMP* can update this information in several ways based on network usage, as discussed later in Section 5.4.

- *L2 coherence cache* (L2C$): Since the owner tile can change on write misses, this structure must track the owner tile for each block allocated in any L1 cache. This structure replaces the directory structure required by directory protocols and it is accessed each time a request fails to locate the owner tile. This information must be updated whenever the owner tile changes through control messages. These messages must be processed by the L2C$ in the very same order in which they were generated in order to avoid any incoherence when storing the identity of the owner tile, as described later in Section 5.3.3.3.

Figure 5.6 shows a tile design for directory protocols and for direct coherence protocols. A comparison among the extra storage and structures required by the protocols evaluated in this chapter can be found in Section 5.5.

In our particular implementation, the pointers used to avoid indirection are only stored in the L1C$. On the other hand, the directory information stored in the tags' part of the L1 cache is only valid for blocks in the owned state. In order to make better use of the cache storage, the pointers could be also stored in the tags' part of the L1 caches for blocks in the shared or the invalid state. However, for the sake of clarity, the protocol implemented in this chapter does not consider this improvement.

5.3.3 Description of the cache coherence protocol

5.3.3.1 Requesting processor

When a processor issues a request that misses in its private L1 cache, it sends the request directly to the owner tile in order to avoid indirection. The identity of the potential owner tile is obtained from the L1C$, which is accessed at the time that the cache miss in detected. If there is a hit in the L1C$, the request is sent to the obtained owner tile. Otherwise, the request is sent to the home tile, where the L2C$ will be accessed to get the identity of the current owner tile.

5.3.3.2 Request received by a tile that is not the owner

When a request is received by a tile that is not the current owner of the block, it simply re-sends the request. If the tile is not the home one, the request is re-sent to it. Otherwise, if the request is received by the home tile and there is a hit in the L2C$, the request is sent to the current owner tile. In absence of race conditions the request will reach the owner tile. Finally, if there is a miss in the L2C$ and the home tile is not the owner of the block, the request is solved by providing the block from main memory, where, in this case, a fresh copy of the block resides. This is because the L2C$ always keeps an entry for the blocks stored in the private L1 caches. If the owner copy of the block is not present in either any L1 cache or in the L2 cache, it resides off-chip. After the off-chip access, the block is allocated in the requesting L1 cache, which gets the ownership of the block, but not in the L2 cache (as occurs in the directory protocol)[2]. In addition, it is necessary to allocate a new entry in the L2C$ pointing to the current L1 owner tile.

[2]As mentioned in Section 3.2, we assume that the L1 and the L2 cache are non-inclusive. However, direct coherence protocols are equally applicable with other configurations for the L1 and L2 caches, obtaining similar results to those presented in this chapter.

5.3.3.3 Request received by the owner tile

Every time a request reaches the owner tile, it is necessary to check whether this tile is currently processing a request from a different processor for the same block (a previous write waiting for acknowledgements). In this case, the block is in a busy or transient state, and the request must wait until all the acknowledgements are received.

If the block is not in a transient state, the miss can be immediately solved. If the owner is the L2 cache at the home tile all requests (reads and writes) are solved by deallocating the block from the L2 cache and allocating it in the private L1 cache of the requester. Again, the identity of the new owner tile must be stored in the L2C$.

When the owner is an L1 cache, read misses are completed by sending a copy of the block to the requester and adding it to the sharing code field kept along with the block. Since our protocol is also optimized for the migratory-sharing pattern, as all the protocols implemented in this thesis, read misses for migratory blocks invalidate the copy in the owner tile and send the exclusive data to the requesting processor.

For write misses, the owner tile sends invalidation messages to all the tiles that hold a copy of the block in their L1 caches and, then, it sends the data block to the requester. Acknowledgement messages are collected at the requesting core. If the miss is an upgrade, it is necessary to check the sharing code field kept along with the data block to know whether the requester still holds a copy of the block (note that a previous write miss from a different processor could have invalidated its copy and, in this case, the owner tile should also provide a fresh copy of the block). As previously shown in Figure 5.5, upgrade misses that take place in the owner tile just need to send invalidations and receive acknowledgements (two hops in the critical path).

Finally, since the L2C$ must store up-to-date information regarding the owner tile, every time that this tile changes, the old owner tile also sends a control message to the L2C$ indicating the identity of the new owner tile. These messages must be processed by the L2C$ in the very same order in which they were generated. Otherwise, the L2C$ could fail to store the identity of the current owner tile. Fortunately, there are several approaches to ensure this order. In the implementation evaluated in this chapter, once the L2C$ processes the message reporting an ownership change from the old owner tile, it sends a confirmation response to the new one. Until this confirmation message is received by the new owner tile, it could access the data block (if already received), but

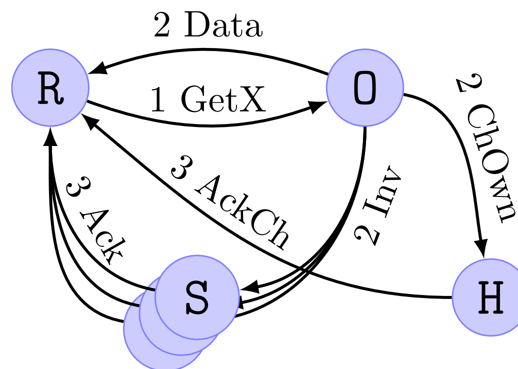


Figure 5.7: Example of ownership change upon write misses. R=Requester; O=Owner; S=Sharers; H=Home.

cannot give the ownership to another tile (the miss status hold register –MSHR– allocated on the cache miss is still held). Since these two control messages are not in the critical path of the cache miss, they do not introduce extra latency.

As an example, Figure 5.7 illustrates a write miss for a shared block. It assumes that the requester has valid and correct information about the identity of current owner tile in the L1C$ and, therefore, it directly sends the request to the owner tile (*1 GetX*). Then the owner tile must perform the following tasks. First, it sends the data block to the requester (*2 Data*). Second, it sends invalidation messages to all the sharers (*2 Inv*), and it also invalidates its own copy. The information about the sharers is obtained from the sharing code stored along with every owner block. Third, it sends the message informing about the ownership change to the home tile (*2 ChOwn*). All tiles that receive an invalidation message respond with an acknowledgement message to the requester once they have invalidated their local copies (*3 Ack*). When the data and all the acknowledgements arrive to the requesting processor the write operation can be performed. However, if another write request arrives to the tile that previously suffered the miss, it cannot be solved until the acknowledgement to the ownership change issued by the home tile (*3 AckCh*) is received.

5.3.3.4 Replacements

In our particular implementation, when a block with the ownership property is evicted from an L1 cache, it must be allocated at the L2 cache along with the up-to-date directory information. Differently from directory and *Hammer*

protocols and similarly to *Token*, replacements are performed by sending the writeback message directly to the home tile (instead of requiring three-phase replacements). This operation can be easily performed in direct coherence protocols because the tile where these blocks are stored is the responsible for keeping cache coherence and, as consequence, no complex race conditions can appear. When the writeback message reaches the home tile, the L2C$ deallocates its entry for this block because the owner tile is now the home one. On the other hand, replacements for blocks in shared state are performed transparently, i.e., no coherence actions are needed.

Finally, no coherence actions must be performed in case of an L1C$ replacement. However, when an L2C$ entry is evicted, the protocol should ask the owner tile to invalidate all the copies from the private L1 caches. Luckily, as happens to the directory cache in directory protocols, an L2C$ with the same number of entries and associativity than the L1 cache is enough to completely remove this kind of replacements, as previously explained in Chapter 4.

5.3.4 Preventing starvation

Directory protocols avoid starvation by enqueuing requests in FIFO order at the directory buffers. Differently in *DiCo-CMP*, write misses can change the tile that keeps coherence for a particular block and, therefore, some requests can take some extra time until this tile is finally found. If a memory block is repeatedly written by several processors, a request could take some time to find the owner tile ready to process it, even when it is sent by the home tile. Hence, some processors could be solving their requests while other requests are starved. Figure 5.8 shows an example of a scenario in which starvation appears. R_1 and R_2 tiles are issuing write requests repeatedly and, therefore, the owner tile is continuously moving from R_1 to R_2 and vice versa. On every change of owner the home tile is notified, and the requesting core is acknowledged. However, at the same time, the home tile is trying to re-send the request issued by R_3 tile to the owner one, but the request is always returned to the home tile because the write request issued by R_1 or R_2 arrives before to the owner tile.

DiCo-CMP detects and avoids starvation by using a simple mechanism. In particular, each time that a request must be re-sent to the L2C$ in the home tile, a counter into the request message is increased. The request is considered starved when this counter reaches a certain value (e.g, three accesses to the L2C$ for the evaluation carried out in this thesis). When the L2C$ detects a starved request, it re-sends the request to the owner tile, but it records the address of

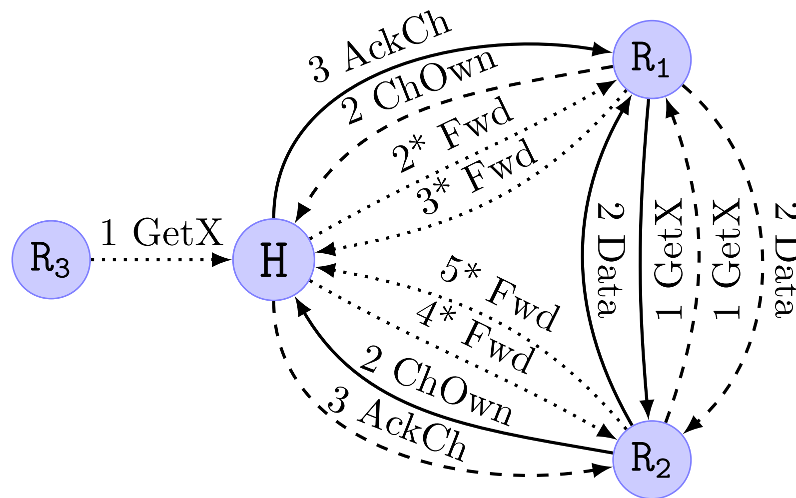


Figure 5.8: Example of a starvation scenario in direct coherence protocols. R_x=Requester; H=Home. Continuous arrows represent cache misses that take place in R_1, dashed arrows represent misses in R_2 and dotted arrows represent misses in R_3.

the block. If the starved request reaches the current owner tile, the miss is solved, and the home tile is notified, ending the starvation situation. If the starved request does not reach the owner tile is because the ownership property is moving from a tile to another one. In this case, when the message informing about the change of the ownership arrives to the home tile, it detects that the block is suffering from starvation, and the acknowledgement message required on every ownership change is not sent. This ensures that the owner tile does not change until the starved request can complete.

5.4 Updating the L1 coherence cache

DiCo-CMP uses the L1C$ to avoid indirection by keeping pointers that identify the owner tile of certain blocks. Several policies can be used to update the value of these pointers. A first approach consists in recording the information about the last core that invalidated or provided each block, i.e., the last core that wrote the block. When a block is invalidated from an L1 cache, the L1C$ records the identity of the processor causing the invalidation. In case of a read miss, the identity of the provider of the block is also stored. Additionally, when an owner

block is evicted from an L1 cache, some control messages are sent to the sharers to inform about the new location of the owner tile, the home tile. We call this policy the *Base* policy.

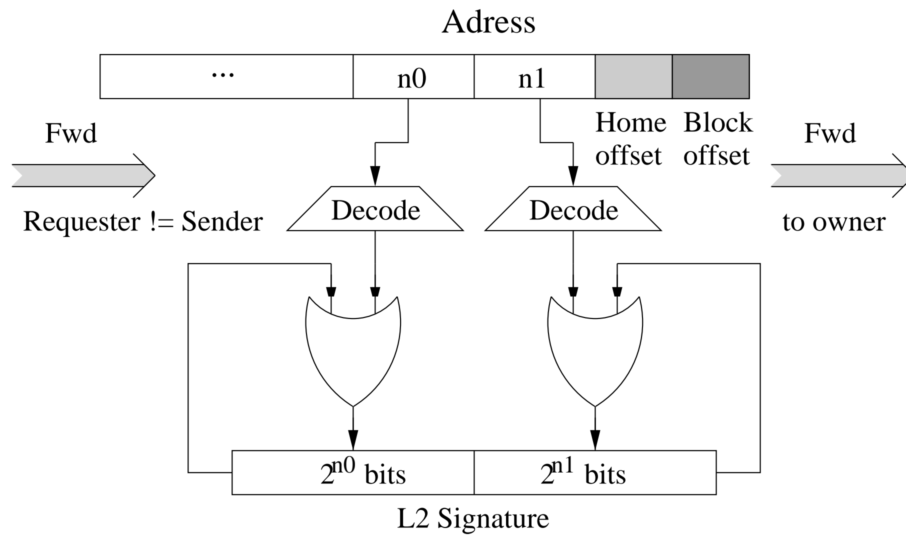
Unfortunately, in most cases this information is not enough to obtain accurate owner predictions and it must be enhanced by sending some *hints*. Hints are control messages that inform the L1C$ about ownership changes. Since sending hints to all cores on each change is not efficient in terms of network traffic, it is necessary to keep track of those cores that need to receive hints for each memory block.

The first alternative proposed for sending hints is the *frequent sharers* mechanism. This mechanism requires the addition of a new field to each cache entry. This field keeps a bit-vector that identifies the requesting cores for each owner block. When there is a cache-to-cache transfer of an owner block, hints are sent to the frequent sharers of that block to update the L1C$s. Moreover, the frequent sharers vector is also sent along with the data message. Since we choose not to store the frequent sharer information at the L2 cache level in order to keep storage requirements low, this field is reset whenever there is an L1 cache eviction of an owner block. We call this policy *Hints FS*.

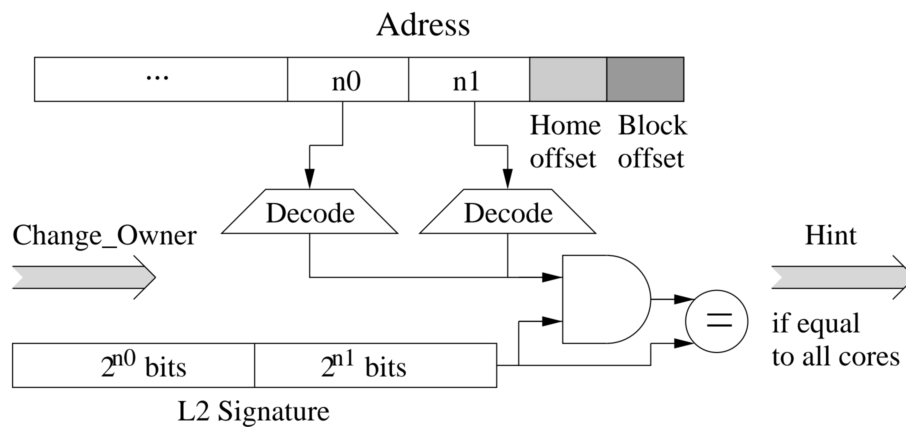
The *frequent sharers* mechanism is not very suitable for large-scale tiled CMPs since the area required by the bit-vector does not scale with the number of cores. Additionally, this mechanism does not filter hint messages for those blocks in which the *Base* mechanism works well, thus consuming precious network bandwidth and, therefore, energy.

Another more elaborate alternative that achieves better scalability in terms of area requirements consists in using *address signatures*. We call this policy *Hints AS*. Address signatures have been recently employed in transactional memory systems for disambiguating address across threads [25, 121], and to allow to encode a set of addresses into a register of fixed size, following the principles of hash-encoding with allowable errors as described in [21]. The disadvantage of address signatures is that false positives can happen. However, in the hints mechanism this is not a correctness issue but maybe a performance issue. Particularly, it increases network traffic.

As shown in Figure 5.9, each home tile includes an address signature (*L2 Signature*) that encodes a certain set of addresses. In order to filter some useless hints we only store the addresses for those cache misses mis-predicting the owner tile, i.e., the home tile receives a request from a core that is not the requester one, as shown in Figure 5.9(a). In this way, when the home tile is informed about the ownership change for a particular block, it checks the sig-



(a) Codifying the memory blocks that need hints into the signature.



(b) Checking if a memory block needs hints.

Figure 5.9: Organization of the address signature mechanism proposed to send hints.

nature to broadcast hints if the address is present, as illustrated in Figure 5.9(b). Note that when invalidation messages are required it is not necessary to send hints to the cores that receive them.

Since this scheme only uses one signature for all cores, and hints are broadcast to them, some cores will receive hints for blocks that they are not actually requesting, thus overloading the L1C$. To avoid this effect, we add another address signature (L1 Signature) to each core. On each cache miss, the address of the block is stored in the signature. Then, when a hint is received, it is only stored in the L1C$ if the address is found in the signature.

Particularly, addresses are encoded using a double-bit-select signature implementation [121], as Figure 5.9 shows. The signature is divided into two sets. The $log_2(b) - 1$ less significant bits are decoded and ORed with the first set, being b the size of the signature. The $log_2(b) - 1$ subsequent less significant bits are decoded and ORed with the second set. An address belongs to the signature if the corresponding bit is present in both sets.

When we refer to the less significant bits we do not take into consideration the block offset. For the L2 signature, we neither take the home offset, as illustrated in Figure 5.9. This offset comes from assigning an address to a home tile according to the less significant bits ($log_2 n$).

5.5 Area and power considerations

In this section we compare the memory overhead and the extra structures needed by the three protocols evaluated in this chapter: *Token*, *Directory* and *DiCo-CMP*. Moreover, we discuss how frequently these structures are accessed to demonstrate that direct coherence protocols will not have significant impact on the power consumed by these structures and, therefore, significant reductions on total power consumption can be expected as a result of the important savings in terms of network traffic that *DiCo-CMP* entails (see Section 5.6.3).

For the particular configuration employed along this thesis (a 4×4 tiled CMP with 128KB L1 private caches), the number of bits required for storing a full-map sharing code is 16 (2 bytes), whereas just $log_2 16 = 4$ bits are needed for storing a single pointer. Table 5.1 details the structures, the size, and the memory overhead respect to the size of the data caches required by *Token*, *Directory* and the different implementations of *DiCo-CMP* evaluated in this chapter. For both *DiCo-Hints FS* and *DiCo-Hints AS* the structures and overhead shown are added to the *DiCo-Base* policy. Likewise, table 5.2 shows the area overhead in

terms of mm^2, which has been calculated using the CACTI tool with the parameters specified in Chapter 3. Now, we present all the structures needed by each policy for *DiCo-CMP*. For these structures we do not show the number of entries because is the same as in Table 5.1. Note that these tables concentrate on the structures used for keeping coherence information and, therefore, does not account for the extra structures required by *Token* and *DiCo-CMP* to avoid starvation. Note that the large tables required by *Token* for keeping active persistent requests can become an issue for large-scale CMPs due to their lack of scalability in terms of area requirements [37].

Table 5.1: Memory overhead introduced by coherence information (per tile) in a 4x4 tiled CMP.

	Structure	**Entry size**	**Entries**	**Total size**	**Overhead**
Data	L1 cache	tag + 64 bytes	2K	134.25KB	
	L2 cache	tag + 64 bytes	16K	1070KB	
Token	L1$ tokens	5 bits	2K	1.25KB	**0.93%**
	L2$ tokens	5 bits	16K	10KB	
Directory	L2$ dir. inf.	2 bytes	16K	32KB	**3.59%**
	Dir. cache	tag + 2 bytes	2K	11.25KB	
DiCo-Base	L1$ dir. inf.	2 bytes	2K	4KB	**4.19%**
	L2$ dir. inf.	2 bytes	16K	32KB	
	L1C$	tag + 4 bits	2K	7.25KB	
	L2C$	tag + 4 bits	2K	7.25KB	
DiCo-Hints FS	L1$ freq. sh.	2 bytes	2K	4KB	**+0.34%**
DiCo-Hints AS	L1 signature	128 bytes	1	0.125KB	**+0.02%**
	L2 signature	128 bytes	1	0.125KB	

Token needs to keep the token count for any block stored both in the L1 and L2 caches. This information only requires $\lceil log_2(n+1) \rceil$ bits (the owner-token bit and the non-owner token count), where n is the number of processing cores. These additional bits are stored in the tags' part of both cache levels. The memory overhead of this protocol is 0.93% in terms of bits and 4.65% in terms of mm^2 for a 16-tile CMP. As already commented, although *Token* needs extra hardware to implement both the persistent requests and the timeout mechanisms we do not consider this extra hardware in the overhead analysis.

Table 5.2: Area overhead introduced by coherence information (per tile) in a 4x4 tiled CMP.

	Structure	Entry size	Total area	Overhead
Data	L1 cache	tag + 64 bytes	1.15 mm^2	
	L2 cache	tag + 64 bytes	6.25 mm^2	
Token	L1 cache	tag + 5 bits + 64 bytes	1.19 mm^2	**4.65%**
	L2 cache	tag + 5 bits + 64 bytes	6.55 mm^2	
Directory	L1 cache	tag + 64 bytes	1.15 mm^2	**16.12%**
	L2 cache	tag + 2 bytes + 64 bytes	7.36 mm^2	
	Dir. cache	tag + 2 bytes	0.08 mm^2	
DiCo-Base	L1 cache	tag + 2 bytes + 64 bytes	1.28 mm^2	**18.11%**
	L2 cache	tag + 2 bytes + 64 bytes	7.36 mm^2	
	L1C$	tag + 4 bits	0.05 mm^2	
	L2C$	tag + 4 bits	0.05 mm^2	
DiCo-Hints FS	L1 cache	tag + 4 bytes + 64 bytes	1.43 mm^2	**20.04%**
	L2 cache	tag + 2 bytes + 64 bytes	7.36 mm^2	
	L1C$	tag + 4 bits	0.05 mm^2	
	L2C$	tag + 4 bits	0.05 mm^2	
DiCo-Hints AS	L1 cache	tag + 2 bytes + 64 bytes	1.28 mm^2	**18.14%**
	L2 cache	tag + 2 bytes + 64 bytes	7.36 mm^2	
	L1C$	tag + 4 bits	0.05 mm^2	
	L2C$	tag + 4 bits	0.05 mm^2	
	L1 signature	128 bytes	0.96 μm^2	
	L2 signature	128 bytes	0.96 μm^2	

Directory protocols store the on-chip directory information either in the L2 tags when the L2 cache holds a copy of the block or in a distributed directory cache when the block is stored in any of the L1 caches but not in the L2 cache. In our implementation, the number of entries of a directory bank is the same as the number of entries of an L1 cache, since this size is enough to always find the directory information for on-chip misses, i.e., without incurring in directory misses, as discussed in Chapter 4. The directory must be accessed on each cache miss. The memory overhead of this protocol is 3.59%, while its area overhead is 16.12%.

DiCo-CMP stores the directory information for blocks held in any L1 or L2 caches in the owner tile (L1 or L2). Moreover, it uses two structures that store a pointer to the owner tile, the L1 and L2 coherence caches. The L1C$ is accessed only when it is known that there is a cache miss in order to keep power consumption low. The L2C$ is necessary for locating the owner tile whenever the information in the L1C$ is not correct. This structure is only accessed for misses affected by indirection (about 22% of the cache misses as shown in Section 5.6.1). The number of entries of the L1C$ is the same as an L1 cache. On the other hand, as happens with the on-chip directory cache in the directory protocol, the L2C$ does not require more entries than the number of entries of the L1 caches. Differently from a directory cache, just one pointer is stored in each entry. In this way, the L2C$ required by *DiCo-CMP* has smaller size than the directory cache employed in the directory protocol. The L1C$ and the L2C$ used in the evaluation of this chapter have the same size. Therefore, the memory overhead of the *DiCo-Base* protocol is 4.19%, while its area overhead is 18.11%.

The use of hints improves performance at the cost of increasing both storage requirements and network traffic. In particular, the *frequent sharers* mechanism requires to store the set of the frequent sharers in the tags' part of the L1 cache –$O(n)$– and, consequently, its memory overhead is the highest (memory overhead of 4.53% and area overhead of 20.04%). On the other hand, the *address signature* mechanism only uses two signatures per tile (e.g, 1024 bits, each one) and, therefore, the storage required is reduced compared to the *frequent sharers* mechanism (memory overhead of 3.21% and area overhead of 18.14%) and it also scales with the number of cores. In general, we can see that direct coherence has an overhead close to a directory protocol.

