University of Cantabria
Electronic and Computers Department

NEW SCALABLE CACHE COHERENCE PROTOCOLS FOR ON-CHIP MULTIPROCESSORS

Author:
Lucía Gregorio Menezo

Advisors:
Valentín Puente Varona
José Ángel Gregorio Monasterio

Santander, March 2014

Universidad de Cantabria
Departamento de Electrónica y Computadores

NUEVOS PROTOCOLOS DE COHERENCIA ESCALABLES PARA MULTIPROCESADORES EN CHIP

Autora:
Lucía Gregorio Menezo

Directores:
Valentín Puente Varona
José Ángel Gregorio Monasterio

Santander, Marzo 2014

Abstract

During the last decade, the increasing complexity of new processors, as well as their high cost and energy consumption, has reoriented the industry towards the introduction of multiple cores into the chip, i.e. Chip Multiprocessors (CMPs). This reorientation attempts to exploit thread-level parallelism; it limits each core's complexity and reduces the energy required to perform any task. However, besides their advantages, CMPs also present new challenges that have to be faced, such as the bandwidth-wall.

In order to mitigate this problem, one of the steps taken consists of using complex memory hierarchies within the chip, thus reducing the number of off-chip accesses. This raises the possibility of having multiple copies of the same data and therefore, the necessity of having to maintain the coherence among them. This coherence can be maintained via software, leaving all the responsibility to the programmers/compilers, at the expense of their productivity, or otherwise in hardware, freeing them from such an arduous task.

In this thesis, firstly an analysis has been done of the problems associated with cache coherence in the field of CMPs. The main existing solutions, both in real machines and from a purely academic perspective, have been summarized, along with the characteristics of the interconnection networks that have been or may be introduced into such systems. This analysis has enabled the detection of a set of needs and opportunities, which have been incorporated into two new proposals.

On the one hand, considering a medium-term future in which the number of processors in the chip will be a few tens and with a view to fully exploiting the high bandwidth availability due to the use of point-to-point networks and scalable cache architectures, a new coherence protocol denominated LOCKE is proposed. This proposal uses a broadcast-based approach, which focuses on improving the reactiveness of the on-chip memory hierarchy, as well as the system stability against the effects of contention.

On the other hand, considering the long-term future, large-scale CMPs will include hundreds or thousands of processors, and it should be noted that the interconnection network will not be able to support such high numbers of messages. Therefore, MOSAIC is proposed, which provides a scalable hybrid coherence protocol (broadcast- and directory-based). It uses the CMP bandwidth availability in a controlled way, significantly reducing the maintenance costs of other existing protocols.

Resumen

Durante la última década, la creciente complejidad de diseño de nuevos procesadores, así como su elevado coste y consumo energético ha reorientado la industria hacia la introducción de múltiples cores dentro del chip, i.e. Chip Multiprocessors (CMPs). Dicha reorientación tiene como finalidad explotar el paralelismo a nivel de *thread*, limitar la complejidad de cada core, y reducir la energía requerida para llevar a cabo cualquier tarea. No obstante, además de ventajas, los CMPs también presentan nuevos retos a superar, como el *bandwidth-wall*.

Una de las soluciones adoptadas para mitigar este problema consiste en usar jerarquías de memoria complejas dentro del chip, logrando así reducir el número de accesos al exterior. Sin embargo, ello lleva consigo la potencial existencia de múltiples copias de un mismo dato y por lo tanto la necesidad de mantener la coherencia entre todas ellas. Dicha coherencia debe mantenerse bien vía software, delegando en el programador/compilador toda responsabilidad en detrimento de su productividad, o por el contrario vía hardware, liberándole de tal ardua tarea.

En este trabajo, en primer lugar se ha llevado a cabo un análisis sobre la problemática asociada a la coherencia cache en el ámbito de los CMPs. Extrayendo las principales soluciones existentes actualmente, tanto implementadas en máquinas reales, como soluciones propuestas desde el punto de vista puramente académico, así como las características de las redes de interconexión que han sido o pueden ser introducidas en este tipo de sistemas. Este análisis ha permitido detectar un conjunto de necesidades y de oportunidades, que se han materializado en dos nuevas propuestas.

Por un lado, considerando un futuro a medio plazo en el que el número de procesadores dentro del chip será de pocas decenas y buscando explotar plenamente la alta disponibilidad de ancho de banda, como consecuencia de la utilización de redes punto-a-punto y de la utilización de arquitecturas de cache escalables, se propone un protocolo de coherencia, denominado LOCKE. Esta propuesta utilizando una aproximación basada en *broadcast*, se centra en mejorar la reactividad de la jerarquía de memoria *on-chip* y estabilidad del sistema antes los efectos derivados de la contención.

Por otro lado, poniendo la mira a largo plazo, sobre CMPs de gran escala que incluirán cientos o miles de procesadores, se debe tener en cuenta que la red de interconexión no podrá soportar cantidades tan elevadas de mensajes. Por esta razón, se propone MOSAIC, un protocolo escalable hibrido *broadcast*-directorio que, usando de forma controlada la disponibilidad de ancho de banda del CMP, logra disminuir significativamente el coste del mantenimiento de la coherencia característico en otras soluciones existentes.

Agradecimientos

Mamá, cada una de las letras, comas y puntos de este trabajo son gracias a ti. A tu vida, a tu tiempo, a tu dedicación, a tus abrazos y a tu ejemplo de trabajo, constancia y sacrificio. Gracias.

Papá, no puedo imaginarme estos años sin ti. A pesar de la dureza del camino, de la que ya me avisaste, no cambiaría por nada del mundo todas las horas que hemos compartido juntos gracias a tanto protocolo y coherencia. Gracias.

Javivi, gracias por ser siempre mi sinónimo de compañía. Nunca fallas y siempre estás conmigo para ayudarme y hacerme reír hasta con la tontería más grande. Te quiero broder.

David, Haf, Asier y el resto de la Paco's, fuisteis los primeros en notar el amor que he sentido desde el principio por la arquitectura de computadores y nunca dudasteis de mí. Que profesionales como vosotros confíen en mí me reconforta sobremanera.

Mis chicas del banco del patio, Eila, Leti y Sani, no sois conscientes de lo que me han ayudado en el día a día esas cortas y largas conversaciones por cualquiera de las vías. Gracias por soportarme.

Y el resto de mi familia y amigos, no os puedo enumerar a todos, pero tenéis la "culpa" de mi sonrisa. Eider, María, Mortadelos, Chiringuiteros, Hamstargs, Marqués, Pradejoneros… gracias por todas las horas de playa, cañas, barbacoas, celebraciones, viajes, juegos, calimochos, deporte, McDonald's, etc. Habéis pintado de colores todos estos años.

No puedo omitir a mis compañeros de trabajo. Javi, gracias porque hiciste que mis inicios en esto fueran mucho más amables de lo que podrían haber sido. Ah! Y por crear *gosset,* impagable. Pablo A. Pablo P. y Adrián, gracias por vuestro compañerismo y vuestro ánimo y apoyo constante. Sin vosotros todo hubiese sido mucho más oscuro. Jose, tu paciencia, comprensión y dedicación son inestimables, gracias. Y Valentin, me siento tremendamente afortunada de haberte tenido como director de tesis. No podría haber escogido como guía a nadie mejor que tú. Gracias.

Y por último, Jose, riojano de mis amores. Es difícil escribir en un párrafo el agradecimiento que siento por todo lo que haces día a día por mí. La verdad que en la portada tendría que aparecer también tu nombre, porque nada de esto hubiese sido posible si tú no hubieras estado a mi lado desde mucho antes del principio. Gracias por ese microsegundo en el que decidiste venirte a Santander y comenzar tu nueva vida aquí, conmigo. Gracias por todos esos abrazos que me han recargado de energía para seguir adelante. Gracias por tu visión optimista de la vida, tu actitud siempre trabajadora y tu carácter soñador. Las tres me las has transmitido y han sido clave para conseguir esta tesis. Te quiero.

Table of contents

Chapter 1. Introduction

Nowadays, thousands of millions of transistors are available in a single-die and the best known way of taking full advantage of this is to include multiple processing cores. Since Gordon Moore's prediction in 1965 [1], advances in technology allowed to duplicate the number of transistors that can be integrated into the chip to be doubled approximately every 18 months. However, performance improvement kept growing at that rate up until the end of the last century. From that point, the additional hardware required to exploit instruction level parallelism (ILP) only allowed the performance to grow as the square root of the number of transistors needed [2]. This happened because the substantial amount of logic that had to be placed in the chip in order to be able to support a large number of instructions in-flight increases the energy consumption considerably. Moreover, as transistor size shrinks, wire delay does not decrease and so the relative distance between processing units also increases. Additionally, higher complexity causes another negative effect, which is the appearance of difficulties in the verification of the whole system, implying consequences that have to be taken into account (economics, time-to-market, etc.).

On the other hand, although frequency has been constantly increased, power has been limited thanks to the voltage scaling [3]. The power consumed by a chip has a linear dependency on the frequency (and capacity), but a quadratic one on the voltage, i.e. $P \propto C*V^2*f$. For the last 30 years, voltage has gone from 12V to less than 1V, which means a reduction of more than 150X. However, under 1V it is physically difficult, to keep on decreasing the power threshold of the transistor and so an increment of the frequency means an increment in the power that cannot be countered by reducing the supply voltage.