In all the configurations a full-map is used as the sharing code for the directory information in order to concentrate on the impact that our proposal has on performance, avoiding any interference caused by the particularities of the sharing code as, for example, the presence of unnecessary coherence messages. However, next chapter studies the traffic-area trade-off of direct coherence protocols by employing compressed sharing codes.

5.6 Evaluation results and analysis

We have implemented all the policies described in Section 5.4 for direct coherence protocols: *Base*, *Hints FS* and *Hints AS*. In addition, to find out the potential of *DiCo-CMP*, we have implemented an *Oracle* policy in which the identity of

the current owner tile is immediately known by the requester on every cache miss. These implementations have been exhaustively checked using the tester program provided by GEMS that checks all race conditions to raise any incoherence.

We compare these implementations of *DiCo-CMP* with both the *Token* and the directory protocols described in Section 2.4. First, we study to what extent *DiCo-CMP* reduces indirection compared to a directory protocol. Then, we show that the reduction in the number of misses affected by indirection impacts in the cache miss latency. On the other hand, we also analyze the savings that *DiCo-CMP* obtains in terms of traffic in the on-chip interconnection network compared to *Token*. Then, we show the impact of these improvements on applications' execution time. Finally, we analyze the traffic-indirection trade-off obtained by all the different implementations of *DiCo-CMP* evaluated in this chapter.

5.6.1 Impact on the number of hops needed to solve cache misses

In general, *DiCo-CMP* reduces the average number of hops needed to solve a cache miss by avoiding the indirection introduced by the access to the home tile. However, in *DiCo-CMP*, some misses can increase the number of hops compared to a directory protocol due to owner mis-predictions. In order to study how *DiCo-CMP* impacts on the number of hops needed to solve cache misses, we classify each miss in one of the following categories:

- *2-hop misses*: Misses belonging to this category does not suffer from indirection since the number of hops in the critical path of the miss is two. In directory protocols, misses fall into this category either when the home tile of the requested block can provide the copy of the block or when the miss takes place in the home tile, and in both cases it is not necessary to invalidate blocks from other tiles. *Token* solves all misses that do not require persistent requests in two hops. Finally, *DiCo-CMP* solves cache misses using two hops either when the request is directly sent to the current owner tile and invalidations are not required or when the miss takes place in the tile where the owner block resides (upgrades).

  In all protocols, when the miss takes place in the home tile and this tile holds the owner block in the L2 cache, the miss is solved without generating network traffic (0-hop miss). These misses are also included in this category because they do not introduce indirection.

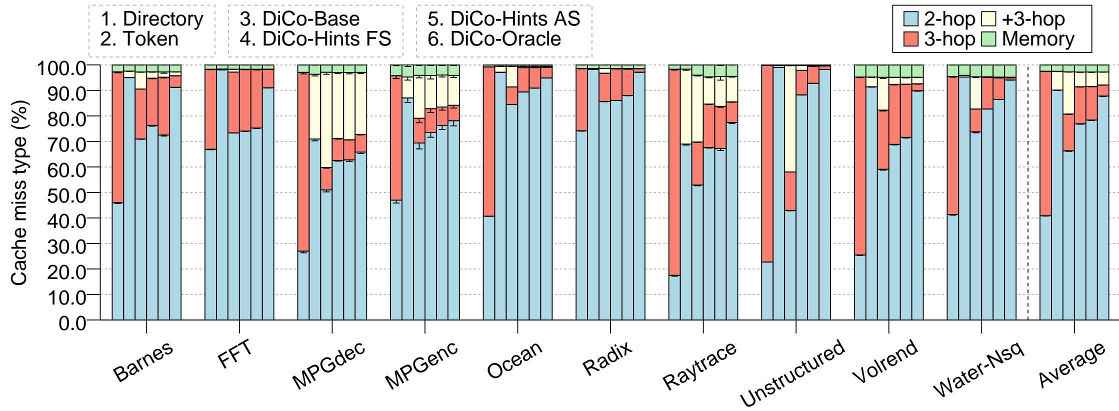


Figure 5.10: How each miss type is solved.

- *3-hop misses*: A miss belongs to this category when three hops in the critical path are necessary to solve it. This category represents the misses suffering from indirection in directory protocols. In contrast, 3-hop misses never take place in *Token*.

- *+3-hop misses*: We include in this category misses that need more than three hops in the critical path to be solved. This type of misses only happens in *DiCo-CMP*, when the identity of the owner tile is mis-predicted, or in *Token*, when persistent requests are required to solve the miss. The directory protocol evaluated in this thesis never requires more than three hops to solve cache misses since the acknowledgements to invalidation messages are collected by the requesting core, as described in Section 2.4.

- *Memory misses*: Misses that require off-chip accesses since the owner block is not stored on chip fall into this category.

Figure 5.10 shows the percentage of cache misses that fall into each category. As commented in the introduction of this chapter, in tiled CMP architectures that implement a directory protocol it is not very frequent that the requester be at the home tile of the block because the distribution of blocks among tiles is performed in a round-robin fashion. However, the fact that sometimes a coherent copy of the block is found in the L2 cache bank of the home tile, decreases the number of misses with indirection. In this way, the first bar in Figure 5.10 shows that most applications have an important fraction of misses suffering from indirection, like *Barnes*, *MPGdec*, *MPGenc*, *Ocean*, *Raytrace*, *Unstructured*, *Volrend*

and *Water-Nsq*, while other applications have most of the misses solved in two hops, like *FFT* and *Radix*. Obviously, *DiCo-CMP* will have more impact for the applications that suffer more indirection, although this impact will also depend on the cache miss rate of each application. We also can observe that *Token* solves most of the misses (90%) needing just two hops (see second bar).

As shown in the third bar of Figure 5.10, *DiCo-Base* increases the percentage of cache misses without indirection compared to a directory protocol (from 41% to 66% on average). On the other hand, *DiCo-Base* solves 17% of cache misses needing more than three hops. This fact is due to owner mis-predictions that can arise for two reasons: (1) staled owner information was found in the L1C$ or (2) the owner tile is changing or busy due to race conditions and the request is sent back to the home tile. The first case can be removed with a precise hints mechanism, as clearly happens in *Unstructured*. In the second case, the extra number of hops entailed by *DiCo-CMP* is equivalent to the cycles that the requests wait at the home tile until they are processed in the base directory protocol and, consequently, it does not suppose extra miss latency. This kind of *+3-hop* misses mainly appears in applications with high levels of contention, like *MPGdec*, *MPGenc* and *Raytrace*, and they also occur in *Token* as persistent requests.

The two hints mechanisms implemented for *DiCo-CMP*, *DiCo-Hints FS* and *DiCo-Hints AS* (fourth and fifth bars, respectively), increase the percentage of misses solved in two hops with respect to *DiCo-Base* in 11% and 12% on average, respectively. The main advantage of *DiCo-Hints AS* is its low storage overhead. Although for some applications the use of hints slightly increases the percentage of two-hop misses, for others, especially *Unstructured*, hints significantly help to achieve accurate predictions. Hints are mainly useful for applications in which the migratory-sharing pattern is common since writes (or migratory reads) for blocks following this pattern do not send invalidations because the owner tile has the only valid copy of the block. Therefore, in *DiCo-Base*, the cores requesting migratory blocks do not update the pointer stored in their L1C$.

The *DiCo-Oracle* implementation (last bar) gives us the potential of *DiCo-CMP*. The results in terms of indirection avoidance are similar to the ones obtained by *Token*. In both cases, the misses falling into the *+3-hop* category are for contended blocks that cannot be solved in two hops due to race conditions. Although *DiCo-Hints AS* does not obtain the same percentage of 2-hop misses than *DiCo-Oracle* (10% less on average), it has similar percentage of *+3-hop* misses, which makes this solution perform close to the oracle case.

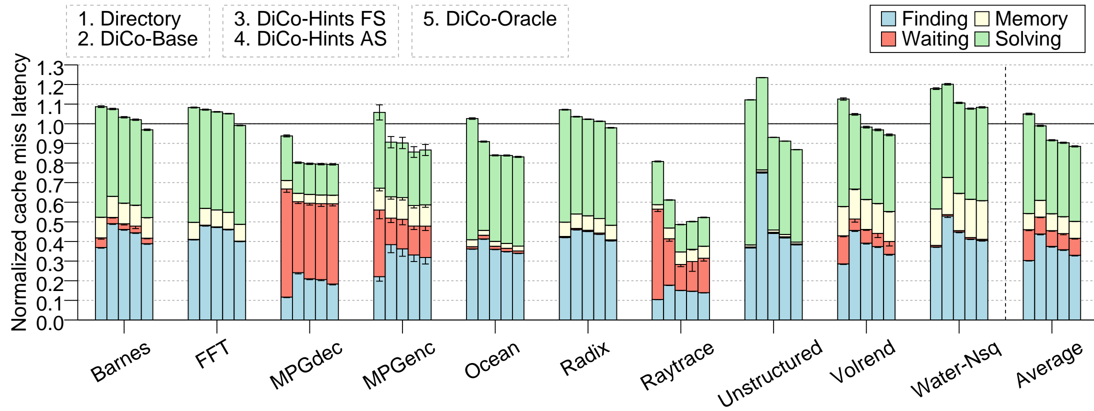


Figure 5.11: Normalized L1 cache miss latency.

5.6.2 Impact on cache miss latencies

The avoidance of the indirection shown by *DiCo-CMP* reduces the average cache miss latency. In addition, *DiCo-CMP* removes the transient states at the directory by putting together the provider of the block and the directory information. This fact reduces the time that the requests need to be waiting at the directory to be processed, which results in even more latency reductions. Figure 5.11 shows L1 cache miss latency for the applications evaluated in this thesis normalized with respect to *Token*. This figure does not consider the overlapping of the misses, and latency is calculated considering each miss individually. Latency is broken down into four segments to understand better in what way *DiCo-CMP* reduces the cache miss latency:

- *Finding*: It is the time elapsed between the issue of a request missing in the local cache and the arrival of the request to the serialization point, i.e., the home tile for directory protocols and the owner tile for *DiCo-CMP*.

- *Waiting*: In directory protocols, it is the time spent waiting at the home tile, because another request for the same block is being processed. In *DiCo-CMP*, this segment represents the period elapsed between the first time that the owner tile is found and the time when the owner tile processes the request to solve the miss. In both cases this period finishes when the directory information for the requested block is accessed.

- *Memory*: It is the time spent getting the data block from main memory when the fresh copy of it is not stored on chip.

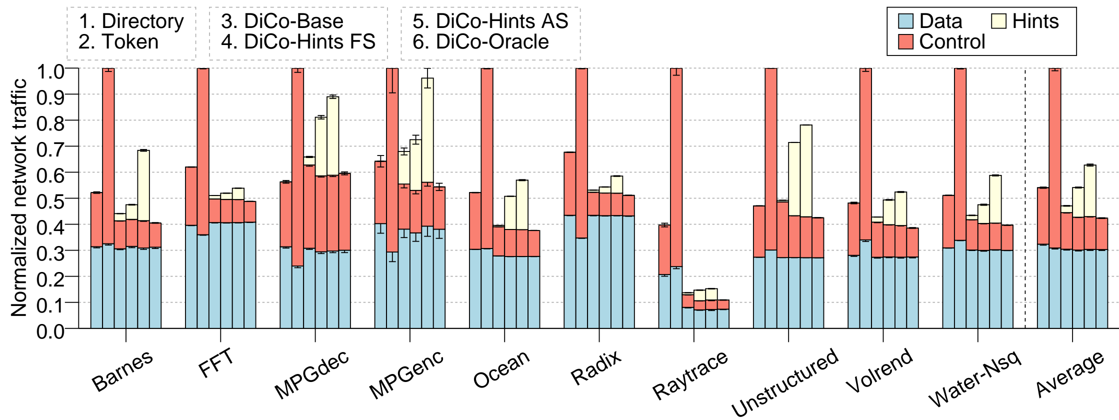
- *Solving*: It is the time elapsed between the request leaves the serialization point and the block is accessed by the requesting processor. This period includes the need of forwarding the request in a directory protocol, and the issue of data, invalidation and acknowledgement messages for both directory and direct coherence protocols.

In general, we can see that all the polices implemented for *DiCo-CMP* reduce the average cache miss latency compared to both a directory protocol and *Token*. In particular *DiCo-Hints AS* obtains reductions of 14% on average over a directory protocol and 10% over *Token*. Moreover, its average latency is very close to the one obtained for the oracle case, which obtains reductions of 16% over a directory protocol.

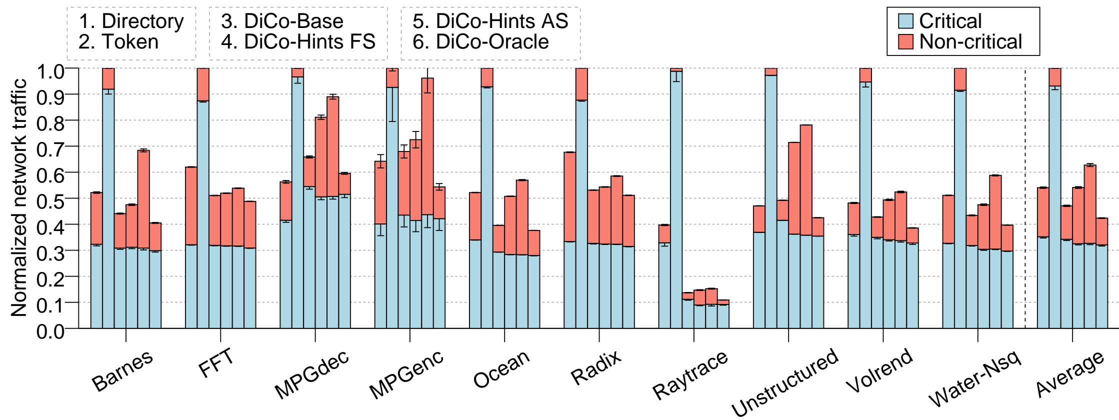
Looking at the different segments into which cache miss latency is split, we can observe that, in general, the finding time is shorter for the directory protocol. This is because in a directory protocol this period always comprises a single hop. However, *DiCo-CMP* can take several hops until the owner tile is found. As we can see, the more accurate are the owner predictions, the shorter is this segment. For example, *Unstructured* has a lot of mis-predictions (*+3-hop* misses) when the base policy is considered, which doubles the finding time compared to a directory protocol. Nevertheless, hints significantly help to reduce this extra latency. Note that for some other applications the increase in the number of *+3-hop* misses is due to race conditions, which do not increase the latency of the cache misses. Finally, for other applications like *FFT*, *Ocean*, and *Radix*, the finding time in the oracle case is a bit shorter than in a directory protocol. This is because sometimes the owner tile is closer to the requesting core than the home tile.

Regarding the waiting time, we can observe that only some applications (*MPGdec*, *MPGenc*, *Raytrace* and *Volrend*) have requests waiting at the home tile during a meaningful time. Since *DiCo-CMP* removes transient states at the directory, this waiting is shortened. This waiting time is usually caused by contended locks and, therefore, reductions in these requests result in a faster acquisition of locks and, finally, in reductions in the number of memory requests, as happens in *Raytrace*.

The memory time does not vary significantly for the evaluated protocols since they incur in the same number of off-chip accesses. However, the solving time is always reduced when *DiCo-CMP* is implemented because for most misses it requires just one hop, in contrast to the two hops (forwarding and data) needed for a directory protocol. This time is not affected by the policy employed



(a) Classified by data, control, and hint messages.



(b) Classified by critical and not critical messages.

Figure 5.12: Normalized network traffic.

because the policy only tries to find the owner tile as soon as possible, but the response goes directly from the owner tile to the requesting one.

5.6.3 Impact on network traffic

Figure 5.12 compares the network traffic generated by the protocols considered in this chapter. In particular, each bar plots the number of bytes transmitted through the interconnection network (the total number of bytes transmitted by all the switches of the interconnect) normalized with respect to *Token*. As we can

see, the fact that *Token* needs to broadcast requests makes this protocol obtain the highest traffic levels. Network traffic can be dramatically reduced when the directory protocol is employed (46% on average). This is because requests are sent to a single tile, the home one, which in turn sends coherence messages just to the cores that must receive them.

Figure 5.12(a) shows the network traffic split into three types: data, control, and hint messages. Compared to the directory protocol, in some applications, mainly in *Raytrace*, the traffic generated by data messages is reduced when *DiCo-CMP* is employed. This reduction is due to the decrease of cache misses commented in the previous section. On the other hand, *DiCo-CMP* saves a meaningful fraction of control messages when compared to the directory protocol. These savings are originated by the elimination of the control messages between the home tile and the other tiles. This reduction allows *DiCo-Base* to reduce network traffic by 13% compared to the directory protocol. In contrast, *DiCo-CMP* introduces hint messages in some configurations in order to achieve more accurate owner predictions. The hints that appear for *DiCo-Base* come as consequence of evictions of owner blocks, as explained in Section 5.4. In general, hints increase network traffic, especially in *Unstructured* in which they are crucial to obtain good performance. However, this traffic is always lower than the reached by *Token* because hints are only sent (if necessary) when the owner cache changes. *DiCo-Hints AS* requires more traffic than *DiCo-Hints FS* at the cost of reducing considerably the storage requirements for the hints mechanism. In general, we can observe that *DiCo-Hints AS* reduces the traffic compared to *Token* up to 37%.

Figure 5.12(b) shows the network traffic split into critical and non-critical messages. This classification is important to know how each protocol could be optimized under heterogeneous networks [32], in which non-critical messages could be sent through low-power wires to save power consumption. In *Token* all broadcast requests are considered critical because it is unknown which ones are going to be actually in the critical path. *Directory* highly reduces the number of critical messages respect to *Token*. We can also observe that *DiCo-CMP* reduces even more this kind of messages, because hints are out of the critical path of the miss. Therefore, under heterogeneous networks, *DiCo-CMP* can save more power consumption than the other protocols (mainly compared to *Token*), and even other more aggressive hints policies could be implemented with small overhead in terms of power.

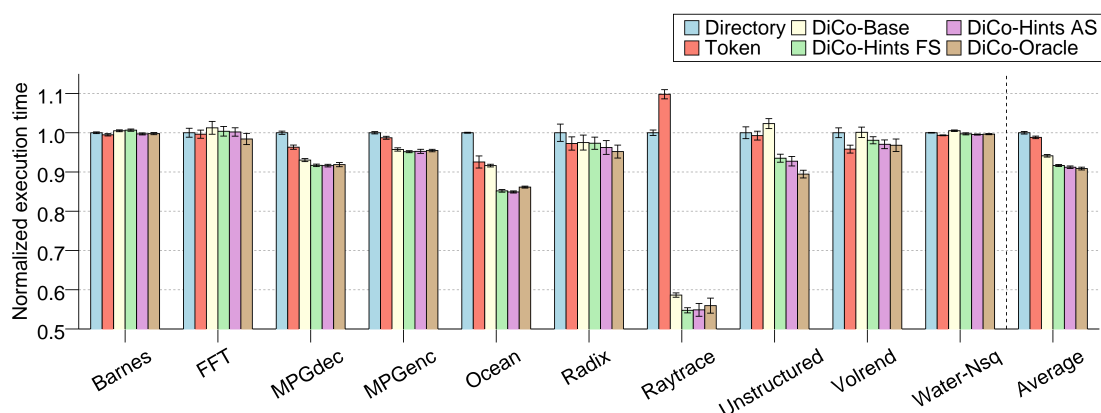


Figure 5.13: Normalized execution time.

5.6.4 Impact on execution time

The ability of avoiding indirection and the savings in network traffic compared to *Token* that *DiCo-CMP* shows, translates into applications' execution time. Figure 5.13 plots the execution times that are obtained for the applications evaluated in this thesis. All the results have been normalized with respect to those observed for the directory protocol.

In general, we can see from Figure 5.13 that *Token* slightly improves a directory protocol (1%). As already discussed, *Token* avoids indirection by broadcasting requests to all caches and, consequently, the network contention can become critical. For applications with few misses with indirection, like *FFT* and *Radix*, *Token* does not have too much advantage with respect to a directory protocol. On the other hand, for *Raytrace* the directory protocol outperforms *Token*. This is caused as consequence of the high contention shown in this application which makes *Token* issue a lot of persistent requests.

DiCo-CMP does not rely on broadcasting but requests are just sent to the potential owner tile. It is clear that the performance achieved by *DiCo-CMP* will depend on its ability to find the current owner tile. We observe improvements in execution time for *DiCo-Base* of 6% compared to the directory protocol and 5% compared to *Token*. In particular, *Raytrace* obtains important reductions in execution time for *DiCo-CMP* over directory and *Token*. This is because *DiCo-CMP* reduces the waiting time at the directory, thus making faster the acquisitions of locks and reducing the number of cache misses more than 50% compared to directory and *Token*.

On the other hand, the use of hints increases the fraction of two-hop misses

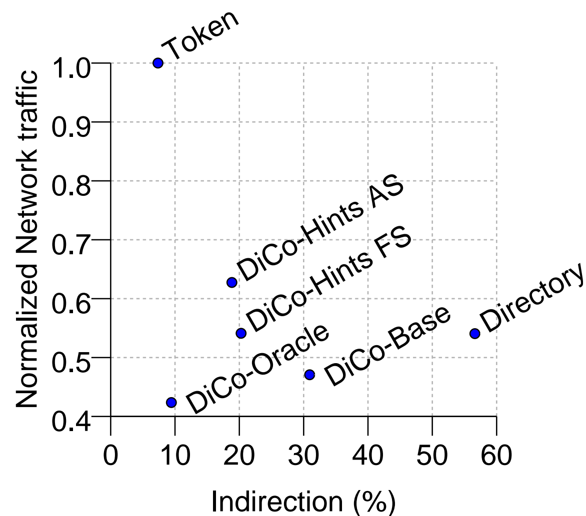


Figure 5.14: Trade-off between network traffic and indirection.

which translates into improvements in terms of execution time. Particularly, *DiCo-Hints AS* obtains better performance than *DiCo-Hints FS* at the cost of generating more network traffic. *DiCo-Hints FS* reduces the execution time by 8% compared to the directory protocol (7% compared to *Token*), while *DiCo-Hints AS* improves the directory protocol by 9% (8% compared to *Token*). Looking back at Figure 5.10, we can observe that these improvements are due to the fact that *DiCo-Hints AS* obtains more accurate predictions than *DiCo-Hints FS*. Finally, we can see that the *DiCo-Hints AS* policy potentially obtains the same results than the unimplementable *DiCo-oracle* policy.

5.6.5 Trade-off between network traffic and indirection

As shown in the introduction of this chapter, direct coherence protocols try to achieve a good trade-off between the small amount of network traffic required by directory protocols and the indirection avoidance of token protocols. Figure 5.14 shows this trade-off for all the protocols evaluated in this chapter. The results presented in this figure are the average of the ten applications evaluated.

As discussed, *Directory* introduces indirection in the critical path of cache misses while *Token* is the protocol that generates higher levels of network traffic. We can observe that all *DiCo-CMP* protocols achieve a good trade-off between network traffic and indirection, since they remove the disadvantages of *Token* and directory protocols. Particularly, *DiCo-Base* is the *DiCo-CMP* policy that

less network traffic generates. On the other hand, *DiCo-Hints AS* is the one that obtains better indirection avoidance. Looking at this figure we can see that *DiCo-Hints FS* has a better trade-off than the other policies but, unfortunately, it requires the addition of an extra full-map sharing codes to each cache entry, which makes it less attractive than *DiCo-Hints AS*. In the next chapter, we discuss with more detail the traffic-area trade-off of direct coherence protocols. Finally, *DiCo-Oracle* presents the best trade-off that can be achieved for direct coherence protocols.

5.7 Conclusions

Directory protocols are the best alternative to design large-scale cache coherence protocols for tiled CMPs. However, they introduce indirection in the critical path of cache misses by having to access the home tile before any coherence action can be performed. An alternative cache coherence protocol that avoids this indirection problem is *Token*. However, *Token* relies on broadcasting requests to all cores, which can lead to prohibitive power levels of traffic (and, therefore, power consumption in the interconnection network) in large-scale tiled CMPs.

In this chapter, we present a family of cache coherence protocols called direct coherence protocols. Direct coherence meets the advantages of directory and token protocols and avoids their problems. In direct coherence, the task of storing up-to-date sharing information and ensuring ordered accesses for every memory block is assigned to the owner tile, instead of to the home one. This protocols avoid the indirection that the access to the directory entails (in directory protocols) by directly sending the requests to the owner tile, thus reducing the average latency of cache misses. Compared to *Token*, direct coherence protocols reduces network traffic by sending coherence messages just to the tiles that must receive them.

We implement *DiCo-CMP*, a cache coherence protocol for tiled CMP architectures based on the *direct coherence* concept. Since we realize that for some applications the base *DiCo-CMP* protocol has difficulties to predict the identity of the owner tile, we decide to enhance owner predictions by using two hints mechanisms. The first one, called *frequent sharers* (FS), needs an extra full-map sharing code per cache entry, which entails more area requirements. The second one, called *address signatures* (AS), encodes the frequent sharers by using bloom filters, thus requiring less area at the cost of increasing network traffic.

In this way, we show that *DiCo-Hints AS*, the best policy for *DiCo-CMP* in

terms of performance, is able to reduce the indirection compared to a directory protocol from 56% to 18% on average. Both this reduction in misses with indirection and the decrease in the waiting time for some applications that *DiCo-CMP* achieves result in reductions of 14% in the average cache miss latencies. Finally, the improvements obtained for the cache miss latencies along with a slight reduction in the number of cache misses lead to improvements of 9% on average in execution time compared to the directory protocol. Moreover, *DiCo-Hints AS* slightly increases traffic requirements compared to a directory protocol and it considerably reduces traffic compared to *Token* (by 37%) and, consequently, the total power consumed in the interconnection network is also reduced. Additionally, we show that the structures and complexity required by *DiCo-CMP* are comparable to those used in a directory protocol, which confirms that the protocol proposed in this chapter is a viable alternative to current cache coherence protocols for tiled CMPs.

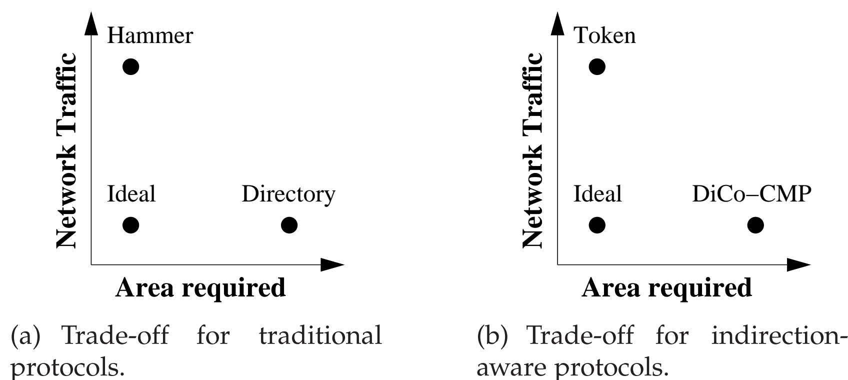
Chapter 6

Traffic-Area Trade-Off in Direct Coherence Protocols

6.1 Introduction

In the previous chapter, we have proposed direct coherence protocols to cope with the trade-off between network traffic and indirection in cache coherence protocols for many-core CMPs. Although we have shown that direct coherence protocols are able to obtain a good traffic-indirection trade-off, the implementations previously evaluated do not take care of the area overhead entailed by the coherence information. In particular, we show that the area overhead required by the most efficient implementation of direct coherence (*DiCo-Hints AS*) is similar to the area overhead introduced by traditional directory protocols. Unfortunately, this area overhead could become prohibitive for many-core CMPs [15].

On the other hand, among the cache coherence protocols described in Section 2.4, the most efficient one in terms of area requirements is *Hammer* [92]. *Hammer* avoids keeping coherence information at the cost of broadcasting requests to all cores. Unfortunately, although it is very efficient in terms of area requirements, it generates a prohibitive amount of network traffic, which translates into excessive power consumption in the interconnection network and other structures as private caches. Figure 6.1(a) shows the trade-off between *Hammer* and directory protocols. Since neither the network traffic generated by *Hammer* nor the extra area required by directory protocols scale with the number of cores, a great



(a) Trade-off for traditional protocols.

(b) Trade-off for indirection-aware protocols.

Figure 6.1: Traffic-area trade-off in cache coherence protocols.

deal of attention was paid in the past to address this traffic-area trade-off in the context of shared-memory multiprocessors [8, 26, 35, 44, 55].

Additionally, these traditional cache coherence protocols introduce the indirection problem addressed in the previous chapter. This problem appears because in both *Hammer* and directory protocols, the ordering point for the requests to the same memory block is the home tile, and all cache misses must reach this ordering point before performing coherence actions. *Token*, described in Section 2.4, and direct coherence protocols have been proposed to deal with this indirection problem. We refer to these protocols as indirection-aware protocols. Particularly, they avoid the access to the home tile through alternative serialization mechanisms. *Token* only cares about requests ordering in case of race conditions. In those cases, a persistent requests mechanism is responsible for ordering all requests to the same memory block. In direct coherence the ordering point for each block is its current owner tile. In this way, indirection is avoided by directly sending the requests to the corresponding owner tile. Therefore, these indirection-aware protocols reduce the latency of cache misses compared to both *Hammer* and directory protocols, which finally translates into performance improvements. Although *Token* entails low memory overhead, it is based on broadcasting requests to all tiles, which is clearly non-scalable. Otherwise, direct coherence sends requests to just one tile, but the implementations presented in the previous chapter assume that a full-map sharing code is added to each cache entry to keep track of the sharers, which does not scale with the number of cores. Figure 6.1(b) shows the traffic-area trade-off for these indirection-aware protocols.

The aim of this chapter is to address the traffic-area trade-off of indirection-

aware protocols for tiled CMPs. Although this trade-off has been widely studied for traditional protocols, in this chapter we consider protocols that try to avoid indirection. Particularly, we perform this study by relaxing the accuracy of the sharing codes used in direct coherence protocols.

As discussed in the previous chapter, the area overhead entailed by direct coherence protocols mainly comes as consequence of the sharing code field added to each cache entry, and the L1 and L2 coherence caches. The area required by the signatures used in *DiCo-Hints AS* can be considered negligible. The most significant area requirements are introduced by the sharing information since a non-scalable full-map sharing code is employed.

In Chapter 4, we have proposed a scalable directory organization for keeping the sharing information in directory-based protocols. Unfortunately, this organization cannot be employed for the sharing information in direct coherence protocols, since they store it in the owner tiles instead of in the home ones, and the proposed scalable organization requires a specific directory interleaving for the information stored in the home tile. Therefore, in this chapter we focus on reducing the size of the sharing information through other techniques, e.g., the utilization of compressed sharing codes.

We have implemented and evaluated several cache coherence protocols based on the direct coherence concept which differ in the sharing code scheme employed. Particularly, *DiCo-LP-1*, which only stores the identity of one sharer along with the data block, *DiCo-BT*, which codifies the directory information just using three bits (in a system with 16 cores), and *DiCo-NoSC*, which does not store any coherence information in the data caches, are the best alternatives. *DiCo-LP-1* presents a good traffic-area trade-off by requiring slightly more area than *Token*, but the same order –$O(log_2 n)$–, and slightly increasing network traffic compared to *DiCo-CMP* (5% on average). *DiCo-BT*, requires less area overhead than *Token* and increases traffic requirements by 9% compared to *DiCo-CMP*. *DiCo-NoSC* does not need to modify the structure of caches to add any extra field and, therefore, introduces less area requirements than *Token* and *DiCo-BT*, and similar to *Hammer* for a 16-tile configuration, but with the same order as *Token* –$O(log_2 n)$–. However, it increases network traffic by 19% compared to *DiCo-CMP*, but still reducing the traffic when compared to *Token* by 25%. Finally, *DiCo-BT* obtains similar execution time than *DiCo-CMP*, while *DiCo-LP-1* increases execution time by 1% and *DiCo-NoSC* by 2% compared to *DiCo-CMP*, due to the increase in network traffic. We believe that the best alternative for a tiled CMP design will depend on its particular constraints.

The rest of the chapter is organized as follows. Section 6.2 gives a classifica-

tion of the cache coherence protocols that could be used in CMP architectures. The different implementations of direct coherence evaluated in this chapter are described in Section 6.3. Section 6.4 shows the performance results and, finally, Section 6.5 concludes the chapter.

6.2 Classification of cache coherence protocols

This section classifies the four cache coherence protocols considered along this thesis as potential candidates to be employed in tiled CMPs (i.e., with unordered networks). In particular, we classify these cache coherence protocols into *traditional* protocols, in which cache misses suffer from indirection, and *indirection-aware* protocols, which try to avoid the indirection problem. For each type, we also differentiate between area-demanding and traffic-intensive protocols.

6.2.1 Traditional protocols

In traditional protocols, the requests issued by several cores to the same block are serialized through the home tile, which enforces cache coherence. Therefore, all requests must be sent to the home tile before any coherence actions can be performed. Then, the request is forwarded to the corresponding tiles according to the coherence information (or it is broadcast if the protocol does not maintain any coherence information). All processors that receive the forwarded request answer by sending either an acknowledgement or a data message to the requesting core. The requesting core accesses the block when it receives all the acknowledgement and data messages. The access to the home tile introduces indirection, which causes that cache misses are solved with three hops in the critical path.

Examples of these traditional protocols are *Hammer* and directory protocols. As commented in the introduction, *Hammer* has the drawback of generating a considerable amount of network traffic. On the other hand, directory protocols that use a precise sharing code to keep track of cached blocks introduce an area overhead that does not scale properly with the number of cores.

6.2.2 Indirection-aware protocols

Recently, new cache coherence protocols have been proposed to avoid the indirection problem of traditional protocols. For example, *Token* avoids indirection by broadcasting requests to all tiles and maintains coherence through a token

counting mechanism. Although the storage required to store the tokens of each block is reasonable, network requirements are prohibitive for may-core CMPs. On the other hand, the direct coherence protocols presented in the previous chapter keep traffic low by sending requests to only one tile. However, coherence information used in these implementations include a full-map sharing code per cache entry, whose area do not scale with the number of cores.

6.2.3 Summary

Table 6.1: Summary of cache coherence protocols.

	Traditional	**Indirection-aware**
Traffic-intensive	Hammer	Token
Area-demanding	Directory	DiCo

Table 6.1 summarizes all cache coherence protocols considered in this thesis. *Hammer* and *Token* are based on broadcasting requests on every cache miss. Therefore, although the storage required by these protocols to keep coherence is small, they generate a prohibitive amount of network traffic. On the other hand, directory and DiCo achieve more efficient utilization of the interconnection network at the cost of increasing storage requirements compared with *Hammer* and *Token* protocols.

6.3 Reducing memory overhead

DiCo-CMP needs two structures that keep the identity of the tile where the owner copy of the block resides, the L1C$ and the L2C$. These two structures do not compromise scalability because they have a small number of entries and each one stores a tag and a pointer to the owner tile ($log_2 n$ bits, where n is the number of cores). The L2C$ is needed to solve cache misses in *DiCo-CMP*, since it ensures that the tile that keeps coherence for each block can always be found. On the other hand, the L1C$ is required to avoid indirection in cache misses and, therefore, it is essential to obtain good performance. Moreover, the L2C$ allows read misses to be solved by sending only one forwarding request to the owner tile, since it stores the identity of the owner tile, which significantly reduces network traffic when compared to broadcast-based protocols.

Apart from these structures, *DiCo-CMP* also adds a full-map sharing code to each data cache entry. The memory overhead introduced by this sharing code could become prohibitive in many-core CMPs. In this section, we describe some alternatives that differ in the sharing code scheme added to each entry of the data caches. Since these alternatives always include the L1C$ and the L2C$, they have area requirements of at least $O(log_2 n)$. The particular compressed sharing code employed impacts on the number of invalidations sent for write misses. Next, we comment on the different implementations of direct coherence protocols that we have evaluated.

DiCo-FM is the *DiCo-CMP* protocol described in the previous chapter and, therefore, it adds a full-map sharing code to each data cache. Particularly, it uses address signatures to keep the memory overhead introduced by the hints mechanism as low as possible.

DiCo-CV-K reduces the size of the sharing code field by using a *coarse vector* [44] instead of a full-map sharing code. In a coarse vector, each bit represents a group of K tiles, instead of just one. A bit is set when at least one of the tiles in the group holds the block in its private cache. Therefore, even when just one of the tiles in the group requested a particular block, all tiles belonging to that group will receive an invalidation message before the block can be written. Particularly, we study a configuration that uses a coarse vector sharing code with $K = 2$. In this case, 8 bits are needed for a 16-core configuration. Although this sharing code reduces the memory required by the protocol, its size still increases linearly with the number of cores.

DiCo-LP-P employs a *limited pointer* sharing code [26]. In this scheme, each entry has a limited number of pointers for the first P sharers of the block. Actually, since *DiCo-CMP* always stores the information about the owner tile in the L2C$, the first pointer is employed to store the identity of the second sharer of the block. When the sharing degree of a block is greater than $P + 1$, write misses are solved by broadcasting invalidations to all tiles. Therefore, apart from the pointers, it is necessary an extra bit indicating the overflow situation. However, this situation is not very frequent since the sharing degree of the applications is usually low [38]. In particular, we evaluate this protocol with a P value of 1. Under this assumption, the number of bits needed to store the sharing information considering 16 cores is 5.

DiCo-BT uses a sharing code based on a *binary tree* [1]. In this approach, tiles are recursively grouped into clusters of two elements, thus leading to a binary tree with the tiles located at the leaves. The information stored in the sharing code is the smallest cluster that covers all the sharers. Since this scheme assumes

that for each block the binary tree is computed from a particular leave (the one representing the home tile), it is only necessary to store the number of the level in the tree, i.e., 3 bits for a 16-core configuration.

Finally, *DiCo-NoSC* (no sharing code) does not maintain any coherence information along with the owner block. In this way, this protocol does not need to modify the structure of data caches to add any field. This lack of information implies broadcasting invalidation messages to all tiles upon write misses, although this is only necessary for blocks in shared state because the owner tile is always known in *DiCo-CMP*. This scheme incurs in more network traffic compared to the previous ones. However, it falls into less traffic than *Hammer* and *Token*. This is because *Hammer* requires broadcasting requests on every cache miss, and what is more expensive in a network with multicast support, every tile that receives the request answers with a control message. On the other hand, although *Token* avoids these response messages, it also relies on broadcasting requests for all cache misses.

Table 6.2: Bits required for storing coherence information.

Protocol	**Sharing Code**	**Bits L1 cache and L2 cache**	**Bits L1C$ and L2C$**	**Order**
DiCo-FM	Full-map	n	$log_2 n$	$O(n)$
DiCo-CV-K	Coarse vector	$\frac{n}{K}$	$log_2 n$	$O(n)$
DiCo-LP-P	Limited pointers	$1 + P \times log_2 n$	$log_2 n$	$O(log_2 n)$
DiCo-BT	Binary Tree	$\lceil log_2(1 + log_2 n) \rceil$	$log_2 n$	$O(log_2 n)$
DiCo-NoSC	None	0	$log_2 n$	$O(log_2 n)$

Table 6.2 shows the bits required for storing coherence information in each implementation, both for the coherence caches (L1C$ and L2C$) and for the data caches (L1 and L2). Other compressed sharing codes, like *tristate* [8], *gray-tristate* [85] or *binary tree with subtrees* [1] could also be implemented instead of the shown in this table. However, for a 16-core tiled CMP, they incur in similar overhead than *DiCo-CV-2* (8, 8 and 7 bits respectively), that does not significantly increases network traffic, as we will see in Section 6.4.2. For a greater number of cores, these compressed sharing codes can be more appropriate.

6.4 Evaluation results and analysis

In this section, we compare the different alternatives described in the previous section with all the base protocols described in this thesis. First, we study the area overhead of each protocol and the impact that the different sharing codes have on network traffic. We also study the traffic-area trade-off for this protocols. Then, we show the results in terms of applications' execution time and, finally, we perform an overall analysis of the evaluated protocols taking into account the three main design goals for cache coherence protocols: performance, network traffic, and area requirements.

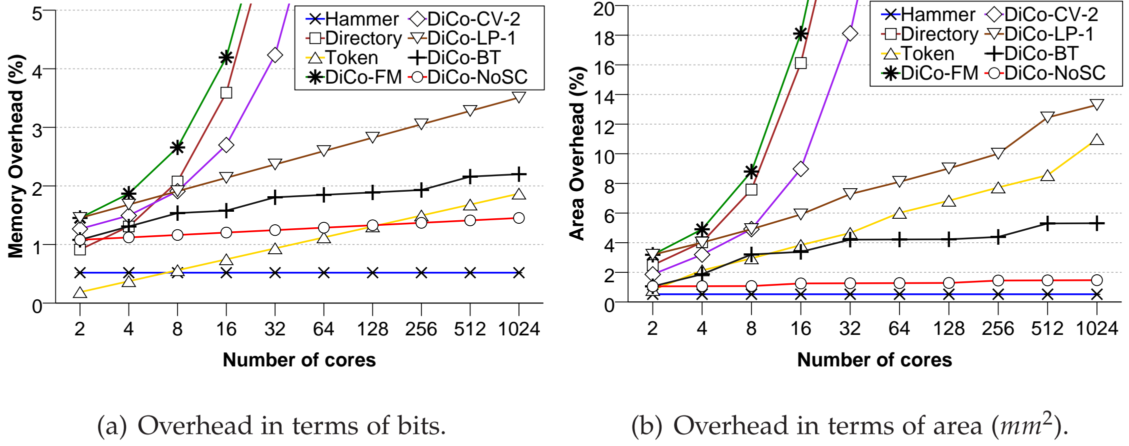
6.4.1 Impact on area overhead

First, we compare the memory overhead introduced by the coherence information for the cache coherence protocols evaluated in this chapter. Although some protocols can entail extra overhead as a consequence of the additional mechanisms that they demand (e.g., timeouts for reissuing requests or large tables for keeping active persistent requests in *Token*), we only consider the amount of memory required to keep coherence information. Obviously, the extra tags required to store this information (e.g., for the L1C$ and L2C$) are also considered in this study. Figure 6.2 shows the storage overhead introduced by these protocols in terms of both number of bits and estimated area (calculated with CACTI, as explained in Chapter 3). The overhead is plotted for varying number of cores from 2 to 1024.

Although the original *Hammer* protocol does not require any coherence information, our optimized version for CMPs adds a new structure to the home tile. This structure is a 512-set 4-way cache that contains a copy of the tags for blocks stored in the L1 caches but not in the L2 cache. Remember that our protocols do not force inclusion between both cache levels. However, this structure introduces a slight overhead which keeps constant with the number of cores.

Directory stores the directory information either in the L2 tags, when the L2 cache holds a copy of the block, or in a distributed directory cache, when the block is stored in any of the L1 caches but not in the L2 cache. Since the information is stored using a full-map sharing code, the number of required bits is n, and consequently the width of each directory entry grows linearly with the number of cores.

Token keeps the token count for any block stored both in the L1 and L2 caches. This information only requires $\lceil log_2(n+1) \rceil$ bits for both the owner-token bit



(a) Overhead in terms of bits.

(b) Overhead in terms of area (mm^2).

Figure 6.2: Overhead introduced by the cache coherence protocols.

and the non-owner token count. These additional bits are stored in the tags' part of both cache levels. In this way, *Token* has acceptable scalability in terms of area. Note that we do not account for the size of the tables needed by the persistent requests mechanism, which could restrict the scalability of token coherence in some situations [37].

DiCo-CMP stores directory information along with each owner block held in the L1 and L2 caches. Therefore, a full-map sharing code is added to the tags' part of each cache entry. Moreover, it uses two structures that store the identity of the owner tile, the L1C$ and the L2C$. Each entry in these structures contains a tag and an owner field, which requires $log_2 n$ bits. Therefore, this is the protocol that more area overhead entails.

In this chapter, we propose to reduce this overhead by introducing compressed sharing codes in *DiCo-CMP*, also named in this chapter as *DiCo-FM*. *DiCo-CV-2* saves storage compared to *DiCo-FM* but it is still non-scalable. In contrast, *DiCo-LP-1*, which only adds a pointer for the second sharer of the block (the first one is given by the L2C$) has better scalability –$O(log_2 n)$–. *DiCo-BT* reduces even more the area requirements compared to *DiCo-LP-1*, and it scales better than *Token*. Finally, *DiCo-NoSC*, which does not require to modify data caches to add coherence information, is the implementation of DiCo with less overhead (although it still has order $O(log_2 n)$ due to the need of the coherence caches), at the cost of increasing network traffic. Finally, we can see that a small

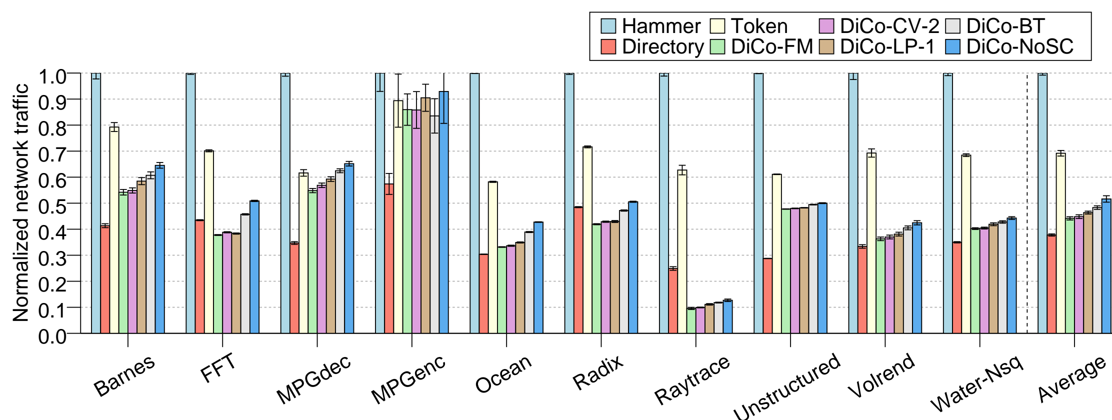


Figure 6.3: Normalized network traffic.

overhead in the number of required bits results in a significant overhead when the area of the structures is considered.

6.4.2 Impact on network traffic

Figure 6.3 compares the network traffic generated by the protocols discussed previously. Each bar plots the number of bytes transmitted through the interconnection network normalized with respect to *Hammer*.

As expected, *Hammer* introduces much more network traffic than the other protocols due to the lack of coherence information, which implies broadcasting requests to all cores and receiving the corresponding acknowledgements. *Directory* reduces considerably traffic by adding a full-map sharing code that filters unnecessary invalidations. *Token* generates more network traffic than *Directory*, because it relies on broadcasts, and less than *Hammer*, because it does not need to receive acknowledgements from tiles without tokens (i.e., the tiles that do not share the block). Finally, *DiCo-FM* slightly increases traffic requirements compared to *Directory*. This increase is due to the issue of hints to achieve good owner predictions.

In general, we can see that compressed sharing codes increase network traffic compared to a full-map sharing code. However, the increase in traffic is admissible. Particularly, the most scalable alternatives, *DiCo-LP-1*, *DiCo-BT* and *DiCo-NoSC*, increase network traffic by 5%, 9% and 19% compared to *DiCo-FM*, respectively. Even *DiCo-NoSC*, which does not have any sharing code, generates

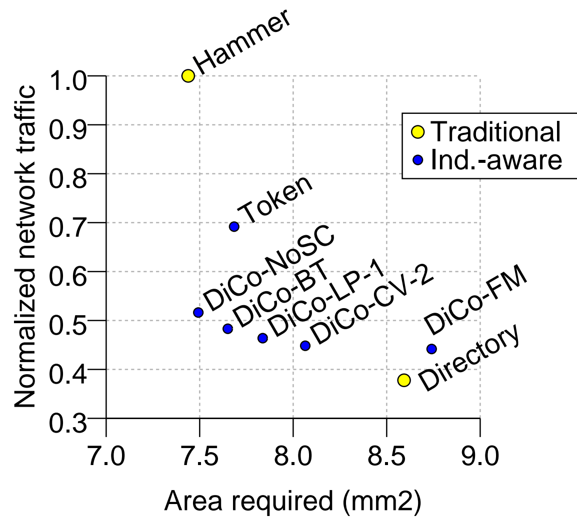


Figure 6.4: Traffic-area trade-off.

an acceptable amount of network traffic (25% less traffic than *Token* and 48% less traffic than *Hammer*).