During the last decade, this performance loss of the transistor has reoriented the industry to introduce multiple cores inside the same chip, i.e. Chip Multiprocessors (CMPs) or multicore processors [4][5][6][7]. Although they have some limitations that cannot be ignored, their benefits are much greater. Firstly, the complexity of each of the cores that form the CMP limits the complexity of the whole chip. Secondly, if it is possible to take advantage of each core's performance, it is possible to reduce the power consumption that is required to finish a task. Thus, a task may be accomplished by two cores with half the frequency needed when only one core is present in the chip (assuming ideal parallelism). However, lowering the frequency allows the transistor power threshold to be decreased, which means that it is possible to reduce the supply power, and therefore to decrease quadratically the energy required to carry out that task.

Obviously, this new paradigm with multiple cores also presents some new challenges. Among them, we can consider the most crucial one to be the one denominated bandwidth-wall [8]. This obstacle is due to the limited growth of the number of pins and of the operating frequency due to physical and packaging cost restrictions. The off-chip communication necessities grow as the number of cores and their complexity increase. However, the available off-chip bandwidth does not increase at the same rate, becoming a bottleneck of the whole CMP. Some studies [9] predict that this problem will limit the number of cores that can be introduced inside the chip. Fortunately, there are a wide variety of solutions that are able to mitigate the problem. Among them, the one that appears to have most benefits is the introduction of large amounts of memory inside the chip. This solution aims to require the cores to have to go outside the chip to find the requested information less often. In this way, based on the spatial and temporal locality of the applications, the memory hierarchy will help to keep the most useful data closer to the processing units of the CMP.

Therefore, if we place enough memory within the chip so that the majority of the applications' working sets fit inside, external memory accesses will decrease and so the performance will not be limited by the off-chip bandwidth restrictions. However, the efficient organization and management of large amounts of memory associated to each of the cores is not a straightforward task. There is a consensus among computer architects which assigns some cache memory to each of the cores in a stepped way at different levels. From the performance point of view, there cannot be "steps" with excessive difference in capacity among them [10]. So it is established that there should be one very small first level of private cache memory, with tens of KB and with low associativity, in order to provide fast access time to data close to the processors. Nowadays, it also seems clear that a second private cache memory level is also convenient, larger than the first one and capable of absorbing a large percentage of the miss accesses that happen in the first level. Concerning the last level of the cache hierarchy inside the chip, commonly referred to as LLC (Last Level Cache), there is much less unanimity about its distribution and characteristics. Some distribute it as a private cache and others design it to be shared among some or all the cores in the chip. This decision will have a significant effect on the CMP usage. In almost all the CMPs implemented so far, the LLC is shared among the cores in the chip (Bulldozer [5], Haswell [6], Sparc T5 [7]), because it seems that a better utilization of the memory is possible. Other companies are tending to maintain the LLC as local caches for each core although all the banks are used as victim cache by the rest (Power7 [4]). Although the most common number of levels used nowadays is three [4][11], there are already some new commercial systems which include more levels, such as the IBM z196 [12] which includes a fourth level (although it is outside the chip in order to mitigate the time access gap between the

LLC and the off-chip memory). As soon as the technology allows it, with mechanisms such as 3D stacking [13], it will not take long to see more levels introduced inside the chip.

In any case, from the moment there are multiple copies of the same block in the system, coherence has to be enforced. Therefore, it is important to decide how it will be managed: via hardware and/or via software. There are numerous works analyzing the advantages of exposing to the programmer-compiler the capability of handling the data coherence of the blocks allocated in the private caches [14][15]. However, most of those studies are focused on performance comparison, i.e. execution time, and only considering a very specific type of applications. When taking into account general purpose applications, with the large amounts of memory in a multi-level hierarchy, coherence management is not a trivial task. For this reason, programming parallel applications without the hardware support to do this might hinder the productivity of programmers because they will have to pay too much attention to this duty. Finally, as in [16]'s discussion, we believe that for the next few years, data coherence maintenance should be done by hardware and the search for efficient solutions for this has been the main target of this thesis.

1.1 Objectives. Coherence in CMPs

The main aim of this thesis was to search for efficient mechanisms to maintain the coherence of data present in a CMP memory hierarchy. As long as the off-chip bandwidth limitation exists and, due to the increasing number of cores inside the chip, the cache capacities will grow gradually. For this reason, the more memory that can be placed inside the chip, the greater the number of levels and therefore it will be more difficult to control all the copies of a certain block in order to keep the system coherent.

The first step of this work was to analyze in detail the state of the art of the most important proposals for improving coherence protocols in multiprocessor systems, with special emphasis on chip multiprocessor systems (CMPs). This analysis focused on those works that appear to have more future either because of their performance improvements, their scalability characteristics or both.

The second objective was to fully design and implement a coherence protocol which is able to exploit the large on-chip bandwidth availability while improving cache-coherent CMP performance and maintaining its efficiency. The whole proposal is based on the idea that the coherence protocol should use all the on-chip network bandwidth availability to avoid adding extra latency produced by indirections. Although bandwidth demand is still a concern, with a

suitable interconnection network design, it is possible to increase the whole system performance by improving the coherence protocol behavior, without paying a significant energetic cost.

The third target was to design a new coherence protocol which addressed the challenges of complex multilevel cache hierarchies in future many-core systems. This really entails a search through some coherence mechanisms that enable the number of cores and the number of cache levels in the system to be increased, while limiting the overhead caused by the hardware coherence maintenance. In the long term, with the advent of 3D stacking or beyond-CMOS technologies, the tendency of increasing the amount of private cache per core will be accentuated. Under these conditions, the amount of precise sharing information required by coherence protocols will be increased and therefore, it is necessary to look for new scalable ones.

The last objective, transversal to the others, was the acquisition of the necessary expertise of the simulation tools required for a trustworthy validation of the proposals made. For an acceptable confidence level, it is essential to use powerful tools that emulate the behavior of a full CMP executing a realistic workload in detail. The use of these tools is highly complex but it is essential both for this work and for future research in the area where this thesis is located.

1.2 Thesis contributions

The main contributions of this thesis are directly related to the achievement of the goals that have been outlined in the previous section. Next, we give a brief description of each of them:

- An insight into the most relevant related work on cache coherence protocols for CMPs. This step was absolutely necessary in order to contextualize the two main proposals presented in this work. Obviously, the number of research works related to this topic is very high, so the work done has been to analyze those considered to be the most relevant ones from the point of view of their impact and which can be considered the basis of current developments.
- The full design and implementation of a new coherence protocol suitable for medium-scale CMPs, named LOCKE (LOCator of toKEns). Considering that an interconnection network is suitably designed to support multicast traffic and that the protocol maximizes the potential advantages that direct coherence brings, we demonstrate that a multicast-based coherence protocol could reduce energy requirements of a CMP memory hierarchy. The protocol establishes a suitable level of on-chip network throughput to accelerate synchronization by two means: avoiding the protocol serialization, inherent to any directory-based coherence protocol, and reducing average access time of other snoop-based coherence protocols,

especially when shared data is highly contended. LOCKE is developed on top of a Token coherence performance substrate [17], with a new set of simple proactive policies that speeds up data synchronization and eliminates the passive token starvation avoidance mechanism.

- A brand new coherence protocol called MOSAIC, suitable for large-scale CMPs, which successfully addresses the challenges of complex multilevel cache hierarchies in future many-core systems. The design and implementation of MOSAIC introduces a new approach to tackle the inclusiveness problem in the memory hierarchy. In energy terms, the protocol scales like a conventional directory coherence protocol, but relaxes the inclusiveness needed for the directory information, overcoming the performance implications of a reduction in directory size and associativity. Contrary to the common assumption about inclusiveness being inescapable while attempting to maintain complexity constrained, MOSAIC is even simpler than a conventional directory.
- A thorough evaluation of every proposed implementation is provided. This basically meant an exhaustive exploration of the whole set of full-system simulation tools used. In order to be able to perform full system evaluation, we used the Simics [18] functional simulator in conjunction with the processor and memory hierarchy timing models from GEMS [19] and the detailed network simulator TOPAZ [20]. The implementation of the proposals was done using the domain specific language SLICC (Specification Language for Implementing Cache Coherence).

Both coherence protocol proposals presented, LOCKE and MOSAIC, were published during the development of this thesis with the following references:

- L.G. Menezo, V. Puente, P. Abad, J.A. Gregorio, *Improving Coherence Protocol Reactiveness by Trading Bandwidth for Latency*, ACM International Conference on Computing Frontiers (CF'12), July 2012.
- L.G. Menezo, V. Puente, J.A. Gregorio, *The Case for a Scalable Coherence Protocol for Complex On-Chip Cache Hierarchies in Many Core Systems*, IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques (PACT), Edinburgh, September 2013.