6.4.3 Trade-off between network traffic and area requirements

Figure 6.4 shows the traffic-area trade-off for all the protocols evaluated in this chapter. Results of network traffic represent the average of all applications. The figure also differentiates between traditional and indirection-aware protocols. We can see that, in general, the base protocols aimed to be used with tiled CMPs do not have a good traffic-area trade-off: both *Hammer* and *Token* are constrained by traffic whilst both *Directory* and *DiCo-FM* are constrained by area.

However, the use of different compressed sharing codes for *DiCo-CMP* can lead to a good compromise between network traffic and area requirements. In particular, *DiCo-LP-1*, *DiCo-BT* and *DiCo-NoSC* are very close to an ideal protocol with the best of the base protocols, and also avoiding the indirection problem. The difference is that *DiCo-LP-1* is more efficient in terms of network traffic whilst *DiCo-NoSC* is more efficient in terms of area requirements. Particularly, *DiCo-LP-1* requires slightly more area than *Token*, but the same order –$O(log_2 n)$–, and slightly increases network traffic compared to *DiCo-FM* (9% on average). *DiCo-BT* requires less area overhead than *Token* for the configuration used in

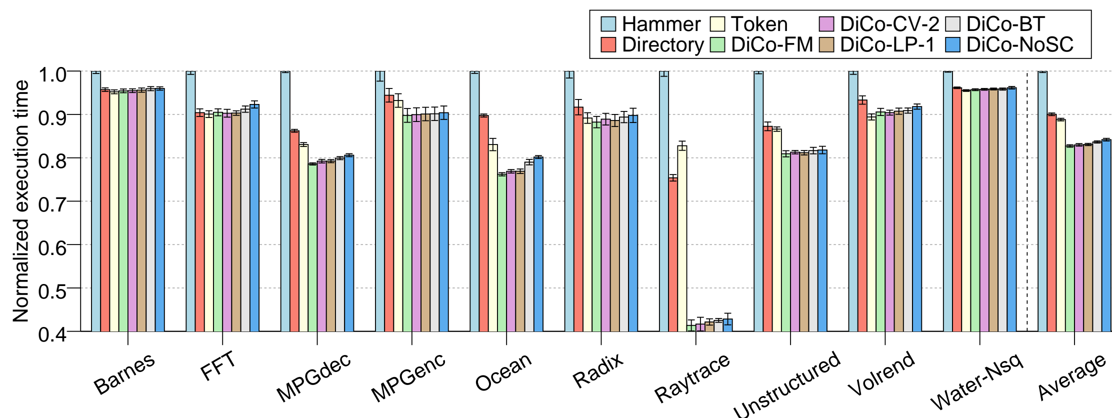


Figure 6.5: Normalized execution time.

this study and it reduces considerably the network traffic by 30%. The main advantage of *DiCo-NoSC* is that it does not need to modify the structure of the caches to add any extra field and, therefore, it introduces less area requirements than *Token* and similar requirements than *Hammer*, but with the same order than *Token* –$O(log_2 n)$–. However, it increases network traffic by 19% compared to *DiCo-FM*, but still reducing the traffic when compared to *Token* by 25%.

6.4.4 Impact on execution time

Figure 6.5 plots the average execution times for the applications evaluated in this thesis normalized with respect to *Hammer*. Compared to *Hammer*, *Directory* improves performance for all applications as a consequence of an important reduction in terms of network traffic. Moreover, on each miss *Hammer* must wait for all the acknowledgement messages before the requested block can be accessed. On the contrary, in *Directory* only write misses must wait for acknowledgements.

On the other hand, indirection-aware protocols reduce average execution time when compared to traditional protocols. Particularly, *Token* obtains average improvements of 11% compared to *Hammer* and 1% compared to *Directory*. *DiCo-FM* improves the execution time by 17%, 9% and 8% compared to *Hammer*, *Directory* and *Token*, respectively. On the other hand, when *DiCo-CMP* employs compressed sharing codes, the execution time slightly increases. However, it remains close to *DiCo-FM*. For *DiCo-CV-2* and *DiCo-LP-1* the increase in exe-

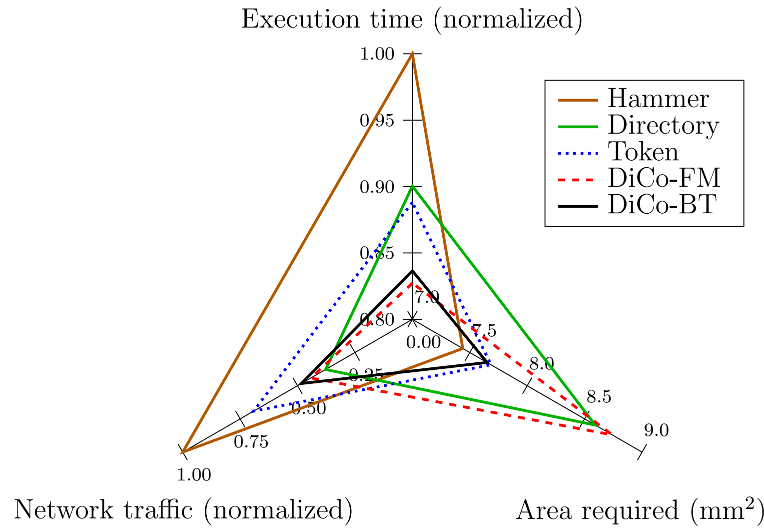


Figure 6.6: Trade-off among the three main design goals for coherence protocols.

cution time is negligible. *DiCo-BT* increases execution time by 1% and, finally, *DiCo-NoSC* increases execution time by 2%.

6.4.5 Overall analysis

Figure 6.6 shows the trade-off among execution time, network traffic, and area requirements for the base protocols evaluated in this chapter and *DiCo-BT*, which constitutes a good alternative when these three metrics are considered. In this way, this graph summarizes the evaluation carried out both in this chapter and in the previous one. Results in terms of execution time and network traffic represent the average of all applications, normalized with respect to *Hammer*. Results in terms of area requirements correspond to the area in mm^2 of each protocol considering both the data caches and the extra structures required to keep the coherence information.

We can see that, in general, the base protocols do not obtain good results for all the metrics. In particular, *Hammer* has the highest traffic levels and execution time, but it also has the lowest area requirements ($7.4mm^2$). In contrast, *Directory* reduces both execution time and network traffic compared to *Hammer* (by 10% and 61%, respectively) at the cost of increasing area requirements ($8.59mm^2$ for a 16-tiled CMP, and $O(n)$). Although *Token* has acceptable area requirements ($7.68mm^2$ for a 16-tiled CMP) it is limited by traffic, requiring twice the traffic

required by *Directory*. Finally, *DiCo-FM*, that reduces both execution time and traffic requirements when compared to *Token* (by 9% and 36%, respectively), is the one with the highest area requirements ($8.74mm^2$ for a 16-tiled CMP, and $O(n)$).

However, the use of different compressed sharing codes for *DiCo-CMP* can lead to a good compromise between network traffic and area requirements, and still guaranteeing low average execution time. In general, *DiCo-LP-1*, *DiCo-BT* and *DiCo-NoSC* are very close to an ideal protocol with the best characteristics of the base protocols, but for the sake of clarity, we only show the trade-off for *DiCo-BT*. *DiCo-BT* requires less area ($7.65mm^2$ for a 16-tiled CMP) than all evaluated protocols except *Hammer* (and *DiCo-NoSC*), it also generates similar network traffic than *DiCo-FM* and, finally, it has a low average execution time (increasing just by 1% the best approach, *DiCo-FM*).

6.5 Conclusions

Two traditional protocols that can be used in tiled CMPs with unordered networks are *Directory* and *Hammer*. *Directory* commonly keeps track of the sharers by means of a full-map sharing code that does not scale with the number of cores. On the other hand, *Hammer* avoid this extra storage by broadcasting requests, which is not power-efficient. Therefore, researchers put their effort in addressing this trade-off. However, both protocols introduce the indirection problem leading to inefficiencies in terms of performance.

Token has been recently proposed to cope with the indirection problem of traditional protocols. Although it can avoid indirection in most cases, this protocol is also based on broadcasting requests for every cache miss, like *Hammer*. On the other hand, in the previous chapter, we proposed direct coherence protocols to address the trade-off between network traffic and indirection in cache coherence protocols. Although we have shown that direct coherence protocols are able to obtain a good traffic-indirection trade-off, the implementations evaluated in the previous chapter do not take especial care of the area overhead entailed by the coherence protocol. Since *Token* does not scale in terms of traffic requirements and *DiCo-CMP* is not scalable in terms of area requirements, it is necessary to design new cache coherence protocols with the advantages of the existing ones.

Particularly, this chapter addresses the traffic-area trade-off of indirection-aware cache coherence protocols through several implementations of direct coherence protocols for tiled CMPs. We evaluate several cache coherence protocols

that differ in the amount of coherence information that they store. *DiCo-LP-1*, which only stores the identity of one sharer along with the data block, *DiCo-BT*, which codifies the directory information just using three bits, and *DiCo-NoSC*, which does not store any coherence information in the data caches, are the alternatives that achieve a better compromise between traffic and area. It is important to note that *DiCo-NoSC* does not need to modify the structure of the caches and, therefore, has less area requirements than *Token* and all the other implementations of direct coherence protocols, and similar requirements compared to *Hammer* for the configurations evaluated. However, it increases network traffic by 19% compared to *DiCo-CMP*, but still reducing the traffic when compared to *Token* by 25%.

Finally, all these implementations obtain similar execution times when compared to *DiCo-CMP*, being *DiCo-NoSC* the one that obtains the worse result, but just by 2%. Note that *DiCo-NoSC* still improves execution time compared to *Token* and *Directory* by 6% and 7%, respectively. Therefore, since some of the evaluated alternatives avoid the indirection problem requiring both low area and traffic requirements, we believe that they can be considered for being implemented in future many-core tiled CMPs depending on the particular system constraints.

Chapter 7

A Distance-Aware Mapping Policy for NUCA Caches

7.1 Introduction

In the previous chapters of this thesis, we have focused on cache coherence protocols for large-scale CMPs. For that study we have assumed a shared L2 cache organization with a physical mapping of blocks to cache banks. In this chapter, we discuss the perks and drawbacks of this organization, and we propose an alternative mapping policy.

As discussed in the introduction of this thesis, an important decision when designing tiled CMPs is how to organize the last-level on-chip cache, i.e., the L2 cache in this thesis, since cache misses at this level result in long-latency off-chip accesses. The two common ways of organizing this cache level are *private* to the local core or *shared* among all cores. The private L2 cache organization, ensures fast access to the L2 cache. However, it has two main drawbacks that could lead to an inefficient use of the aggregate L2 cache capacity. First, local L2 banks keep a copy of the blocks requested by the corresponding core, potentially replicating blocks in multiple L2 cache banks. Second, load balancing problems appear when the working set accessed by all the cores is heterogeneous, i.e., some banks may be over-utilized while others are under-utilized. Since these drawbacks can result in more off-chip accesses, which are very expensive, latest commercial CMPs implement a shared cache organization [63, 104, 103, 53].

The shared L2 cache organization, also called non-uniform cache architecture

(NUCA) [57], provides more efficient use of the L2 cache by storing only one copy of each block and by distributing the copies across the different banks. The main downside of this organization for many-core CMPs is the long L2 access latency, since it depends on the bank wherein the block is allocated, i.e., the home bank or home tile.

The most straightforward way of distributing blocks among the different tiles is by using a physical mapping policy in which a set of bits in the block address defines the home bank for every block. Some recent proposals [52, 123] and commercial CMPs [63, 104] choose the less significant bits[1] for selecting the home bank. In this way, blocks are assigned to banks in a round-robin fashion with block-size granularity. This random distribution of blocks does not take into account the distance between the requesting core and the home bank on a L1 cache miss. Moreover, the average distance between two tiles in the system significantly increases with the size of the CMP, which can become a performance problem for many-core CMPs.

On the other hand, page-size granularity seems to be a better choice than block-size granularity for future tiled CMPs because (1) it is more appropriate for new technologies aimed to reduce off-chip latencies, like 3D stacking memory architectures [69], and (2) it provides flexibility to the OS for implementing more efficient mapping policies [34], such as *first-touch*, which has been widely used in NUMA architectures to achieve better locality in the memory accesses. The behavior of a first-touch policy is similar to that of a private cache organization but without replication. One nice aspect of this policy is that it is dynamic in the sense that pages are mapped to cache banks depending on the particular memory access pattern. However, this policy can increase off-chip accesses when the working set of the application is not well-balanced among cores. Figure 7.1 shows the trade-off between a *round-robin* policy and *first-touch* policy in NUCA caches. This chapter addresses this trade-off.

Additionally, many-core CMP architectures are very suitable for throughput computing [29] and, therefore, they constitute a highly attractive choice for commercial servers in which several programs are running at the same time using different subsets of the cores available on chip. The use of these architectures as commercial servers emphasize the need of efficient mapping policies because (1) data is shared by cores that are placed in a small region of the chip, but with a round-robin policy they could map to any bank in the chip, and (2) more work-

[1] In this chapter, when we refer to the less significant bits of an address we do not consider the block offset.

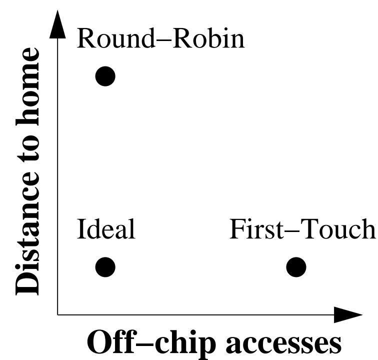


Figure 7.1: Trade-off between *round-robin* and *first-touch* policies.

ing set imbalance can occur in these systems since the applications running on them could have very different memory requirements.

In this chapter, we propose the *distance-aware round-robin* mapping policy, an OS-managed policy which does not require extra hardware structures. This policy tries to map the pages to the local bank of the first requesting core, like a first-touch policy, but also introduces an upper bound on the deviation of the distribution of memory pages among cache banks, which lessens the number of off-chip accesses.

We also observe that OS-managed distance-aware mapping policies can hurt the L1 cache hit rate in some cases. This happens when the same bits that define the home bank are used for indexing the private L1 caches. In these cases, some sets in the cache are overloaded while others remain almost unused. This imbalance increases conflict misses. Hence, we propose to avoid the home bank bits for indexing the L1 caches when distance-aware mapping policies are employed.

Since the proposed mapping policy is particularly appropriate for systems where several applications are running simultaneously, we have extended the workloads used in this thesis to evaluate the proposal presented in this chapter with four multi-programmed workloads, which have been already explained in Chapter 3. Our proposal obtains average improvements of 5% for parallel applications and 14% for multi-programmed workloads over a round-robin policy. In terms of network traffic, our proposal obtains average reductions of 31% for parallel applications and 68% for multi-programmed workloads. When compared to a first-touch policy average improvements of 2% for parallel applications and

7% for multi-programmed workloads are obtained, slightly increasing on-chip network traffic.

The rest of the chapter is organized as follows. The related work and a background on mapping policies for NUCA caches is given in Section 7.2. Section 7.3 describes the distance-aware round-robin mapping policy. The impact of distance-aware mapping policies on private cache miss rate is discussed in Section 7.4. Section 7.5 shows the performance results. Finally, Section 7.6 concludes the chapter.

7.2 Background and related work

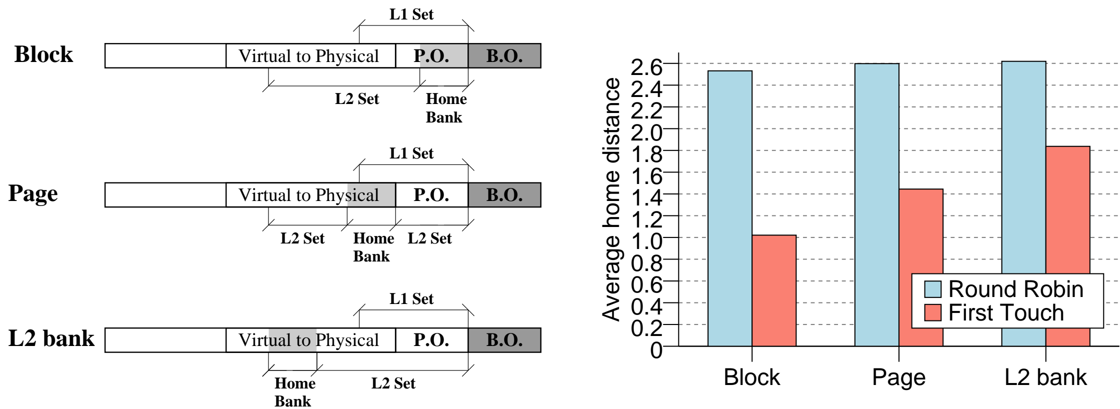
This section present a background on mapping policies for NUCA caches. Then, we comment on a review of the previous works which are related to the mapping policy proposed in this chapter.

7.2.1 Background on mapping policies in NUCA caches

Non-uniform cache access (NUCA) caches [57] are a set of cache banks distributed across the chip and connected through a point-to-point network. Although cache banks are physically distributed, they constitute a logically shared cache (the L2 cache level in this thesis). Therefore, the mapping of memory blocks to cache entries is not only defined by the cache set, but also by the cache bank. The cache bank where a particular block maps is called the *home* bank for that block.

Most CMP architectures that implement NUCA caches map memory blocks to cache banks by taking some fixed bits of the physical address of the block [63, 104]. This physical mapping spreads blocks uniformly among cache banks, resulting in optimal utilization of the cache storage. Commonly, the bits taken to select the cache bank for a particular block are the less significant ones, leading to a block-grained interleaving (*Block* diagram in Figure 7.2(a)). One of the advantages of this interleaving is that it offers less contention at the home tile by distributing contiguous memory blocks across different cache banks.

Another option is to use an interleaving with a granularity of at least the size of a page (e.g., *Page* and *L2 bank* diagrams in Figure 7.2(a)). As shown in Figure 7.2(b), when a physical mapping policy is considered, the granularity of the interleaving does not affect significantly the average distance to the home bank (see the *Round Robin* bars). However, this interleaving becomes an important



(a) Different granularities of interleaving (P.O.=Page offset, B.O.=Block offset).

(b) Impact on average home distance for the SPLASH-2 benchmark suite and 16 cores.

Figure 7.2: Granularity of L2 cache interleaving and its impact on average home distance.

decision when either 3D-stacked memory or OS-managed mapping techniques are considered (as shown with the *First Touch* bars).

A 3D-stacked memory design can offer latency reductions for off-chip accesses when a coarse-grained interleaving (at least of page size) is employed. In tiled CMPs with 3D stacking memory, each tile includes a memory controller for the memory bank that it handles [69]. Low-latency, high-bandwidth and very dense *vertical* links [39] interconnect the on-chip controller with the off-chip memory. These vertical links provide fast access to main memory. On an L2 cache miss, it is necessary to reach the memory controller of the memory bank where the block is stored. If the memory controller is placed in a different tile than the home L2 bank, a *horizontal* on-chip communication is entailed. Since blocks in memory are handled at page-size granularity, it is not possible to assign the same mapping for the L2 cache if a block-size granularity is considered. Differently, with a granularity of at least the size of a page the same mapping can be assigned to both memories, thus avoiding the *horizontal* latency.

The other advantage of a coarse-grained interleaving is that it allows the OS to manage the cache mapping without requiring extra hardware support [34]. The OS maps a page to a particular bank the first time that the page is referenced, i.e, a memory miss. At that moment, the OS assigns a physical address to the virtual address of the page. Therefore, some bits in the address of the page are going to change (*Virtual to Physical* field in figure 7.2(a)). Then, the

OS can control the cache mapping by assigning to this page a physical address that maps to the desired bank. For example, a first-touch policy can be easily implemented by assigning an address that physically maps to the tile wherein the core that is accessing the page resides. The OS only needs to keep in software a list of available physical addresses for each memory bank. With a first-touch mapping policy, finer granularity offers shorter average distance between the missing L1 cache and the home L2 bank, as shown in Figure 7.2(b). Therefore, it is preferable to use a grain size as fine as possible. Since block granularity is not suitable for OS-managed mapping, the finest granularity possible is achieved by taking the less significant bits of the *Virtual to Physical* field, i.e., a page-grained interleaving.

The drawback of a first-touch policy is that applications with a working set not balanced among cores do not make optimal use of the total L2 capacity. This happens more frequently in commercial servers where different applications with different memory requirements run on the same system, or when some applications are running in a set of cores while the other cores remain idle. To avoid this situation, policies like *cache pressure* [34] can be implemented. Cache pressure uses bloom filters to collect cache accesses in order to determine the *pressure* of the different data mapping to cache banks. In this way, newly accessed pages are not mapped to the most pressured caches. However, this approach has several drawbacks, as we explain in the following section.

7.2.2 Related work

There are several ways of reducing cache access latency in NUCA caches. The most relevant ways are data migration, data replication or to perform an intelligent data mapping to cache banks. Next, we comment on the most important works for these approaches.

Kim *et al.*[57] presented non-uniform cache access (NUCA) caches. They studied both a static mapping of blocks to caches and a dynamic mapping based on *spread sets*. In such dynamic mapping, a block can only be allocated in a particular *bank set*, but this bank set can be comprised of several cache banks that act as *ways* of the bank set. In this way, a memory block can migrate from a bank far from the processor to another bank closer if the block is expected to be accessed frequently. Chishti *et al.* [33] achieved more flexibility than the original dynamic NUCA approach by decoupling tag and data arrays, and by adding some pointers from tags to data, and vice versa. The tag array is centralized and accessed before the data array, which is logically organized as distance-

groups. Again, memory blocks can reside in different banks within the same bank set. Differently from the last two proposals, Beckmann *et al.* [18], considered block migration in multiprocessor systems. They proposed a new distribution of the components in the die, where the processing cores are placed around the perimeter of a NUCA L2 cache. Migration is also performed among cache banks belonging to the same bank set. The block search is performed in two phases, both requiring broadcasting the requests. Unfortunately, these proposals have two main drawbacks. First, there are data placement restrictions because data can only be allocated in a particular bank set and, second, data access requires checking multiple cache banks, which increases network traffic and power consumption.

Zhang *et al.* [123] proposed victim replication, a technique that allows some blocks evicted from an L1 cache to be stored in the local L2 bank. In this way, the next cache miss for this block will find it at the local tile, thus reducing miss latency. Therefore, all L1 cache misses must look for the block at the local L2 bank before the request is sent to the home bank. This scheme also has two main drawbacks. First, replication reduces the total L2 cache capacity. Second, forwarding and invalidation requests must also check the L2 tags in addition to the L1 tags. Later on, in [122], they proposed victim migration as an optimization that removes some blocks from the L2 home bank when they are frequently requested by a remote core. Now, the drawback is that an extra structure is required to keep the tags of migrated blocks. Moreover, in both proposals, write misses are not accelerated because they have to access the home tile since coherence information does not migrate along with the data blocks.

Differently from all the previous approaches, and closer to ours, Cho and Jin [34] proposed using a page-size granularity (instead of block-size). In this way, the OS can manage the mapping policy, e.g, a first-touch mapping policy can be implemented. In order to deal with the unbalanced utilization of the cache banks, they propose using bloom filters that collect cache access statistics. If a cache bank is *pressured*, the neighbouring banks can be used to allocate new pages. However, this proposal has several implementation issues which are avoided in our proposed mapping policy. First, it requires extra hardware, (e.g., bloom filters that have to be reset after a timeout period). Second, an accurate metric to decide whether a cache is pressured or not can be difficult to implement. In fact, they do not evaluate the cache pressure mechanism. Third, this mechanism only considers neighbouring banks, i.e., banks at one-hop distance. Finally, as far as we know, neither parallel nor multi-programmed workloads have been evaluated using this technique. In contrast, in our proposal pages are

distributed among all banks, if necessary, in an easy way and without requiring any extra hardware. Moreover we present results for either parallel and multi-programmed workloads. On the other hand, they do not care about the issue of the private cache indexing since they use 16KB 4-way L1 caches, in which the number of bits used to index them is smaller than the number of bits of the off-set of the 8KB pages considered in that work, and they can use virtually indexed L1 caches.