These two publications entailed large previous analysis work that originated indirect publications in different conferences and journals, mainly related to the interconnection network and its interrelation with coherence protocol design and performance:

- P. Abad, P. Prieto, L.G. Menezo, A. Colaso, V. Puente, J.A. Gregorio, *Interaction of NoC design and Coherence Protocol in 3D-stacked CMPs*, Euromicro Conference on Digital System Design (DSD), September 2013.
- P. Abad, V. Puente, L. G. Menezo, J.A. Gregorio, *Adaptive-Tree Multicast: Efficient Multi-destination Support for CMP Communication Substrate*. IEEE Transactions on Parallel and Distributed Systems. Vol. 23, no. 11, pp. 2010 – 2023, Nov 2012.
- P. Abad, P. Prieto, L.G. Menezo, A. Colaso, V. Puente, J.A. Gregorio, *TOPAZ: An Open-Source Interconnection Network Simulator for Chip Multiprocessors and Supercomputers*, IEEE/ACM International Symposium on Networks on Chips (NoCS), Denmark, 2012.
- L.G. Menezo, A. Colaso, V. Puente, J.A. Gregorio, *Beneficios del uso de la Red de Interconexión en la Aceleración de la Coherencia*, XXII Jornadas de Paralelismo, La Laguna (Spain), September 2011.
- L.G. Menezo, A. Colaso, V. Puente, J.A. Gregorio, *Exploring Coherence Protocol Acceleration through the Interconnection Network*, Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES), Italy, July 2011.
- J. Merino, L.G. Menezo, P. Abad, P. Prieto, V. Puente, *Arquitectura Cache Adaptativa para CMPs*, XXI Jornadas del Paralelismo dentro del marco del III Congreso Español de Informática (CEDI 2010), Valencia, September 2010.
- P. Abad, P. Prieto, J. Merino, L.G. Menezo, V. Puente, *Impact of Interconnection Network resources on CMP performance*, 4th Workshop on Interconnection Network Architectures: On-Chip, Multi-Chip (INA-OCMC 2010), Pisa-Italia, January 2010.
- J. Merino, L.G. Menezo, V. Puente, *Needs of CQoS for future, many-core CMPs to support server consolidation and cloud computing*, XX Jornadas de Paralelismo, La Coruña, September 2009.
- J. Merino, L.G. Menezo, V. Puente, *Needs of CQoS for many-core CMPs: a case study*, V International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES), La Mola (Barcelona), July 2009.
- L.G. Menezo, J. Merino, P. Prieto, P. Abad, J.A. Gregorio, *Impacto de la Red de Interconexión sobre los Protocolos de Coherencia en Sistemas Multiprocesador*, XIX Jornadas del Paralelismo, Castellón, September 2008.
- J. Merino, L. G. Menezo, P. Prieto, V. Puente, *Implementación de un protocolo de coherencia basado en token en el simulador GEMS*, XIX Jornadas del Paralelismo, Castellón, September 2008.

1.3 Thesis overview

The remainder of this thesis is organized as follows.

Chapter 2 presents the foundations needed to contextualize the rest of this memory. The coherence definition, and how coherence is achieved in a multiprocessor system will be analyzed, and the two main types of coherence protocols will be explained. To end the chapter, the special role of the interconnection network in coherence will be studied.

In **Chapter 3** we will provide a general overview of the state of the art of cache coherence. We will analyze the main problems that cache coherence protocol designers face nowadays and how their concerns are being solved so far. Last, we try to predict the evolution of the different aspects of coherency.

In **Chapter 4** we introduce LOCKE, beginning with the motivation for trading bandwidth for latency in order to improve coherence protocol reactiveness. We describe the proposal itself, explaining the foundations of the coherence protocol, and we provide insight into LOCKE performance and efficiency when compared with other alternatives.

In **Chapter 5**, we describe the new scalable coherence protocol for large architectures, MOSAIC. From a conceptual approach to all the implementation details, and through a complete performance evaluation, the proposal is fully described. This chapter finalizes with a scalability analysis of the coherence protocol.

This document also includes an **Appendix A** with a summary of the simulations tools used to evaluate both coherence protocol proposals made in this thesis.

Finally, **Chapter 6** will finish this thesis with several conclusions summarizing the contributions of this work, and with an outlook of future research lines.

Chapter 2. Coherence protocols

The extended Von Neumann model established the existence of three main blocks in any machine: processor, memory and input/output systems. As was mentioned before, Moore's Law indicates that the number of transistors that can be put into an integrated circuit has doubled every 18 months. This has facilitated an increment in the processor frequency during the last decades. As processors became faster, too much difference appeared between the processor speed and the speed of access to memory. This difference was alleviated by including a multi-level memory hierarchy. The smallest but faster level resides close to the processor (lower levels in this document) and the biggest but slower ones are farther away (upper levels in this document). Thus, the increasing difference between the speed of the processors and the speed of memory access is concealed. With various levels of memory and only one processor, it becomes relatively simple to control possible multiple copies of a datum. The architecture implementation of the caches itself solves this problem with write-through and write-back policies. Whenever a data block is written in the closest level to the processor, it has to be updated in the farther levels. Depending on the update implementation this is done at the same time on all of them (write-through) or it is only done when a block is being removed from one level, then updating the next one (write-back).