Lin *et al.* [67] applied Cho and Jin's proposal to a real system. They studied the dynamic migration of pages and the high overheads that it causes. Recently, Awasthi *et al.* [14] and Chaudhuri [30] proposed several mechanisms for page migration that reduce the overhead of migration at the cost of requiring extra hardware structures. Unfortunately, since migration of pages entails an inherent cost (e.g., flushing caches or TLBs), this mechanism cannot be performed frequently. Although migration can be used along with our proposal, this chapter focuses on the initial mapping of pages to cache banks. Finally, Awasthi *et al.* do not consider the private cache indexing issue because they use small caches that can be virtually indexed, and Chaudhuri do not take care about the indexing bits despite one bit matches with the home offset bits.

7.3 Distance-aware round-robin mapping

In this chapter, we propose distance-aware round-robin mapping, a simple OS-managed mapping policy for many-core CMPs that assigns memory pages to NUCA cache banks. This policy minimizes the total number of off-chip accesses as happens with a round-robin mapping, and reduces the access latency to a NUCA cache (the L2 cache level) as a first-touch policy does. Moreover, this policy addresses this trade-off without requiring any extra hardware support.

In the proposed mechanism, the OS starts assigning physical addresses to the requested pages according to a first-touch policy, i.e, the physical address chosen by the OS maps to the tile of the core that is requesting the page. The OS stores a counter for each cache bank which is increased whenever a new physical page is assigned to this bank. In this way, banks with more physical pages assigned to them will have higher value for the counter.

To minimize the amount of off-chip accesses we define an upper bound on the deviation of the distribution of pages among cache banks. This upper bound can be controlled by the OS through a threshold value. In this way, in case that the counter of the bank where a page should map following a first-touch policy

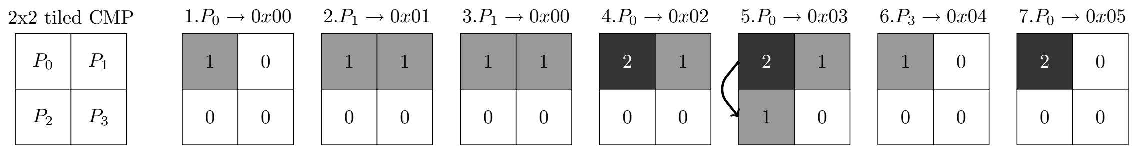


Figure 7.3: Behavior of the distance-aware round-robin mapping policy.

has reached the threshold value, the page is assigned to another bank. The algorithm starts checking the counters of the banks at one hop from the initial placement. The bank with smaller value is chosen. Otherwise, if all banks at one hop have reached the threshold value, then the banks at a distance of two hops are checked. This algorithm iterates until a bank whose value is under the threshold is found. The policy ensures that at least one of the banks has always a value smaller than the threshold value by decreasing by one unit all counters when all of them have values different than zero.

Figure 7.3 shows, from left to right, the behavior of this mapping policy for a 2×2 tiled CMP with a threshold value of two. First, processor P_0 accesses a block within page $0x00$ which faults in memory (1). Therefore, a physical address that maps to the bank 0 is chosen for the address translation of the page, and the value for the bank 0 is increased. Then, processor P_1 perform the same operation for page $0x01$ (2). When processor P_1 accesses page $0x00$ no action is required for our policy because there is a hit in the page table (3). The next access of processor P_0 is for a new page, which is also stored in bank 0, which reaches the threshold value (4). Then, if processor P_0 accesses a new page again, this page must be allocated in another bank (5). The closer bank with a smaller value is bank 2. Finally, when processor P_3 accesses a new page, the page is assigned to its local bank and all counters are decreased (6), allowing bank 0 to map a new page again (7).

The threshold defines the behavior of our policy. A threshold value of zero denotes a round-robin policy in which a uniform distribution of pages is guaranteed, while an unlimited threshold implies a first-touch policy. Therefore, with a small threshold value, our policy reduces the number of off-chip accesses. Otherwise, if the threshold value is high, our policy reduces the average latency of the accesses to the NUCA cache. Note that the threshold value serves as a proxy approximation for the cache pressure since the actual pressure does not directly depend on the uniform distribution of pages, but on the utilization of blocks within pages. However, pages are distributed among all the cache banks, thus

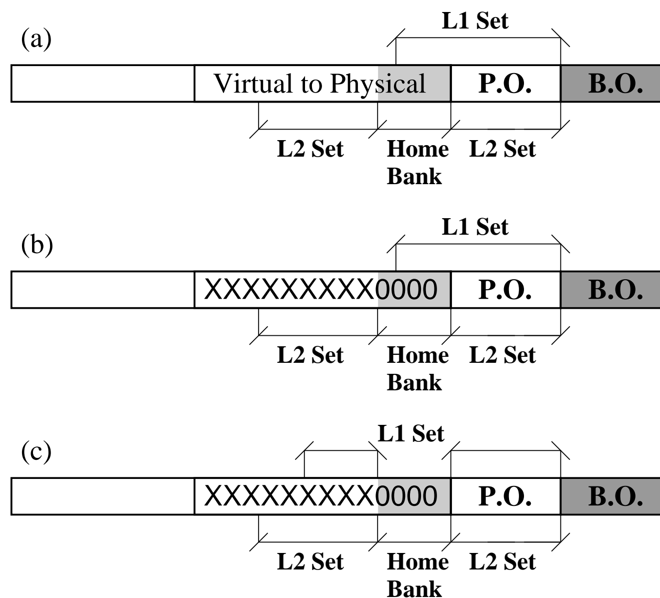


Figure 7.4: Changes in the L1 cache indexing policy.

performing an efficient use of the shared cache. Although, the OS could choose different thresholds depending on the workload, we have found that values between 64 and 256 work well for the workloads considered in this chapter.

7.4 First-touch mapping and private cache indexing

In this section, we study how OS-managed mapping can hurt the hit rate of private L1 caches, mainly when a first-touch policy is implemented. Figure 7.4(a) shows the cache mapping and indexing used in an OS-managed mapping policy. As mentioned in Section 7.2.1, it is important to choose the smallest granularity (the less significant bits of the virtual to physical field), to achieve shorter average distance to the home bank. On the other hand, the bits used to index the private caches, i.e, to select the set for a particular block, are commonly the less significant bits of the block address. When the number of bits used to index the L1 cache is greater than the number of bits of the page offset two main issues appear. First, no virtual indexing can be used to accelerate the L1 cache access [47]. Second, the L1 hit rate can be reduced as consequence of the changes in the assignment of physical addresses.

A first-touch mapping policy tries to map blocks frequently requested by a processor to its closest (local) cache bank. This is carried out by the OS when it assigns the physical address to the requested page (e.g., Figure 7.4(b) represents a physical address that maps to the bank 0). Therefore, most of the blocks that the processor's private L1 cache holds have these bits with the same value. If some of the bits used for selecting the home tile are also used for indexing private L1 caches (Figure 7.4(b)), most of the blocks will map to a specific range of the L1 cache sets, while other sets will remain under-utilized. This factor increases the number of conflict misses in the L1 cache.

This problem also arise with the mapping policy proposed in this chapter. The closer our policy is to a first-touch policy (high threshold value) the more set imbalance will occur in private caches. Therefore, we propose to avoid the bits used to define the home tile when indexing private caches, as shown in Figure 7.4(c). This change allows for better utilization of the private L1 caches, which in turn results in higher L1 cache hit rates, as we show in Section 7.5.1.

7.5 Evaluation results and analysis

For the evaluation of the approaches presented in this chapter, we have implemented new cache mapping and indexing policies in GEMS. Particularly, we have implemented the three OS-managed policies evaluated in this chapter. The first one, named as *RoundRobin*, is an OS-managed policy that assigns physical pages in a round-robin fashion to guarantee the uniform distribution of pages among cache banks. Therefore, this policy does not take into consideration the distance to the home bank. The second one, named as *FirstTouch*, maps memory pages to the local cache bank of the first processor that requested the page. Although this policy is distance-aware, it is not concerned about the pressure on some cache banks. Finally, we also implement the policy proposed in this chapter. We simulate our proposal with threshold values ranging from 2^0 to 2^{10}. We name our policy as *DARR-T* (from Distance-Aware Round-Robin), where *T* is the threshold value employed by the OS. On the other hand, we have implemented a cache indexing scheme that skips the bits employed for identifying the home bank, as explained in Section 7.4. Moreover, we have simulated a 3D stacking memory organization where the off-chip memory has the same interleaving as the L2 cache (i.e., page-grained), thus avoiding the *horizontal* traffic that could appear when off-chip accesses take place, as discussed in Section 7.2.1.

Apart from the evaluation of parallel applications, we also show results for

multi-programmed workloads, since the proposal that we present in this chapter is more appropriate for this context. These workloads are *Ocean4*, *Radix4*, *Mix4*, and *Mix8*, and have been previously described in Chapter 3. We classify our workloads as either *homogeneous* or *heterogeneous*. Homogeneous workloads distribute uniformly memory pages among cache banks when a first-touch policy is employed. In contrast, in heterogeneous workloads a few banks allocate more pages than the others considering a first-touch policy. Particularly, we have simulated two homogeneous (*Ocean4* and *Radix4*) and two heterogeneous (*Mix4* and *Mix8*) workloads. The heterogeneous workloads represent the common scenario in systems employed for throughput computing. For the evaluation of these workloads we have employed the parameters described in Chapter 3. However, since we have doubled the number of cores for simulating a 32-tile CMP, we have also halved cache sizes in order to have the same aggregate capacity. Therefore, we simulate 64KB 4-way associative L1 caches and 512KB 8-way associative L2 caches. However, the access latencies are kept unchanged.

On the other hand, we have also shrunk the simulation parameters for the evaluation of the parallel applications, since, in general, the working set of the scientific benchmarks is small. In particular, the size of the L1 cache is 32KB and it is 2-way associative, while the size of each bank of the L2 cache is 128KB and it is 4-way associative. Again, the access latencies are kept unchanged.

In order to show the homogeneity or heterogeneity of the working set for the evaluated workloads, we have measured the number of pages that map to each cache bank when a first-touch policy is employed. Figure 7.5 shows this distribution of pages for both parallel applications and multi-programmed workloads. As already commented, both *Ocean4* and *Radix4* represent homogeneous workloads while both *Mix4* and *Mix8* are the heterogeneous ones. Regarding parallel applications, we can see that *Barnes*, *FFT*, *Ocean4*, and *Water-Nsq* have a homogeneous distribution of pages among tiles with a first-touch policy. In contrast, *MPGdec*, *MPGenc*, *Radix*, *Raytrace*, *Unstructured*, and *Volrend* distribute memory pages in a heterogeneous way.

The rest of the section is organized as follows. First, we evaluate the impact of the change in the bits used for indexing private L1 caches, as described in Section 7.4. Then, to understand the improvements obtained by the distance-aware round-robin mapping policy, we study both the average distance to the home cache banks and the number of off-chip accesses, and how a good trade-off in those metrics can reduce the applications execution time. Finally, we study the network traffic required by our proposal since it has serious impact on the energy consumed in the interconnection network as discussed along this thesis.

Barnes			
85	26	27	26
30	45	25	29
36	26	27	34
44	26	28	37

FFT			
73	59	52	54
52	71	51	55
53	55	53	54
60	61	56	57

MPGdec			
23	26	26	16
26	26	20	26
84	104	40	38
16	16	27	26

MPGenc			
46	46	30	50
54	58	48	40
88	48	60	44
226	172	46	184

Ocean			
83	85	69	71
64	66	65	64
67	66	92	65
69	65	66	76

Radix			
202	52	46	49
54	47	52	63
61	55	86	77
102	94	148	134

Raytrace			
82	64	58	22
147	16	52	60
18	100	37	36
8	38	40	56

Unstructured			
88	40	30	22
260	13	19	16
45	14	15	16
17	13	15	30

Volrend			
52	6	1	6
11	27	4	23
8	18	12	21
1	1	6	9

Water-NSQ			
31	22	22	22
24	25	25	23
21	24	21	24
29	39	31	52

(a) Parallel applications (16-tiled CMP).

Ocean4							
516	468	478	471	475	469	468	469
468	469	469	491	482	470	469	490
481	469	469	469	483	469	469	470
468	487	469	489	468	469	469	490

Radix4							
494	555	494	547	491	549	488	567
527	560	527	562	525	562	520	559
493	543	493	549	499	552	495	553
522	565	521	563	527	548	518	562

Mix4							
659	469	468	469	126	108	86	57
469	469	468	506	123	45	61	85
0	17	18	28	55	48	43	44
29	26	29	87	55	49	43	81

Mix8							
1134	921	126	270	927	922	125	263
921	948	105	188	937	937	108	187
0	38	73	75	0	34	80	75
40	87	79	113	39	89	78	111

(b) Multi-programmed workloads (32-tiled CMP).

Figure 7.5: Number of pages mapped to each cache bank in a first-touch policy.

7.5.1 Private cache indexing and miss rate

As discussed in Section 7.4, an OS-managed mapping policy that tries to reduce the distance to the home bank can increase the miss rate of private L1 caches. In this section, we study this issue and compare the traditional indexing method with the proposed one. Figures 7.6 and 7.7 show the L1 miss rate for both the parallel applications and multi-programmed workloads evaluated in this chapter, and two indexing methods: the traditional method, that we call *less significant bits*, and the proposed one, named as *skip home bits*. Moreover, the

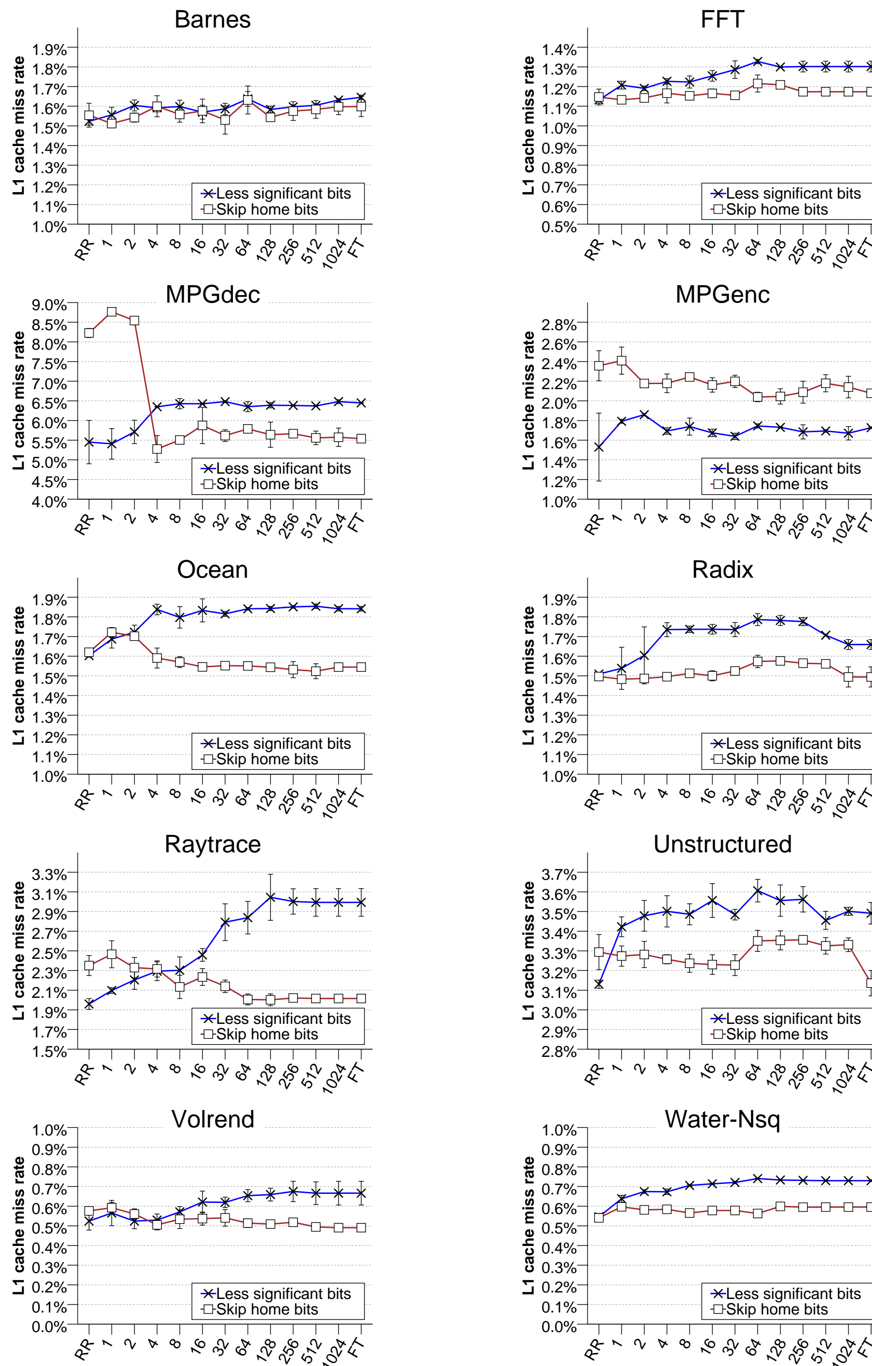


Figure 7.6: Impact of the changes in the indexing of the private L1 caches on parallel applications.

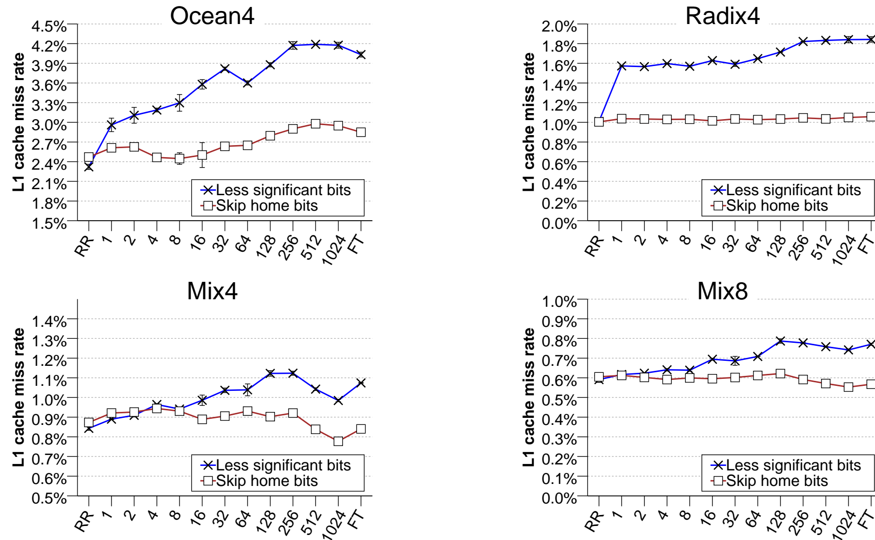


Figure 7.7: Impact of the changes in the indexing of the private L1 caches on multi-programmed workloads.

miss rate is shown for a range of threshold values for our policy, from 0, i.e., a round-robin (*RR*) policy, to unlimited threshold, i.e., a first-touch (*FT*) policy.

First, we can see that when the mapping policy tries to guarantee a uniform distribution of pages (round-robin), the hit rate of the indexing scheme only depends on the locality in the memory accesses of each workload. In this way, some applications, such as *MPGdec*, *MPGenc*, *Raytrace*, and *Unstructured*, have a better hit rate for the *less significant bits* indexing scheme when a round-robin mapping is considered. The remaining applications have similar miss rate for both indexing schemes.

However, when the distance-aware mechanism is more aggressive the *less significant bits* indexing scheme has worse hit rate than the *skip home bits* indexing scheme. In general, for all the workloads the trend of the *less significant bits* indexing scheme is to increase the L1 cache miss rate when the mapping policy transitions from a round-robin policy to a first-touch one. Differently, with the *skip home bits* indexing scheme the miss rate not only do not increase, but also can decrease, as significantly happens in *MPGdec* and *Raytrace*.

Therefore, for the rest of the evaluation we use the *less significant bits* indexing scheme for the round-robin policies and the *skip home bits* indexing scheme for

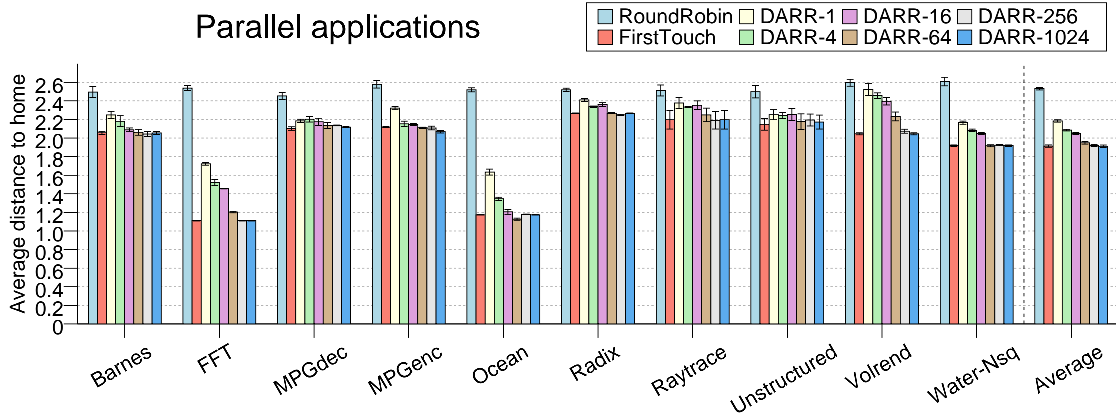
the first-touch and the proposed policy, which are the best schemes for each configuration, on average.

7.5.2 Average distance to the home banks

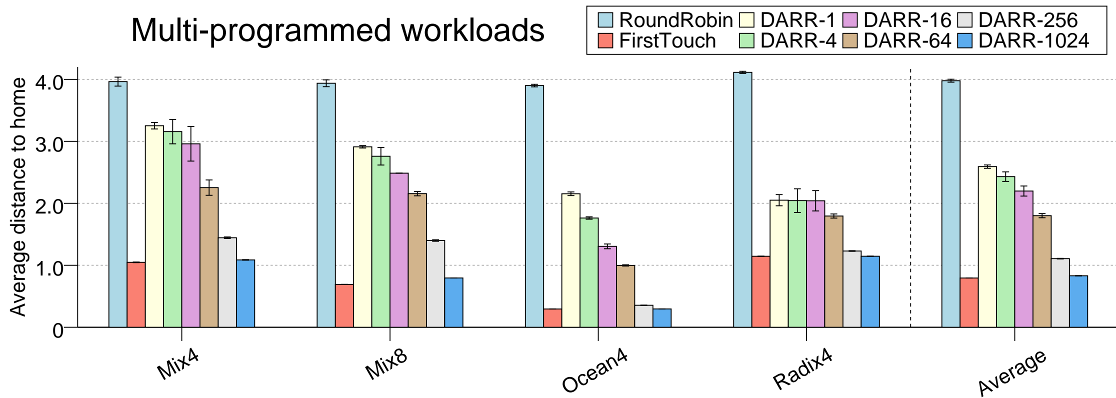
Figure 7.8 plots the average distance in terms of network hops between the tile where the miss takes place and the tile where the home L2 bank is placed. As discussed, a round-robin policy does not care about this distance and, therefore, the average distance for these policies matches up with the average number of hops in a two-dimensional mesh, which can be calculated following the formula $\frac{(x+1)(x-1)}{3x} + \frac{(y+1)(y-1)}{3y}$, where x and y define the size of a $x \times y$ mesh. Therefore, the average number of hops is 2.5 in a 4×4 mesh and 3.875 in a 4×8 mesh. On the other hand, the first-touch policy is the one that requires less hops to solve a miss (1.91 for parallel applications and 0.79 for multi-programmed workloads). As can be observed, the results obtained by our policy always lie between those of the round-robin and first-touch schemes.