At the beginning of the previous decade, the translation of Moore's prediction into performance improvement started to become progressively harder. The reason for this was the diminishing returns obtained with instruction level parallelism and the end of power scalability. The most cost-effective and energy-efficient solution seemed to be putting many simple cores into the chip. Thus, it was possible to maintain the pace of performance improvement achieved until that time by exploiting the thread level parallelism that this new architecture enabled.

In most cases, all these cores inside the chip share a common address space. If this space is maintained in one shared structure, its access becomes a bottleneck, degrading the performance of the whole system. In order to avoid it and to achieve faster data accesses, private levels are added to the memory hierarchy and so each processor will have its own cache space to hold the most recently accessed memory blocks. Under these circumstances, several copies of the same datum may appear in the system necessitating some kind of mechanism in charge of guaranteeing the same view of the whole address space to all the processors. This is the duty of the coherence protocol.

The definition of coherence will be analyzed in section 2.1 and the methodology for achieving hardware coherence will be dealt with in section 2.2. The way coherence protocols are specified will be described in section 2.3. Next, the two main types of coherence protocols will be explained in detail. Section 2.4 will be dedicated to snoopy protocols and section 2.5 to those based on a directory. Advantages and disadvantages will be analyzed in 2.6 and, finally, section 2.7 will close the chapter with an analysis of the special role that the interconnection network has in the coherence.

2.1 What is memory coherence?

There are *multiple* alternatives to define coherence in any system. An intuitive definition determines three invariants that must be fulfilled in order to have a coherent memory system [2]:

- If processor P_1 writes in address X and then reads in the same address, it should read the written value.
- If processor P_1 writes in address X and then another processor P_2 reads in the same address, it should read the previously written value *if both reads are sufficiently separated in time*.
- Writes to the same location are serialized, i.e. two writes to the same location by any two processors are seen in the same order by all processors.

However, in this definition, the concept of '*sufficiently separated in time*' in invariant 2 is ambiguous since it does not explicitly say when the value written in a memory location is propagated to others, as this has to do with the memory consistency model used.

To achieve these 3 invariants it is mandatory to fulfill the following three conditions:

> ***Condition 1***: memory access should be done in order (as expected of a uniprocessor system).
>
> ***Condition 2***: write propagation: the awareness of a write operation has to eventually get to the other processors. Note that precisely **when** it is propagated is not established by the definition of coherence.
>
> ***Condition 3***: write serialization.

The important thing is that any core has to read the last valid value for any memory location. There are two important related concepts: coherence, which concerns what to read and write in the same memory location and consistency, where it matters when any modified value is seen by others, determining the behavior of reads and writes in relation to other memory locations.

Principally, for a system to be coherent there **always has to be a single writer of a location and there may be multiple readers of that same location** [21].

2.2 How is hardware coherence achieved?

Coherence protocols are implemented by state machines called *coherence controllers*. Each storage element of the memory hierarchy has a controller specially dedicated to the task of acting as an interface with other components of the memory hierarchy and the processors. These are in charge of sending and receiving the messages needed to achieve the invariants mentioned in the previous section, integrating all of them as a distributed system.