Most parallel applications do not obtain representative reductions in the average distance, even when a first-touch policy is considered. For example, *Barnes*, *MPGdec*, *MPGenc*, *Radix*, *Raytrace*, *Unstructured* and *Volrend* need more that two hops to reach the home tile on average. This is because the blocks that frequently cause a cache miss are widely shared by the cores in the system. Other applications in which most of the misses are for blocks with a small number of sharers, like *FFT* and *Ocean*, obtain significant reductions in the average distance with a first-touch policy. On the other hand, we can observe that the multi-programmed workloads always achieve important reductions in the average distance. Even when all the applications running in the system are instances of *Radix*, which does not offer reductions in the parallel case, as happens in *Radix4*. This is because data is only shared in the region of the chip running each instance.

Finally, it is important to note that a threshold value of one for our policy reduces the average distance compared to round-robin (by 15% for parallel and 35% for multi-programmed workloads), and also guarantees a uniform distribution of pages. The higher threshold value is employed, the more reductions in the average number of hops are achieved by our proposal.



(a) Parallel applications (16-tiled CMP).

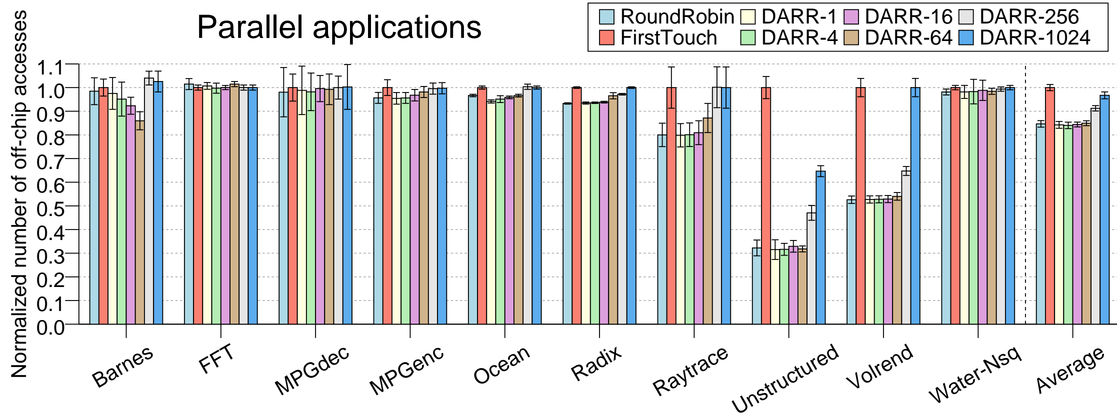


(b) Multi-programmed workloads (32-tiled CMP).

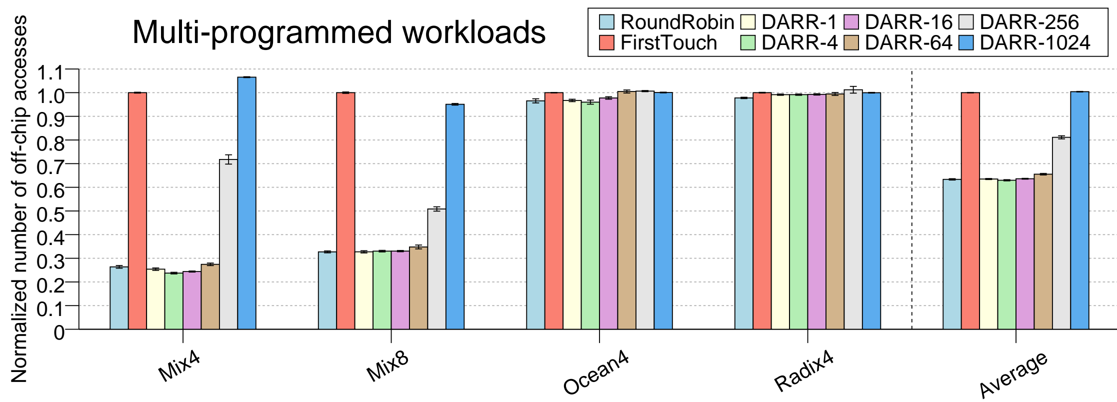
Figure 7.8: Average distance between requestor and home tile..

7.5.3 Number of off-chip accesses

The main issue of the first-touch policy is that it incurs in more off-chip accesses specially for workloads that have unbalanced working sets. Figure 7.9 shows the number of off-chip accesses for the policies evaluated in this chapter normalized with respect to the first-touch policy. We can observe that for homogeneous workloads the difference in the number of off-chip accesses is minimal. On the other hand, when the working set is not well balanced among the pro-



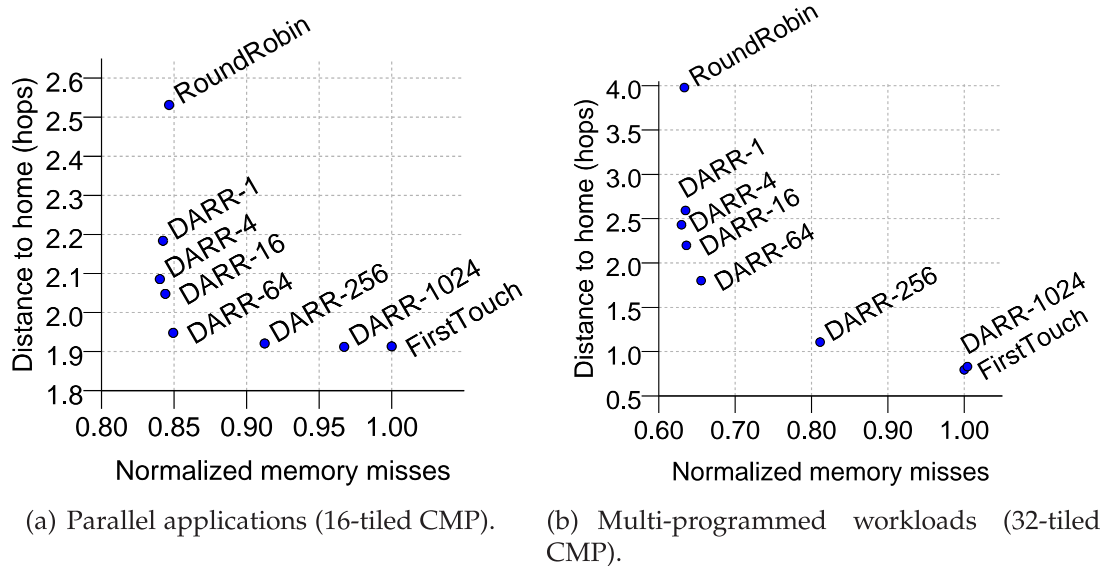
(a) Parallel applications (16-tiled CMP).



(b) Multi-programmed workloads (32-tiled CMP).

Figure 7.9: Normalized number of off-chip accesses.

cessors, the first-touch policy severely increases the number of off-chip accesses. This increment mainly happens in *Unstructured* ($\approx \times 3$), *Volrend* ($\approx \times 2$), *Mix4* ($\approx \times 4$) and *Mix8* ($\approx \times 3$). Note that servers usually run a heterogeneous set of applications, like *Mix4* and *Mix8*. Although, for example, *Radix* has also a heterogeneous distribution of pages the first-touch policy does not significantly increase the number of off-chip accesses compared to round-robin. This is because its working set is larger than the aggregate L2 cache and, therefore, even when a round-robin policy is used, the number of off-chip misses is high.



(a) Parallel applications (16-tiled CMP).

(b) Multi-programmed workloads (32-tiled CMP).

Figure 7.10: Trade-off between distance to home and off-chip accesses.

Regarding the threshold value of our policy, we can observe that with a value smaller than 256 the number of off-chip accesses is kept very close to the round-robin policy. Finally, when the value is very high, the behavior is close to the first-touch policy and the number of off-chip accesses becomes prohibitive.

7.5.4 Trade-off between distance to home and off-chip accesses

In this section we summarize the average results plotted in the two previous sections by showing how our distance-aware round-robin mapping policy can achieve a good trade-off between distance to the home tile and the number of off-chip accesses. Figure 7.10 plots this trade-off both for parallel applications and for multi-programmed workloads.

As we can observe, when our policy uses a small threshold value its behavior is close to a *round-robin* mapping policy. On the other hand, higher values lead to a behavior close to a *first-touch* mapping policy. Values ranging between 64 and 256 achieve a good trade-off between a *round-robin* and a *first-touch* policy. Although better trade-off is obtained for parallel applications, the advantages of the *first-touch* policy for these workloads is smaller than for multi-programmed

ones. Note that the average number of hops is just reduced by 25% (from 2.53 to 1.91) for parallel applications, while it is reduced by 80% (from 3.97 to 0.79) for multi-programmed workloads. The reduction is less significant for parallel applications due to the high sharing degree. Since all cores access most of the blocks, it does not matter too much where they are mapped. Consequently, greater improvements will be achieved for multi-programmed workloads. The values for what the best trade-off is achieved (from 64 to 256), are a few times greater that the number of memory pages that can fit in an L2 cache bank[2].

7.5.5 Execution time

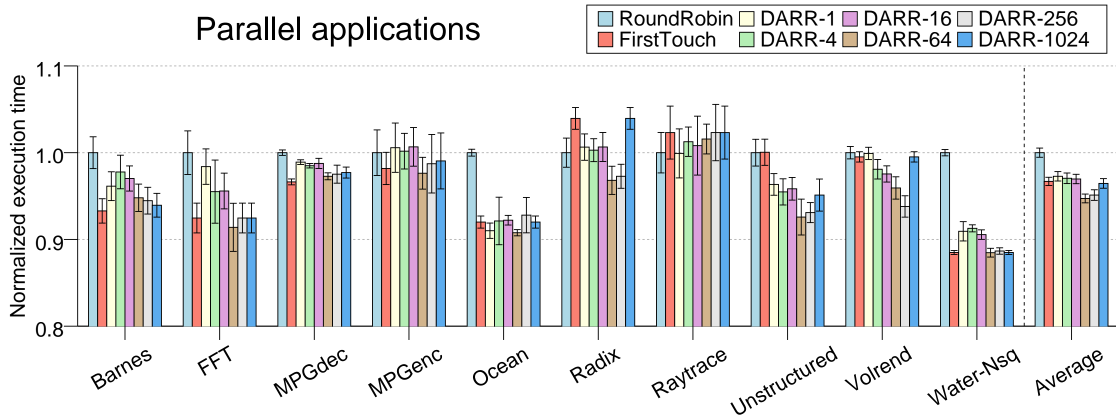
The results discussed in the previous subsections show that our distance-aware round-robin mapping policy is able to achieve a good trade-off between short distance to the home bank and balanced mapping of memory pages. This achievement results in improvements in execution time as Figure 7.11 shows.

As we can observe, the first-touch policy achieves important improvements compared to a round-robin policy when the working set accessed by the different cores is homogeneous, as happens in *Barnes*, *FFT*, *Ocean*, *Water-Nsq*, *Ocean4* and *Radix4*. In contrast, when the distribution of the pages accessed by each core is heterogeneous, as occurs in *Radix*, *Raytrace*, *Mix4* and *Mix8*, the first-touch policy falls into more off-chip accesses, thus degrading performance. In contrast, our proposal achieves the best of a round-robin policy and a first-touch policy with a threshold value between 64 and 256. In this way, we obtain improvements of 5% on average for parallel applications and of 14% on average for multi-programmed workloads compared to a round-robin policy with page-sized granularity. When compared to a first-touch policy we obtain improvements of 2% for parallel applications and 7% for multi-programmed workloads, but additionally avoiding the performance degradation incurred by the first-touch policy in some cases. As previously discussed, the results are better for multi-programmed workloads because more reductions in term of average distance to the home tile can be expected, as consequence of the lack of sharing among program instances.

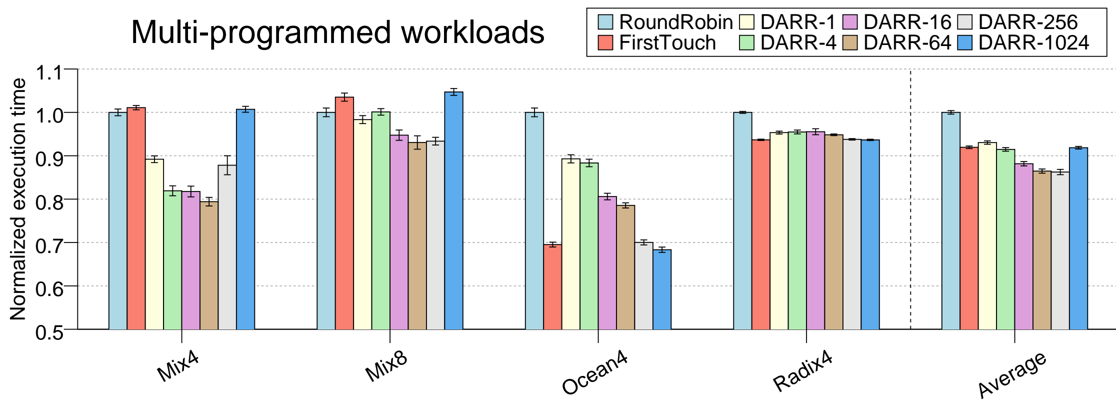
7.5.6 Network traffic

Figure 7.12 compares the network traffic generated by the policies considered in this chapter. In particular, each bar plots the number of bytes transmitted

[2]As described in Chapter 3, the page size is 4BK



(a) Parallel applications (16-tiled CMP).

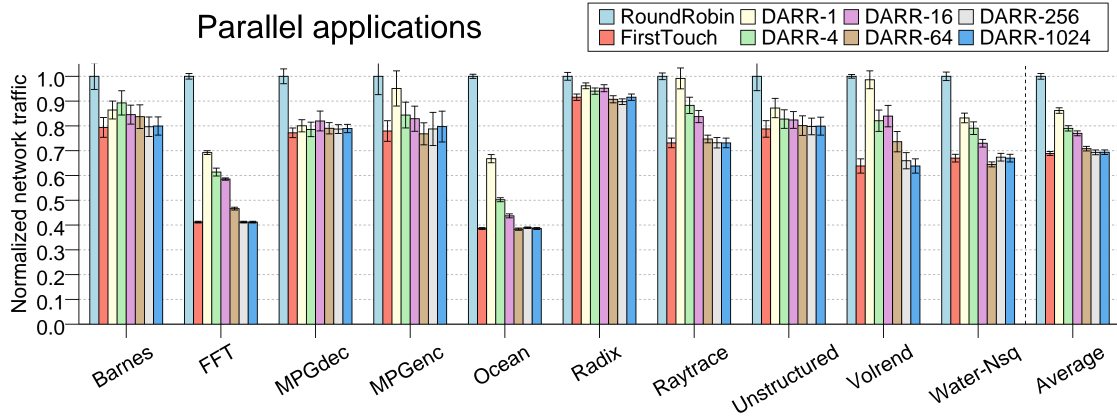


(b) Multi-programmed workloads (32-tiled CMP).

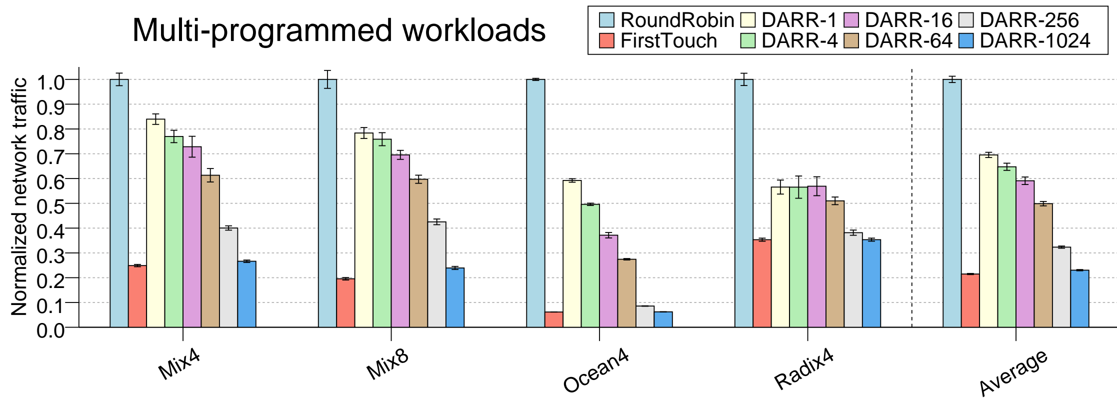
Figure 7.11: Normalized execution time.

through the interconnection network (the total number of bytes transmitted by all the switches of the interconnect) normalized with respect to the *RoundRobin* policy. We can see that the round-robin policy leads to the highest traffic levels because the distance to the home bank is not taken into consideration.

On the other hand, network traffic can be tremendously reduced when a first-touch policy is implemented. In parallel applications, network traffic is reduced by 34% on average. For multi-programmed workloads the savings are greater (79% on average), since most of the blocks are only accessed by cores placed



(a) Parallel applications (16-tiled CMP).



(b) Multi-programmed workloads (32-tiled CMP).

Figure 7.12: Normalized network traffic.

in a small region of the chip. The distance-aware round-robin policy proposed in this chapter obtains reductions in network traffic compared to the round-robin policy, even when the threshold value is just one. When the threshold value increases, the memory-demanding cores can allocate more pages in its local bank and, therefore, less network traffic is generated. As discussed in the previous subsection, a threshold value between 64 and 256 achieves an optimal compromise between round-robin and first-touch. Now, we can see that with a threshold of 256 the network traffic generated by our proposal is reduced by 31%

for parallel applications and 68% for multi-programmed workloads. Obviously, the first-touch policy introduces less traffic than our proposal (3% on average for parallel applications and 34% on average for multi-programmed workloads), at the cost of increasing the number of off-chip accesses.

7.6 Conclusions

In CMP architectures, memory blocks are commonly assigned to the banks of a NUCA cache by following a physical mapping policy in which the home tile of a block is given by a set of bits in the block address. This mapping assigns blocks to cache banks in a round-robin fashion, thus neglecting the distance between the requesting cores and the home NUCA bank for the requested blocks. This issue impacts both cache access latency and the amount of on-chip network traffic generated, and can become a performance problem for large-scale tiled CMPs. On the other hand, first-touch mapping policies, which take into account distance, can lead to an unbalanced utilization of cache banks, and consequently, to an increased number of expensive off-chip accesses.

In this chapter, we propose the *distance-aware round-robin* mapping policy, an OS-managed policy which addresses the trade-off between cache access latency and number of off-chip accesses. Our policy tries to map the pages accessed by a core to its closest (local) bank, like in a first-touch policy. However, we also introduce an upper bound on the deviation of the distribution of memory pages among cache banks, which lessens the number of off-chip accesses. This upper bound can be controlled by a threshold. We have observed that our proposal achieves a good compromise between a round-robin and a first-touch policy with a threshold ranging between 64 and 256 .

We also show that the private cache indexing commonly used in CMP architectures is not the most appropriate for OS-managed distance-aware mapping policies like a first-touch policy or our policy. When the bits used for selecting the home bank are also used for indexing the private L1 cache, the miss rate of private caches can increase significantly. Therefore, we propose to reduce the miss rate by skipping these bits when private L1 caches are indexed.

Our proposal obtains average improvements of 5% for parallel applications and of 14% for multi-programmed workloads compared to a round-robin policy with page granularity (better improvements are obtained compared to a policy that uses block granularity). In terms of network traffic, our proposal obtains average improvements of 31% for parallel applications and 68% for multi-

programmed workloads. When compared to a first-touch policy we obtain average improvements of 2% for parallel applications and 7% for multi-programmed workloads, slightly increasing on-chip network traffic. Finally, one of the main assets of our proposal is its simplicity, because it does not require any extra hardware structure, differently from other previously proposed mechanisms. This characteristic makes our proposal easier to implement in current tiled CMPs than the previous approaches.

Chapter 8

Conclusions and Future Directions

8.1 Conclusions

Tiled CMP architectures (i.e., arrays of replicated tiles connected over a switched direct network) have recently emerged as a scalable alternative to current small-scale CMP designs, and will be probably the architecture of choice for future many-core CMPs. On the other hand, although a great deal of attention was devoted to scalable cache coherence protocols in the last decades in the context of shared-memory multiprocessors, the technological parameters and power constrains entailed by CMPs demand new solutions to the cache coherence problem.

Nowadays, directory-based protocols constitute the best alternative to keep cache coherence in large-scale tiled CMPs, since they can work over unordered interconnects and they can also scale up to a larger number of cores than the solutions based on broadcast. Nevertheless, directory protocols have two important issues that prevent them from achieving better scalability: the directory memory overhead and the indirection problem.

The memory overhead of a directory protocol mainly comes from the structures required to store the sharing (or directory) information. If this information is kept by using a full-map sharing code, the total amount of memory that is required increases linearly with the number of cores in the system, which is not admissible for many-core CMPs.

The indirection problem appears as consequence of the access to the directory information, stored at the home tile of each block. Once the directory information is accessed, the corresponding coherence actions can be performed.

These actions usually imply forwarding requests to the tile that must provide the data block, and invalidating all copies of the requested block in case of write misses. Both actions entail two hops (request and response). Since a previous hop is introduced by the access to the directory information, some misses are solved requiring three hops in the critical path of the miss, which results in long cache miss latencies and increases applications' execution time compared to broadcast-based protocols.

On the other hand, tiled CMPs that share the last-level on-chip cache incur in long cache accesses. Since this cache is physically distributed among the tiles in the CMP, the time needed to access a particular cache bank depends on the tile wherein the bank resides (NUCA cache). Commodity systems that implement a distributed shared cache usually map memory blocks to cache banks in a round-robin fashion by using a physical mapping policy that takes some bits of the block address. Unfortunately this policy does not care about the distance between requesting cores and home cache banks which results in long cache access latencies that hurt applications' performance.

Our efforts in this thesis have focused on these three key issues. In particular, we have proposed, implemented and evaluated different techniques aimed at reducing the memory overhead introduced by the cache coherence protocol, the number of cache misses suffering from indirection, and the long cache access latency that characterize NUCA caches. Additionally, in our proposals, we also care about the amount of network traffic generated by the cache coherence protocol, since it is expected to have great impact on the total power consumption of the CMP. Important conclusions can be extracted from the results that have been obtained in this thesis, as we detail in the next paragraphs.

The first conclusion derived from this thesis is that a scalable organization for a distributed directory can be achieved without hurting applications' performance. This scalable organization, that have been presented in Chapter 4, keeps coherence information as duplicate tags and it uses a granularity of interleaving such as the bits that specify the tile where a block maps (i.e., the home tile) must be a subset of the bits used to index private caches, i.e., to map blocks to a particular set ($bits_home \subseteq bits_private_cache_set$). Since the bits commonly used to index private caches are the less significant ones, we also use these bits for the mapping to home tiles, leading to a block-grained interleaving.

The area employed by this scalable directory represents only 0.53% of the area required for storing data blocks. Moreover, this percentage keeps constant for systems where the number of sets in the private caches is greater or equal than the number of tiles ($num_tiles \leq num_private_cache_sets$). Since it seems

that this rule will be fulfilled for next generation CMPs, the proposed scalable directory organization could be used to avoid the memory overhead problem of directory-based protocols.

The size in bits of each directory bank in this directory organization is $c \times (l_t + 2)$, where c is the number of entries of the private cache and l_t is the size of the tag field. Since this size keeps unchanged meanwhile the previous rule is fulfilled, the same building blocks or tiles could be used for CMP configurations with different number of tiles, thus making easier the design of tiled CMPs.

Furthermore, the mapping employed for this directory organization ensures that the requested blocks and their consequent replacements map to the same home tile. This characteristic allows the cache coherence protocol to merge replacement with request messages, thus saving network traffic. Hence, we have also presented a mechanism that removes all traffic due to L1 cache replacements, which we call the *implicit replacements* mechanism. We have implemented and evaluated a cache coherence protocol that includes a scalable directory organization and performs replacements in a implicit way. We have shown through our simulation tools that a significant fraction of network traffic can be saved. Particularly, our protocol leads to average reductions of 13%, and up to 32%, compared to a directory protocol with ideal directory caches, i.e., unlimited caches with a full-map sharing code. If we consider a protocol where evictions of clean blocks are not silent, as happens in some systems that maintain the directory information as duplicate tags, the implicit replacements mechanism can save 33% of coherence messages on average. These reductions in network traffic finally translate into significant savings in power consumption.

The second conclusion that can be extracted from this thesis is that it is possible to design a cache coherence protocol that avoids the indirection problem for most cache misses without relying on broadcasting requests. Chapter 5 presents this cache coherence protocol, which we name as *direct coherence*, and it also offers a detailed description and evaluation of the proposed protocol.