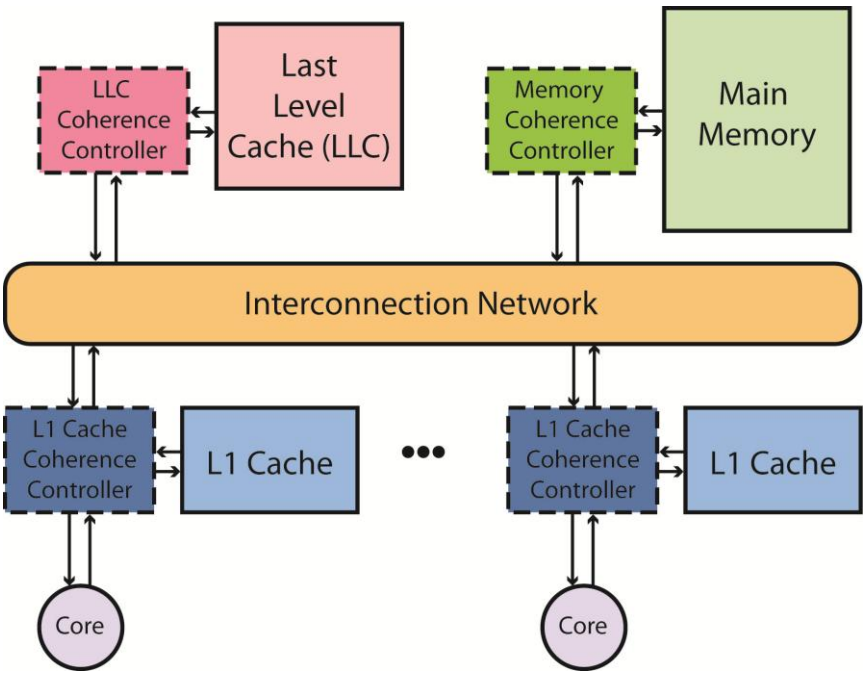

Figure 2-1. Representation of three types of coherence controllers: cache, last level cache (LLC) and memory controllers.

As figure 2-1 shows, coherence controllers are always connected to the interconnection network receiving and sending messages through it. The first level cache coherence controllers are the only ones which will receive load and stores from the core connected to them and they will check whether the requested data address is located in the cache alongside them or not. They may use the interconnection to issue a request to other controllers of the cache hierarchy or to main memory, always depending on the specific implementation of each protocol, as will be seen in the following sections. For this reason, coherence controllers will also have to deal with requests received by other coherence controllers.

Although it may be too strict to absolutely classify the existing coherence protocols into different types, as nowadays the hybridization of different approaches is quite usual, the consensus defines two different types of protocols: snooping and directory protocols. In the following sections, the two types will be described in detail.

2.3 Specifying coherence protocols

Before describing the two main types of coherence protocols, it is necessary to define how to specify them. As was mentioned before, the main aim of the coherence protocols is to regulate how and when a core may access any memory location. Although cores may perform load and store operations at various granularities, coherence protocols are usually specified at cache block granularity, i.e. each block will have specific permissions assigned in order to be read, written or neither.

In order to specify a coherence protocol, and considering that the coherence controller is represented as a state machine, it is necessary to define four important sets: *states*, *events*, *actions* and *transitions*. With all of them it is possible to determine the situation of a cache block, its access permission and the tasks that have to be performed after receiving any message.

2.3.1 States

The coherence protocol states may be divided into two different subsets: *stable* and *transient* states. As their names indicate, the difference between them lies in whether the controller is in the middle of a transition between two stable states or not. In a naïve multicore system where it is not possible to have multiple copies of the same block (i.e. not sharing blocks), the simplest coherence protocol has three states. Intuitively it needs just two stable states: *valid* (V) and *invalid* (I), indicating whether the block is present or not. However, when a private controller receives a request from the processor and it confirms that the block is not available, it issues a request for that block. In this case, the block needs to be in some state that represents this transitional situation of being requested although still not present. For this reason, the protocol needs a third transient state indicating this circumstance (I_V).

When adding optimizations to the coherence protocol, it becomes more complex and so the number of stable and transient states increases. For example, to reduce memory bandwidth consumption, the valid state (V) may be divided into two different ones depending on whether the block has been *modified* (M) or not. In this way it is possible to know whether the data block needs to be written back to a higher level or can be silently replaced. Moreover, increasing the

number of cores in the system gives the protocol designer the possibility of adding new states to improve its performance. For instance, it is possible to distinguish when a core has the cache block in an *exclusive* (E) way and no other core has it so as to avoid subsequent upgrade misses; or also, if the cache block is *shared* (S) among some cores and can only be read but not modified. Additionally, in order to transform memory-to-cache transfers into cache-to-cache ones, from all the possible sharing copies of the block, one of them might be set as the *Owner* (O) of the block, becoming responsible for solving other core requests.

Summarizing, a coherence protocol can be designed to have each cache block in five main situations that will be determined with the following five states:

- *Invalid* (I): the block does not have a valid copy of the datum.
- *Shared* (S): there are various copies of a data block distributed throughout the system. Its holders can only read the datum and they will have to ask for permission if they want to modify it.
- *Modified* (M): the block has been modified (known as 'dirty') and it is the only copy in the system.
- *Exclusive* (E): the block is the only private copy of the data block so its holder may read and write in it.
- *Owner* (O): the cache holding this block is in charge of dealing with other coherence controller requests as it has the ownership of the block. If the block was replaced, the ownership would be given to another coherence controller.

Depending on the specific characteristics of each coherence protocol, the number of states and their particular meaning may differ, because the protocol designer uses these block states to determine all the characteristics of the blocks present in the system.

Moreover, it is important to make clear that coherence controllers at different levels may not have the same list of possible block states, because not all controllers have the same duty in the coherence protocol used in the system.