Direct coherence protocols meet the advantages of both directory and token protocols and avoid their problems. This is achieved by changing the tasks performed for each component involved in a cache miss. In this way, the task of storing the sharing information and ensuring ordered accesses for every memory block is assigned to the tile that provides the block on a cache miss, i.e., the owner tile in a MOESI protocol. In contrast, directory protocols assign this task to the home tile. On the other hand, the task of keeping the identity of the owner tile is assigned to the home tile, but also to the requesting tiles and, in this way, indirection can be avoided by directly sending the requests to the

owner tile instead of to the home one. Moreover, only one message is sent by the requesting tile, differently from *Token* which broadcasts requests on every cache miss.

Several implementations of this protocol for tiled CMPs have been explained and evaluated in this chapter. These implementations differ in how requesting tiles obtain the information about the identity of owner tiles. This information can be decisive to obtain good owner predictions and, therefore, to successfully avoid the indirection. In the first implementation, requesting tiles obtain this information from both invalidation and request messages. Since we find that this information is not enough to obtain accurate owner predictions for some applications, we have designed a *hints mechanism* that sends extra control messages, called *hints*, to inform about the identity of the owner tile on every ownership change. We have proposed two techniques to implement the hints mechanism. The first one, called *frequent sharers*, introduced more storage requirements since it adds a bit-vector to each L1 cache entry, but slightly reduces the traffic generated by hint messages. The second one, named as *address signatures*, considerably reduces the area required by the hints mechanism by using bloom filters, but increases network traffic due to the appearance of false positives.

The results given by our simulation infrastructure have shown that the direct coherence protocol that uses a hints mechanism based on address signatures (*DiCo-Hints AS*) is the best option in terms of average execution time. While in a directory protocol the average percentage of misses with indirection is 56%, this percentage goes down up to 18% on average in *DiCo-Hints AS*. This reduction in the percentage of cache misses suffering from indirection results in savings of 14% in the average cache miss latency, which finally leads to 9% of improvements in execution time, compared to a directory protocol. Although *DiCo-Hints AS* slightly increases traffic requirements compared to a directory protocol, it considerably reduces traffic compared to *Token* (37%) due to the lack of broadcast in direct coherence protocols. Therefore, the total power consumed by the interconnection network is also reduced compared to *Token*. The reduction in network traffic compared to *Token* also reduces the contention at the interconnection network which accelerates cache misses, resulting in improvements of 8% in terms of execution time.

We also have studied the size and complexity of the structures required by direct coherence protocols, which are comparable to those involved in a directory protocol. However, both the directory and the direct coherence implementations evaluated in that chapter track sharers by means of a full-map sharing code. As previously discussed, a scalable directory organization can be achieved for

directory protocols by following a particular interleaving. Unfortunately, this organization is only applicable when the coherence information is stored at the home tile, but it is not applicable to the sharing information stored along with the owner block in direct coherence protocols.

Hence, we have discussed how to reduce the memory overhead of direct coherence protocols in Chapter 6. Particularly, we have addressed this problem by using compressed sharing codes. Since compressed sharing codes introduce extra messages into the interconnection network due to their loss of accuracy, we care about finding a good trade-off between area requirements and network traffic. The base direct coherence protocol that we have employed for this analysis is *DiCo-Hints AS*, which has been shown to perform better than the other alternatives evaluated previously.

We have simulated several implementations of direct coherence protocols with different compressed sharing codes: *coarse vector*, *limited pointers*, and *binary tree*. Additionally, we have also evaluated a direct coherence protocol that does not store information about the sharers of each block. Although this last alternative would fall into a significant increase in network traffic for directory protocols (broadcast would be required for most cache misses), in direct coherence protocols, the owner tile of each block is always known (at least by the home tile), which avoids broadcasting requests both for read misses and write misses with just one sharer. This characteristic allows direct coherence protocols to have a good traffic-area trade-off without requiring the modification of data caches.

The evaluation of the different alternatives shows that a good compromise between network traffic and area requirements can be obtained by using direct coherence protocols. Particularly, *DiCo-LP-1*, which is a direct coherence protocols that only uses one limited pointer for the second sharer of a block (the first one is always stored in the home tile), *DiCo-BT*, which codifies the directory information using just three bits that represent a level in a virtual binary tree, and *DiCo-NoSC*, which does not store any coherence information in the data caches, are the alternatives that achieve the best compromise between traffic and area. The memory overhead required by these compressed sharing codes is less or equal than $O(log_2 n)$ and, therefore, the three implementations scale with order $O(log_2 n)$ due to the addition of the coherence caches (L1C$ and L2C$) in direct coherence protocols. These memory requirements represent an admissible overhead for many-core CMPs. The best alternative in terms of area requirements, *DiCo-NoSC*, increases network traffic by 19% compared to a direct coherence

protocol that keeps a full-map sharing code, but still reducing the traffic when compared to *Token* by 25%, and also requiring less area overhead.

Finally, the use of compressed sharing codes does not impact too much on execution time when compared to direct coherence protocols with a full-map sharing code. This is due to the characteristic of always having a pointer identifying the owner tile, which severely reduces the amount of traffic generated by compressed sharing codes. In particular, *DiCo-NoSC* is the one that obtains the greatest increase, but just by 2%. Note that *DiCo-NoSC* still improves execution time compared to *Token* and *Directory* by 6% and 7%, respectively.

Therefore, since most of the alternatives evaluated avoid the indirection problem requiring both low area and traffic requirements, we believe that direct coherence protocols are a viable alternative to current cache coherence protocols for future many-core tiled CMPs. Obviously, the use of each one of the alternatives could depend on the particular system constraints.

The last conclusion that we can draw is that a good compromise between average access latency and cache miss rate can be achieved for NUCA caches without requiring any extra hardware. Basically, there are two main ways of performing the mapping of memory blocks to cache banks in a NUCA cache. The first one, is a physical mapping that obtains the home bank of a block by looking up some bits of the block address. This policy distributes the blocks in a round-robin fashion, thus guaranteeing an efficient utilization of the total NUCA cache. However, it does not consider the distance between requesting and home tiles, which results in long cache access latencies. The other policy is first-touch, which can be easily implemented by the OS. In this policy, memory pages are mapped to the cache bank of the first tile that requested a block belonging to that page, which reduces the average access latency to a NUCA cache at the cost of increasing its miss rate and, therefore, off-chip accesses, when the working set of the application is not well-balanced among tiles. The compromise between short access latency and low number of off-chip accesses has been discussed in Chapter 7, where a new cache mapping policy called *distance-aware round-robin* is presented and evaluated.

The distance-aware round-robin mapping policy is an OS-managed policy that tries to map memory pages to the local NUCA bank of the first core that requests a block belonging to that page, like in a first-touch policy, but also introduces an upper bound on the deviation of the distribution of memory pages among cache banks, which minimizes the number of off-chip accesses. This upper bound is controlled by a threshold value. We have observed that for a threshold value ranging between 64 and 256, our mapping policy achieves

a good compromise between a round-robin and a first-touch mapping policy. Furthermore, the distance-aware round-robin mapping policy does not require any extra hardware, which makes its implementation straightforward.

Nevertheless, one potential issue of distance-aware mapping policies managed by the OS, like first-touch and ours, is that the private cache miss rate can be hurt if its indexing is not performed carefully. This happens when the same bits that define the home bank are used for indexing the private caches. Due to the OS virtual-to-physical translation, most of the blocks in the private caches will have the same bits in the part of the address representing the home tile. If these bits are also used for indexing the private caches, some sets could be overloaded while others could remain almost unused. This imbalance increases conflict misses and, therefore, we have proposed to skip these bits when private caches are indexed in order to reduce their miss rates.

Compared to a round-robin policy with page granularity, which obtains better results than a policy that uses block granularity, the distance-aware round-robin mapping policy combined with the private indexing that skips the bits used for defining the home tile obtains average improvements of 5% for the parallel applications evaluated, and average improvements of 14% for the four multi-programmed workloads that have been created to evaluate this policy. In terms of network traffic, our proposal obtains average improvements of 31% for parallel applications and 68% for multi-programmed workloads. When compared to a first-touch policy we have obtained average improvements of 2% for parallel applications and 7% for multi-programmed workloads, slightly increasing on-chip network traffic. Finally, the results obtained by these approaches and the fact that they do not require any extra hardware support, make this policy a viable alternative for the efficient mapping of large-scale NUCA caches.

In summary, this thesis presents several techniques aimed at reducing the high costs, in terms of both memory overhead, long cache miss latencies and network traffic (and, consequently, power consumption), that can be entailed by traditional cache coherence protocols and many-core CMP organizations.

8.2 Future directions

A natural extension of the work carried out in this thesis could be the combination of some of the proposals presented here to improve the efficiency of the system. For example, direct coherence protocols could use a scalable directory organization for the L2C$, since it is kept at the home tile. This would reduce

the memory requirements of this structure from $O(log_2 n)$ to $O(cte)$. If we apply this organization to *DiCo-NoSC*, which does not keep any sharing information along with the data caches, the total memory overhead of this protocol will be almost $O(cte)$. The only structure whose area still increases as $O(log_2 n)$ is the L1C$. Although this structure needs few cache entries and it does not add too much area overhead, it will be nice to redesign it in a way that its size does not depend on the number of cores in order to achieve full scalability in terms of area.

Another possible combination is the use of direct coherence protocols with a distance-aware round-robin mapping policy. These two proposals can be combined in a easy way. However, since direct coherence tries to avoid the accesses to the home tile, the improvements obtained by the distance-aware round-robin mapping policy would not be as significant as when a directory-based protocol is considered.

The third combination could be the scalable directory organization with the distance-aware round-robin mapping policy. However, these policies are not compatible. This is due to the fact that a scalable directory is achieved when the bits used to map the home tile are also used to index private caches. However, we have also seen that in distance-aware OS-managed mapping policies, it is preferable to avoid these bits for the private cache indexing. In this way, if we decide to implement a scalable directory where the home tiles are chosen by taking care about the distance of requesting tiles, we will incur in high private cache miss rate, since just a few sets in the cache will be used. On the other hand, if we skip these bits from the indexing function, directory scalability is not possible. Therefore, solving this issue represents an interesting future way of research.

On the other hand, direct coherence protocols offer lots of future research lines. For example, it will be interesting to study the combination of direct coherence and heterogeneous interconnection networks [32]. Note that direct coherence increases the number of messages that are not in the critical path of cache misses (e.g., the hint messages) compared with a directory protocol. Since these messages could be sent using low-power wires without hurting performance, direct coherence protocols would make more extensive use of these wires than a directory protocol, thus resulting in lower power consumption.

Another research direction would consist in the study of direct coherence protocols for throughput computing [29], where several program instances run at the same time on the same chip. As we have shown in this thesis, in a directory protocol all misses must access the home tile, but this tile can be located in

any part of the chip, and not only in the region where the application requesting the block is running. This increases cache miss latencies and it also injects extra traffic into regions where other applications are running, thus hurting their performance [80]. However, the owner tile of a block will always be a tile placed in the region were the application requesting the block is running and, therefore, direct coherence protocols can reduce even more cache miss latencies, and they also can avoid traffic interferences. Moreover, hint messages could be only sent to a particular region, thus saving more network traffic.

Other options regarding the combination of direct coherence protocols with other existing techniques and architectures are the study of direct coherence protocols with interconnection networks with a ring topology [79], in the context of hardware transactional memory [48, 84], and for developing fault tolerant cache coherence protocols [40].

Regarding NUCA caches, another future way of research can be the study of NUCA cache replacement policies. An efficient policy could avoid the eviction of memory blocks used by the slower threads in parallel applications (the ones reaching the barrier in last position) in order to accelerate them and, consequently, the overall performance.

Finally, another interesting future work can be the study of the bits of the block address used for indexing private caches. As we have seen, the election of different bits impacts on the cache hit rate. We have found that the bits used commonly for indexing caches (the less significant ones) are not the ones that achieve the best hit rate for all applications. Even the bits that achieve the best hit rate can change across the phases of the same application. Therefore, it will be interesting to design an adaptive mechanism that can decide the group of bits chosen for indexing caches on each program phase.

Appendix A

Direct Coherence Protocol Specification

This appendix shows a detailed specification of the implementation of direct coherence protocols for tiled CMPs. Particularly, we show the specification of the *DiCo-Hints AS* implementation. First, we describe the messages that travel across the interconnection network to keep cache coherence in direct coherence protocols. Table A.1 shows both these messages and a description of them. We also show the type of each message, i.e., if it is a control message or if it also includes a copy of the data block.

Table A.1: Coherence messages.

Message	Type	Description
GETS	Control	Cache miss. Processor asks for read permission
GETX	Control	Cache miss. Processor asks for write permission
INV	Control	Invalidation
CHANGE_OWNER	Control	Notification about change of ownership
ACK	Control	Acknowledgement of an invalidation
ACK_CHOWN	Control	Acknowledgement of change of owner
ACK_STARVED	Control	The pending starved request has been solved. It also informs about change of ownership
DATA	Data	Data, sharers can exist
DATA_EXCLUSIVE	Data	Exclusive data, no other processor has a copy
WRITEBACK_DATA	Data	Replacement. Store the block in the next cache level

Then, we describe the behavior of the cache coherence protocols by defining three memory controllers: the L1 cache controller, the L2 cache controller and the memory controller. These controllers are detailed using a table-based technique [107], which provides clear and concise information about these controllers, but also gives sufficient detail (all race conditions) about the implementation of the protocol. This is also the technique used to define cache coherence protocols in the SLICC language provided by the GEMS simulator.

Each memory controller is defined by four tables: states, events, actions and transitions. The first table defines all the states of the controller, both base and transient states, and it also gives a definition of each state. The second table describes all the events that can take place in the controller. Events are triggered as consequence of incoming processor or coherence messages. Coherence messages have been defined in Table A.1. On the other hand, processor messages are shown in Table A.2.

Table A.2: Processor messages.

Message	Description
Load	Processor asks read permission for a data block
Ifetch	Processor asks read permission for an instruction block
Store	Processor asks write permission for a data block
Atomic	Processor asks write permission for a data block to perform an atomic read-modify-and-write operation

The third table corresponds to the actions carried out by the controller. Actions are encoded by using one or two characters and a short description of each action is also shown in each table. Different actions are performed depending on the transitions. These transitions are shown in the transitions table. This table defines a transition for each state and event. In the first column, all possible states for the corresponding controller are listed. The first row includes all possible events that cause the controller to take the actions and to potentially change the state. The intersection between states and events shown all possible state transitions. All actions performed in each transition are shown in order. At the end off the list of actions the resulting state is shown (if the state changes). The change of state is denoted with the symbol $>$. For example, $h\ x > MM$ means that after performing actions h and x the state will change to MM.

The tables describing the three controllers implemented for direct coherence protocols are shown next. Tables A.3, A.4, A.5, and A.5 define the L1 cache

controller. Tables A.7, A.8, A.9, and A.9 define the L2 cache controller. Finally, Tables A.11, A.12, A.13, and A.13 define the memory controller.

Table A.3: L1 cache controller states.

	State	Description
Base	**I**	Invalid
	S	Shared
	O	Owned, another sharers exist
	M	Modified (dirty)
	MM	Modified (dirty and locally modified)
Transient	**IS**	Issued GetS
	ISI	Issued GetS, Block will not be stored in cache
	IM	Issued GetX
	SM	Issued GetX, we still have an old copy of the line
	OM	Issued GetX, received data
	MW	Modified (dirty)
	MB	Modified (dirty), blocked until ack_chown arrives
	MMW	Modified (dirty and locally modified)
	MMB	Modified (dirty and locally modified), blocked until ack_chown arrives

Table A.4: L1 cache controller events.

	Event	Description
Local request	**Load**	Load request from the local processor
	Ifetch	Instruction fetch request from the local processor
	Store	Store request from the local processor
	L1_Replacement	Replacement from L1 cache
Remote request	**Fwd_GETS**	Load request from another processor
	Fwd_GETX	Store request from another processor
	Inv	Invalidations from the owner tile
	Change_Owner	Notification of change of ownership
Response	**Ack**	Received acknowledgement message
	Data	Received data message with read permission
	Exclusive_Data	Received data message with write permission
	Ack_Chown	Received Ack_Chown message
Trigger	**All_Acks**	Received all required data and message acks
	Ack_Chown_Received	The Ack_Chown message has been previously received
Timeout	**Use_Timeout**	Lockout period ended

Table A.5: L1 cache controller actions.

Action	Description
a	Issue a GETS message to the predicted owner tile
b	Issue a GETX message to the predicted owner tile
c	Forward request to the home tile
ds	Send shared data to the requesting tile
dx	Send exclusive data to the requesting tile with the number of ACKs expected
e	Send invalidations to sharers from the owner tile
f	Send ACK to the requesting tile as response of an invalidation
g	Send CHANGE_OWNER or ACK_STARVED (if the request was starved) to the home tile
gg	Send ACK_STARVED to home tile if the request was starved
h	Notify processor that the access completed
i	Allocate a new entry in the MSHR structure
ii	Deallocate an entry from the MSHR structure
j	Allocate a new entry in the L1 cache
jj	Deallocate an entry from the L1 cache
kk	Deallocate an entry from the L1C$ (if exist)
m	Increase (Data) or decrease (ACK) the number of ACKs expected
n	Set number of ACKs expected in the MSHR
o	Trigger All_acks event if all ACKs have been received
p	Schedule timeout of use meanwhile the block cannot be deallocated
pp	Unset timeout of use
q	Send a writeback message with the data block to the home tile
s	Add a new sharer to the sharing code
t	Clean all sharers from the sharing code
u	Update L1C$ with information collected from the message received
v	Set the MSHR as waiting an ACK_CHOWN
vo	Trigger Ack_chown_received event if the ACK_CHOWN message has been received
vv	Unset the MSHR from waiting an ACK_CHOWN
w	Write the data block to cache
x	Pop message from queue
z	Send the message to the back of the queue

Table A.6: L1 cache controller transitions.

State	Load/Ifetch	Store	L1_Replacement	Fwd_GETS	Fwd_GETX	Inv	Change_Owner	Ack	Data	Exclusive_Data	Ack_Chown	All_Acks	Ack_Chown_ Received	Use_Timeout
	Event													
I	j i a kk x >IS	j i b kk x >IM	jj	c x	c x	f u x	u x							
S	h x	i b kk x >SM	jj >I	c x	c x	f jj u x >I	u x							
O	h x	i n e o t x >OM	q jj kk >I	ds gg s x	u e dx g jj x >I		x				vv x			
M	h x	h x >MM	q jj kk >I	ds gg s x >O	u e dx g jj x >I		x				vv x			
MM	h x	h x	q jj kk >I	u dx g jj x >I	u e dx g jj x >I		x				vv x			
IS	z	z	z	c x	c x	f x >ISI	x		w u h ii x >S	w u v h p ii x >MW	z			
ISI	z	z	z	c x	c x	f x	x		u h ii jj x >S	w u v h p ii x >MW	z			
IM	z	z	z	c x	c x	f x	x	m o x		w u v m o x >OM	z			
SM	h x	z	z	c x	c x	f x >IM	x	m o x		w u v m o x >OM	z			
OM	h x	z	z	z	z		x	m o x			vv x	h ii p x >MMW		
MW	h x	h x >MMW	z	z	z		x				vv x			pp vo >MB
MB	h x	h x >MMB	z	z	z		x				vv x >M		x >M	
MMW	h x	h x	z	z	z		x				vv x			pp vo >MMB
MMB	h x	h x	z	z	z		x				vv x >MM		x >MM	

Table A.7: L2 cache controller states.

	State	Description
Base	**I**	Invalid
	ILO	Invalid, but L1 owner exists and L1 sharers can exist
	O	Owned, local sharers can exist
Transient	**IG**	Blocked due to an off-chip access
	OI	Blocked, doing a writeback to memory, waiting acks from L1 caches
Starvation	**ILOS**	Starvation. Invalid, but L1 owner exists and L1 sharers can exist
	OS	Starvation. Owned, L1 sharers cannot exist

Table A.8: L2 cache controller events.

	Event	Description
Request	**L1_GETS**	Load request from an L1 cache
	L1_GETX	Store request from an L1 cache
	L1_STARVED	Starved request from an L1 cache
	L1_WB_DATA	Writeback from an L1 cache (contains data)
	L1_Change_Owner	L1 owner cache has changed
	L1_AckSt	A starved request has been solved
	L2_Replacement	Replacement from L2 cache
Response	**Ack**	Received an invalidation acknowledgement from an L1 cache
	Data_Exclusive	Received data from memory
Trigger	**All_Acks**	Received all required acknowledgements

Table A.9: L2 cache controller actions.

Action	Description
a	Forward L1 cache request to memory
c	Forward request to the owner tile, obtained from the L2C$
ds	Send shared data to the requesting tile
dx	Send exclusive data to the requesting tile with the number of ACKs expected
e	Send invalidations to all sharers except to the requesting tile
f	Store new owner identity in the L2C$
g	Send Ack_Chown message to the new owner tile (if necessary)
h	Send hints (CHANGE_OWNER message) to all cores (if necessary)
i	Allocate a new entry in the MSHR structure
ii	Deallocate an entry from the MSHR structure
j	Allocate a new entry in the L2 cache
jj	Deallocate an entry from the L2 cache
kk	Deallocate an entry from the L2C$ (if exist)
m	Decrease the number of ACKs expected
n	Set number of ACKs expected in the MSHR
o	Trigger All_acks event if all ACKs have been received
q	Send a writeback message with the data block to memory
r	Record requesting core in the MSHR structure
s	Add a new sharer to the sharing code
w	Write the data block to cache
x	Pop message from queue
z	Send the message to the back of the queue

Table A.10: L2 cache controller transitions.

State	Event: L1_GETS	L1_GETX	L1_STARVED	L1_WB_DATA	L1_Change_Owner	L1_AckSt	L2_Replacement	Ack	Data_Exclusive	All_Acks
I	i r a x >IG	i r a x >IG								
ILO	c x >IG	c x >IG	c x >ILOS	j w kk h x >O	f g h x					
O	ds s kk >O	e dx f h jj x >ILO					i n e jj >OI			
IG	z	z							dx f h ii x >ILO	
OI	z	z						m o x		q ii x >I
ILOS	z	z	c x	j w kk x >OS	i f x >ILOS	g f x >ILO				
OS	ds s kk >O	e dx f h jj x >ILO					z			

Table A.11: Memory controller states.

State	Description
I	Memory can store a staled copy of the block
M	Memory stores a fresh copy of the block

Table A.12: Memory controller events.

Event	Description
GETS	Load request
GETX	Store request
WB_DATA	Update the data block

Table A.13: Memory controller actions.

Action	Description
d	Send data block to the home tile
w	Update the data block in memory (if dirty)
x	Pop message from queue
z	Send the message to the back of the queue

Table A.14: Memory controller transitions.

	Event		
State	**GETS**	**GETX**	**WB_DATA**
I	z	z	w x >M
M	d x >I	d x >I	

Bibliography

[1] Manuel E. Acacio, José González, José M. García, and José Duato. A new scalable directory architecture for large-scale multiprocessors. In *7th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 97–106, January 2001. 0.3, 0.4, 0.4.2, 4.1, 4.2, 6.3, 6.3

[2] Manuel E. Acacio, José González, José M. García, and José Duato. Owner prediction for accelerating cache-to-cache transfer misses in cc-NUMA multiprocessors. In *SC2002 High Performance Networking and Computing*, pages 1–12, November 2002. 5.2

[3] Manuel E. Acacio, José González, José M. García, and José Duato. The use of prediction for accelerating upgrade misses in cc-NUMA multiprocessors. In *11th Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 155–164, September 2002. 5.2

[4] Manuel E. Acacio, José González, José M. García, and José Duato. An architecture for high-performance scalable shared-memory multiprocessors exploiting on-chip integration. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 15(8):755–768, August 2004. 5.2.1

[5] Manuel E. Acacio, José González, José M. García, and José Duato. A two-level directory architecture for highly scalable cc-NUMA multiprocessors. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 16(1):67–79, January 2005. 4.2, 5.2.1

[6] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996. 1.1

[7] Anant Agarwal, Ricardo Bianchini, David Chaiken, David Kranz, John Kubiatowicz, Beng hong Lim, Kenneth Mackenzie, and Donald Yeung.

The MIT Alewife machine: Architecture and performance. In *22nd Int'l Symp. on Computer Architecture (ISCA)*, pages 2–13, May 1995. 4.2

[8] Anant Agarwal, Richard Simoni, John L. Hennessy, and Mark A. Horowitz. An evaluation of directory schemes for cache coherence. In *15th Int'l Symp. on Computer Architecture (ISCA)*, pages 280–289, May 1988. 0.3, 4.1, 4.2, 6.1, 6.3

[9] Niket Agarwal, Li-Shiuan Peh, and Niraj K. Jha. In-Network Snoop Ordering (INSO): Snoopy coherence on unordered interconnects. In *15th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 67–78, February 2009. 5.2

[10] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *27th Int'l Symp. on Computer Architecture (ISCA)*, pages 248–259, June 2000. 0.1, 1, 1.2

[11] Ardsher Ahmed, Pat Conway, Bill Hughes, and Fred Weber. AMD Opteron™ shared-memory MP systems. In *14th HotChips Symp.*, August 2002. 2.4.1

[12] Alaa R. Alameldeen and David A. Wood. Variability in architectural simulations of multi-threaded workloads. In *9th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 7–18, February 2003. 0.2, 3.3

[13] Alaa R. Alameldeen and David A. Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4):8–17, July 2006. 3.3

[14] Manu Awasthi, Kshitij Sudan, Rajeev Balasubramonian, and John Carter. Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In *15th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 250–261, February 2009. 7.2.2

[15] Mani Azimi, Naveen Cherukuri, Doddaballapur N. Jayasimha, Akhilesh Kumar, Partha Kundu, Seungjoon Park, Ioannis Schoinas, and Aniruddha S. Vaidya. Integration challenges and tradeoffs for tera-scale architectures. *Intel Technology Journal*, 11(3):173–184, August 2007. 0.1, 1, 6.1

[16] Luiz A. Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *27th Int'l Symp. on Computer Architecture (ISCA)*, pages 12–14, June 2000. 0.3.1, 2.1, 2.4.3, 4.1, 4.2.1, 4.3.4

[17] Bradford M. Beckmann, Michael R. Marty, and David A. Wood. ASR: Adaptive selective replication for CMP caches. In *39th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 443–454, December 2006. 5.2

[18] Bradford M. Beckmann and David A. Wood. Managing wire delay in large chip-multiprocessor caches. In *37th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 319–330, December 2004. 7.2.2

[19] Shane Bell, Bruce Edwards, John Amann, Rich Conlin, Kevin Joyce and-Vince Leung, John MacKay, Mike Reif, Liewei Bao, John Brown, Matthew Mattina, Chyi-Chang Miao, Carl Ramey, David Wentzlaff, Walker Anderson, Ethan Berger, Nat Fairbanks, Durlov Khan, Froilan Montenegro, Jay Stickney, and John Zook. TILE64™ processor: A 64-core SoC with mesh interconnect. In *IEEE Int'l Solid-State Circuits Conference (ISSCC)*, pages 88–598, January 2008. 4.6.1

[20] Mårten Björkman, Fredrik Dahlgren, and Per Stenström. Using hints to reduce the read miss penalty for flat COMA protocols. In *28th Int'l Conference on System Sciences*, pages 242–251, January 1995. 5.2

[21] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, July 1970. 5.4

[22] Koen De Bosschere, Wayne Luk, Xavier Martorell, Nacho Navarro, Mike O'Boyle, Dionisos Pnevmatikatos, Alex Ramírez, Pascal Sainrat, André Seznec, Per Stenström, and Olivier Temam. High-performance embedded architecture and compilation roadmap. *Transactions on HiPEAC I*, pages 5–29, January 2007. 0.1, 1, 1.1

[23] Jason F. Cantin, James E. Smith, Mikko H. Lipasti, Andreas Moshovos, and Babak Falsafi. Coarse-grain coherence tracking: RegionScout and region coherence arrays. *IEEE Micro*, 26(1):70–79, January 2006. 5.2

[24] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 27(12):1112–1118, December 1978. 1.1, 2.4.3

[25] Luis Ceze, James Tuck, Calin Cascaval, and Josep Torrellas. Bulk disambiguation of speculative threads in multiprocessors. In *33rd Int'l Symp. on Computer Architecture (ISCA)*, pages 227–238, June 2006. 0.4.1.2, 5.2, 5.4

[26] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *4th Int. Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 224–234, April 1991. 0.3, 0.3.3.1, 0.3.3.2, 0.4.2, 4.1, 4.2, 4.2.1, 6.1, 6.3

[27] Jichuan Chang and Gurindar S. Sohi. Cooperative caching for chip multiprocessors. In *33rd Int'l Symp. on Computer Architecture (ISCA)*, pages 264–276, June 2006. 2.3, 5.2

[28] Yeimkuan Chang and Lasimi N. Bliuyan. An efficient hybrid cache coherence protocol for shared memory Multiprocessors. *IEEE Transactions on Computers*, pages 352–360, March 1999. 4.2

[29] Shailender Chaudhry, Paul Caprioli, Sherman Yip, and Marc Tremblay. High-performance throughput computing. *IEEE Micro*, 25(3):32–45, May 2005. 1.1, 3.4.3, 7.1, 8.2

[30] Mainak Chaudhuri. PageNUCA: Selected policies for page-grain locality management in large shared chip-multiprocessor caches. In *15th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 227–238, February 2009. 7.2.2

[31] Liqun Cheng, John B. Carter, and Donglai Dai. An adaptive cache coherence protocol optimized for producer-consumer sharing. In *13th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 328–339, February 2007. 5.2

[32] Liqun Cheng, Naveen Muralimanohar, Karthik Ramani, Rajeev Balasubramonian, and John B. Carter. Interconnect-aware coherence protocols for chip multiprocessors. In *33rd Int'l Symp. on Computer Architecture (ISCA)*, pages 339–351, June 2006. 0.6, 5.2, 5.6.3, 8.2

[33] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *36th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 55–66, December 2003. 7.2.2

[34] Sangyeun Cho and Lei Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *39th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 455–465, December 2006. 0.5, 0.5.2, 4.6.2, 7.1, 7.2.1, 7.2.2

[35] Jong H. Choi and Kyu H. Park. Segment directory enhancing the limited directory cache coherence schemes. In *13th Int'l Parallel and Distributed Processing Symp. (IPDPS)*, pages 258–267, April 1999. 4.2, 6.1

[36] Alan L. Cox and Robert J. Fowler. Adaptive cache coherency for detecting migratory shared data. In *20st Int'l Symp. on Computer Architecture (ISCA)*, pages 98–108, May 1993. 2.3.1

[37] Blas Cuesta. *Efficient Techniques to Provide Scalability for Token-based Cache Coherence Protocols*. PhD thesis, Universidad de Valencia, June 2009. 5.5, 6.4.1

[38] David E. Culler, Jaswinder P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., 1999. 0.4.3, 1.1, 4.2, 6.3

[39] Shamik Das, Andy Fan, Kuan-Neng Chen, Chuan Seng Tan, Nisha Checka, and Rafael Reif. Technology, performance, and computer-aided design of three-dimensional integrated circuits. In *Int'l Symposium on Physical Design*, pages 108–115, April 2004. 7.2.1

[40] Ricardo Fernández-Pascual, José M. García, Manuel E. Acacio, and José Duato. A low overhead fault tolerant coherence protocol for CMP architectures. In *13th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 157–168, February 2007. 8.2

[41] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *17th Int'l Symp. on Computer Architecture (ISCA)*, pages 15–26, June 1990. 1.1

[42] Kourosh Gharachorloo, M. Sharma, S. Steely, and S. Van Doren. Architecture and Design of AlphaServer GS320. In *9th Int. Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 13–24, November 2000. 2.4.3

[43] James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *10th Int'l Symp. on Computer Architecture (ISCA)*, pages 124–131, June 1983. 2.2

[44] Anoop Gupta, Wolf-Dietrich Weber, and Todd C. Mowry. Reducing memory traffic requirements for scalable directory-based cache coherence schemes. In *Int'l Conference on Parallel Processing (ICPP)*, pages 312–321, August 1990. 0.3, 0.3.3.1, 0.3.3.2, 0.4.2, 4.1, 4.2, 4.2.1, 6.1, 6.3

[45] David B. Gustavson. The scalable coherent interface and related standards proyects. *IEEE Micro*, 12(1):10–22, January 1992. 4.2

[46] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, March 2000. 2.1

[47] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 4th edition, 2007. 1, 7.4

[48] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *20st Int'l Symp. on Computer Architecture (ISCA)*, pages 289–300, May 1993. 8.2

[49] Ron Ho, Kenneth W. Mai, and Mark A. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, April 2001. 0.1, 1.2

[50] Hemayet Hossain, Sandhya Dwarkadas, and Michael C. Huang. Improving support for locality and fine-grain sharing in chip multiprocessors. In *17th Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 155–165, October 2008. 5.2

[51] Hung-Chang Hsiao and Chung-Ta King. Boosting the performance of now-based shared memory multiprocessors through directory hints. In *20th IEEE Int'l Conference on Distributed Computing Systems (ICDCS'00)*, pages 602–609, April 2000. 5.2

[52] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. A NUCA substrate for flexible CMP cache sharing. In *19th Int'l Conference on Supercomputing (ICS)*, pages 31–40, June 2005. 0.5, 2.1, 4.2.1, 5.1, 5.2, 7.1

[53] Intel Corporation, White paper. *First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem)*, April 2009. 2.1, 4.6.1, 7.1

[54] Natalie D. Enright Jerger, Li-Shiuan Peh, and Mikko H. Lipasti. Virtual tree coherence: Leveraging regions and in-network multicast tree for scalable cache coherence. In *41th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 35–46, November 2008. 4.2.1, 5.2

[55] Ross E. Johnson. *Extending the Scalable Coherent Interface for Large-Scale Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin-Madison, 1993. 4.2, 6.1

[56] Chetana N. Keltcher, Kevin J. McGrath, Ardsher Ahmed, and Pat Conway. The AMD opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66–76, April 2003. 1.2, 2.1

[57] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *10th Int. Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 211–222, October 2002. 0.1, 1.2, 7.1, 7.2.1, 7.2.2

[58] Jinseok Kong, Pen-Chung Yew, and Gyungho Lee. Minimizing the directory size for large-scale shared-memory multiprocessors. *IEICE Transactions on Information and Systems*, E88-D(11):2533–2543, November 2005. 4.2

[59] Rakesh Kumar, Victor Zyuban, and Dean M. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *32nd Int'l Symp. on Computer Architecture (ISCA)*, pages 408–419, June 2005. 0.1, 1

[60] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark A. Horowitz, Anoop Gupta, Mendel Rosenblum, and John L. Hennessy. The Stanford FLASH multiprocessor. In *21st Int'l Symp. on Computer Architecture (ISCA)*, pages 302–313, April 1994. 2.4.3, 4.2

[61] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979. 3.2

[62] James Laudon and Daniel Lenoski. The SGI Origin: A cc-NUMA Highly Scalable Server. In *24th Int'l Symp. on Computer Architecture (ISCA)*, pages 241–251, June 1997. 2.1, 2.4.3, 4.2

[63] Hung Q. Le, William J. Starke, J. Stephen Fields, Francis P. O'Connell, Dung Q. Nguyen, Bruce J. Ronchetti, Wolfram M. Sauer, Eric M. Schwarz, and Michael T. Vaden. IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, November 2007. 0.1, 0.5, 1, 1.2, 2.1, 4.2.1, 5.1, 7.1, 7.2.1

[64] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John L. Hennessy, Mark A. Horowitz, and Monica S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992. 2.4.3

[65] Jacob Leverich, Hideho Arakida, Alex Solomatnikov, Amin Firoozshahian, Mark Horowitz, and Christos Kozyrakis. Comparing memory systems for chip multiprocessors. In *34th Int'l Symp. on Computer Architecture (ISCA)*, pages 358–368, June 2007. 1

[66] Man-Lap Li, Ruchira Sasanka, Sarita V. Adve, Yen-Kuang Chen, and Eric Debes. The ALPBench benchmark suite for complex multimedia applications. In *Int'l Symp. on Workload Characterization*, pages 34–45, October 2005. 0.2, 3, 3.4

[67] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *14th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 367–378, February 2008. 7.2.2

[68] Chun Liu, Anand Sivasubramaniam, and Mahmut Kandemir. Organizing the last line of defense before hitting the memory wall for CMPs. In *11th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 176–185, February 2004. 1.2, 2.1

[69] Gabriel H. Loh. 3d-stacked memory architectures for multi-core processors. In *35th Int'l Symp. on Computer Architecture (ISCA)*, pages 453–464, June 2008. 7.1, 7.2.1

[70] Tom Lovett and Russell Clapp. STiNG: A cc-NUMA computer system for the commercial marketplace. In *23rd Int'l Symp. on Computer Architecture (ISCA)*, pages 308–317, June 1996. 4.2

[71] Nir Magen, Avinoam Kolodny, Uri Weiser, and Nachum Shamir. Interconnect-power dissipation in a microprocessor. In *Int'l workshop on System Level Interconnect Prediction (SLIP'04)*, pages 7–13, February 2004. 0.1, 1.1, 5.1

[72] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, February 2002. 0.2, 3, 3.1.1

[73] Milo M.K. Martin. *Token Coherence*. PhD thesis, University of Wisconsin-Madison, December 2003. 5.2

[74] Milo M.K. Martin, Pacia J. Harper, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors. In *30th Int'l Symp. on Computer Architecture (ISCA)*, pages 206–217, June 2003. 5.1, 5.2

[75] Milo M.K. Martin, Mark D. Hill, and David A. Wood. Token coherence: Decoupling performance and correctness. In *30th Int'l Symp. on Computer Architecture (ISCA)*, pages 182–193, June 2003. 1.1, 2.4.2

[76] Milo M.K. Martin, Daniel J. Sorin, Anatassia Ailamaki, Alaa R. Alameldeen, Ross M. Dickson, Carl J. Mauer, Kevin E. Moore, Manoj Plakal, Mark D. Hill, and David A. Wood. Timestamp snooping: An approach for extending SMPs. In *9th Int. Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 25–36, November 2000. 1.1, 5.2

[77] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and

David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, 33(4):92–99, September 2005. 0.2, 3, 3.1.1

[78] Michael R. Marty, J. Bingham, Mark D. Hill, A. Hu, Milo M.K. Martin, and David A. Wood. Improving multiple-cmp systems using token coherence. In *11th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 328–339, February 2005. 0.1, 0.4.3, 2.4.2, 5.1

[79] Michael R. Marty and Mark D. Hill. Coherence ordering for ring-based chip multiprocessors. In *39th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 309–320, December 2006. 8.2

[80] Michael R. Marty and Mark D. Hill. Virtual hierarchies to support server consolidation. In *34th Int'l Symp. on Computer Architecture (ISCA)*, pages 46–56, June 2007. 4.2.1, 8.2

[81] Cameron McNairy and Rohit Bhatia. Montecito: A dual-core, dual-thread itanium processor. *IEEE Micro*, 25(2):10–20, March 2005. 1, 1.2, 2.1

[82] Louis Monier and Pradeep S. Sindhu. The architecture of the dragon. In *30th IEEE Computer Society Int'l Conference*, pages 118–121, February 1985. 2.2

[83] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114—117, 1965. 0.1, 1

[84] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *12th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 254—-265, February 2006. 8.2

[85] Shubhendu S. Mukherjee and Mark D. Hill. An evaluation of directory protocols for medium-scale shared-memory multiprocessors. In *8th Int'l Conference on Supercomputing (ICS)*, pages 64–74, July 1994. 0.3, 4.1, 4.2, 6.3

[86] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers, and Joel Saltz. Efficient support for irregular applications on distributed-memory machines. In *5th Int'l Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 68–79, July 1995. 0.2, 3.4

[87] Ashwini K. Nanda, Anthony-Trung Nguyen, Maged M. Michael, and Douglas J. Joseph. High-throughput coherence control and hardware messaging in Everest. *IBM Journal of Research and Development*, 45(2):229–244, March 2001. 4.1, 4.2, 4.3.4

[88] Håkan Nilsson and Per Stenström. The scalable tree protocol - A cache coherence approach for large-scale multiprocessors. In *4th Int'l Conference on Parallel and Distributed Computing*, pages 498—506, December 1992. 4.2

[89] Håkan Nilsson and Per Stenström. An adaptive update-based cache coherence protocol for reduction of miss rate and traffic. In *6th Int'l Conference on Parallel Architectures and Languages Europe (PARLE)*, pages 363–374, June 1994. 2.2

[90] Brian W. O'Krafka and A. Richard Newton. An empirical evaluation of two memory-efficient directory methods. In *17th Int'l Symp. on Computer Architecture (ISCA)*, pages 138–147, June 1990. 4.2

[91] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Kenneth G. Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *7th Int. Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 2–11, October 1996. 0.1, 1

[92] Jonathan M. Owner, Mark D. Hummel, Derrick R. Meyer, and James B. Keller. *System and method of maintaining coherency in a distributed communication system*. U.S. Patent 7069361, June 2006. 0.4.3, 2.4.1, 6.1

[93] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM Reference Manual, Version 1.0. Technical report, Rice University, Electrical and Computer Engineering Department, August 1997. 2.5

[94] Valentín Puente, José A. Gregorio, and Ramón Beivide. SICOSYS: An integrated framework for studying interconnection network in multiprocessor systems. In *10th Euromicro Workshop on Parallel, Distributed and Network-based Processing*, pages 15–22, January 2002. 0.2, 3, 3.1.2

[95] Alberto Ros, Manuel E. Acacio, and José M. García. A novel lightweight directory architecture for scalable shared-memory multiprocessors. In *11th Int'l Euro-Par Conference*, pages 582–591, August 2005. 0.1.1, 1.3, 4.2, 5.2, 5.2.1

[96] Alberto Ros, Manuel E. Acacio, and José M. García. An efficient cache design for scalable glueless shared-memory multiprocessors. In *ACM Int'l Conference on Computing Frontiers*, pages 321–330, May 2006. 0.1.1, 1.3, 2.3, 5.2.1

[97] Alberto Ros, Manuel E. Acacio, and José M. García. Direct coherence: Bringing together performance and scalability in shared-memory multiprocessors. In *14th Int'l Conference on High Performance Computing (HiPC)*, pages 147–160, December 2007. 0.1.1, 1.3, 2.1, 5.1

[98] Alberto Ros, Manuel E. Acacio, and José M. García. DiCo-CMP: Efficient cache coherency in tiled cmp architectures. In *22nd Int'l Parallel and Distributed Processing Symp. (IPDPS)*, pages 1–11, April 2008. 0.1.1, 1.3, 5.1

[99] Alberto Ros, Manuel E. Acacio, and José M. García. Scalable directory organization for tiled cmp architectures. In *Int'l Conference on Computer Design (CDES)*, pages 112–118, July 2008. 0.1.1, 0.3, 1.3, 4.1

[100] Alberto Ros, Manuel E. Acacio, and José M. García. Dealing with traffic-area trade-off in direct coherence protocols for many-core cmps. *To appear in 8th Int'l Conference on Advanced Parallel Processing Technologies (APPT)*, August 2009. 0.1.1, 1.3

[101] Alberto Ros, Manuel E. Acacio, and José M. García. *Parallel and Distributed Computing*, chapter Cache Coherence Protocols for Many-Core CMPs. IN-TECH, 2009. 0.1.1, 1.3

[102] Alberto Ros, Ricardo Fernández-Pascual, Manuel E. Acacio, and José M. García. Two proposals for the inclusion of directory information in the last-level private caches of glueless shared-memory multiprocessors. *Journal of Parallel Distributed Computing (JPDC)*, 68(11):1413–1424, November 2008. 0.1.1, 1.3, 2.3, 5.2

[103] Nabeel Sakran, Marcelo Uffe, Moty Mehelel, Jack Dowweck, Ernest Knoll, and Avi Kovacks. The implementation of the 65nm dual-core 64b merom processor. In *IEEE Int'l Solid-State Circuits Conference (ISSCC)*, pages 106–590, February 2007. 1.2, 2.1, 7.1

[104] Manish Shah, Jama Barreh, Jeff Brooks, Robert Golla, Gregory Grohoski, Nils Gura, Rick Hetherington, Paul Jordan, Mark Luttrell, Christopher Olson, Bikram Saha, Denis Sheahan, Lawrence Spracklen, and Aaron Wynn.

UltraSPARC T2: A highly-threaded, power-efficient, SPARC SoC. In *IEEE Asian Solid-State Circuits Conference*, pages 22–25, November 2007. 0.1, 0.5, 1, 1.2, 2.1, 2.4.3, 7.1, 7.2.1

[105] Richard Simoni. *Cache Coherence Directories for Scalable Multiprocessors*. PhD thesis, Stanford University, 1992. 4.2

[106] Richard Simoni and Mark A. Horowitz. Dynamic pointer allocation for scalable cache coherence directories. In *Int'l Symp. on Shared Memory Multiprocessing*, pages 72–81, April 2001. 4.2

[107] Daniel J. Sorin, Manoj Plakal, Anne E. Condon, Mark D. Hill, Milo M. K. Martin, and David A. Wood. Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. 13(6):556–578, June 2002. A

[108] Per Stenström. A survey of cache coherence schemes for multiprocessors. *IEEE Transactions on Computers*, 23(6):12–24, June 1990. 2.2

[109] Per Stenström, Mats Brorsson, Fredrik Dahlgren, Håkan Grahn, and Michel Dubois. Boosting the performance of shared memory multiprocessors. *IEEE Transactions on Computers*, 30(7):63–70, July 1997. 2.2

[110] Per Stenström, Mats Brorsson, and Lars Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *20st Int'l Symp. on Computer Architecture (ISCA)*, pages 109–118, May 1993. 2.3.1

[111] Sun Microsystems, Inc., Santa Clara, CA 95054. *OpenSPARC™ T2 System-on-Chip (SoC) Microarchitecture Specification*, December 2007. 0.3.1, 3.1.3, 4.1, 4.2.1, 5.1

[112] Paul Sweazey and Alan J. Smith. A class of compatible cache consistency protocols and their support by the IEEE futurebus. In *13th Int'l Symp. on Computer Architecture (ISCA)*, pages 414–423, June 1986. 2.3

[113] Michael B. Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Benjamin Greenwald, Henry Hoffman, Jae-Wook Lee, Paul Johnson, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matthew Frank, Saman Amarasinghe, and Anant Agarwal. The raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, 22(2):25–35, May 2002. 1

[114] Radhika Thekkath, Amit P. Singh, Jaswinder P. Singh, Susan John, and John L. Hennessy. An evaluation of a commercial cc-NUMA architecture: The CONVEX Exemplar SPP1200. In *11th Int'l Parallel Processing Symp. (IPPS)*, pages 8–17, April 1997. 4.2

[115] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. Cacti 5.1. Technical Report HPL-2008-20, HP Labs, April 2008. 0.2, 3, 3.1.3, 3.2, 4.1

[116] Hangsheng Wang, Li-Shiuan Peh, and Sharad Malik. Power-driven design of router microarchitectures in on-chip networks. In *36th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 105–111, December 2003. 0.1, 1.1, 5.1

[117] Wolf-Dietrich Weber and Anoop Gupta. Analysis of cache invalidation patterns in multiprocessors. In *3th Int. Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 243–256, April 1989. 2.3.1

[118] Steven C. Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder P. Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd Int'l Symp. on Computer Architecture (ISCA)*, pages 24–36, June 1995. 0.2, 3, 3.4

[119] Michael Woodacre, Derek Robb, Dean Roe, and Karl Feind. The SGI Altix™ 3000 global shared-memory architecture. Technical Whitepaper, Silicon Graphics, Inc., 2003. 4.2

[120] Win A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, March 1995. 1.1

[121] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *13th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 261–272, February 2007. 0.4.1.2, 5.4, 5.4

[122] Michael Zhang and Krste Asanovic. Victim migration: Dynamically adapting between private and shared CMP caches. Technical report, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, October 2005. 7.2.2

[123] Michael Zhang and Krste Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *32nd Int'l Symp. on Computer Architecture (ISCA)*, pages 336–345, June 2005. 0.5, 1, 2.1, 4.6.2, 5.1, 5.2, 7.1, 7.2.2