2.3.2 Events

According to the messages received, the coherence controller will trigger specific events for each of them. Most of the protocols have two different types of events: the ones being triggered because of a data block request and the ones triggered because of lack of space in the cache.

Table 2-1 lists the most common events triggered by a coherence controller that are usually present in any coherence protocol. However, it is important to recall that these are generic ones

and a coherence protocol may require variations of these states. Moreover, there are many other possible events that may occur depending on the specific behavior of each coherence controller.

The first three events are operations ordered by the processor attached to the controller so they will only happen in the L1 cache controllers. The next three events are triggered when one controller receives a request from another one. Notice the subtle difference between the *GetExclusive* and the *Upgrade* events, which only differ in whether the requestor has a copy of the data block requested or not. The last four events listed in table 2-1 are used when it is necessary to evict a block from a cache and to send it to higher levels of the hierarchy.

Table 2-1. Main events triggered by a coherence controller.

Events	Description
Load	Processor read request
Ifetch	Processor instruction fetch request
Store	Processor write request
GetS (GetShared)	Read request from another controller asking for a copy of the block
GetX (GetExclusive)	Write request from another controller asking for a copy of the block and for the invalidation of the rest of them
Upgrade	Write request from another controller that has a copy of a data block and asks for the invalidation of the rest of the copies
PutS (PutShared)	The controller replaces a shared block
PutE (PutExclusive)	The controller replaces an exclusive block
PutO (PutOwned)	The controller replaces an owned block
PutM (PutModified)	The controller replaces a modified block

It is important to have a clear definition and differentiation of the coherence protocol events, because they will guide the coherence controller through the different state transitions performing the necessary actions in each of them.

2.3.3 Transitions and actions

When a cache block is in a certain *state* and the coherence controller receives a message, triggering a specific *event*, that cache block may suffer a *transition* to another state. Each transition implies the execution of one or more *actions* in the system. For example, when a coherence controller receives a 'load' message from the processor, it triggers a Load event. If

the requested block is not present in the cache, its state goes from an invalid state I to a transient state between invalid and valid (I_V). During this transition the coherence controller performs the action of sending a message requesting the data block needed by the processor. When the requested data block arrives, the cache block will suffer another transition to a valid state V and it will execute the action of writing in cache the data block received.

2.3.4 Notation

When designing the coherence protocol, the four sets that specify its controllers (states, events, transitions and actions) have to be organized in order to make the whole process of constructing it easier for the designer. Next, we will detail two ways of representing a coherence protocol, one of which is used throughout this document.

One way of representing coherence protocol controller behavior consists of using a *state diagram*, such as the one shown in figure 2-2. Each of the circles represents a possible state of the coherence protocol. In this case, three states are considered: *invalid* (I), *shared* (S) and *modified* (M). Each of the arrows in the diagram represents a state transition and they are tagged with the event that triggers them. When the arrow line is solid it means the event has been triggered by its own coherence controller (load or store in figure 2-2). When the line is dashed, the transition occurs because of a message coming from another coherence controller ('others store' in this case). However, this apparently simple representation becomes hard to understand when the number of stable and transient states increases. Besides, it does not provide the possibility to represent clearly the actions that have to be taken for each transition. Although not entirely concise, it is one of the most popular methods of representation because of its simplicity and easiness of understanding.

Many designers prefer to specify coherence protocols using a clearer method, such as the table-based technique [22], which facilitates the representation of the four main sets of a coherence protocol (states, events, transitions and actions). In the example given in table 2-2, a simplified version of a coherence controller specification using this technique is shown. The states are the

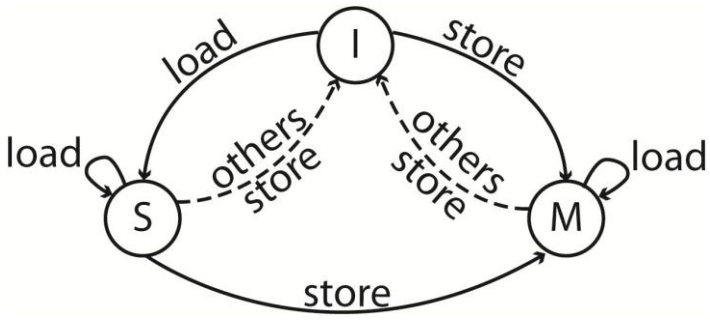

Figure 2-2. Example of a coherence controller specification using a state diagram.

ones shown in the first column of the table. Here again, the three states I, S and M are considered. The events triggered by the coherence controller are represented in the first row of the table: load, store and write-permission request from other controller. The rest of the cells in the table represent a transition from the state in that row to another one, including the actions to be carried out in each of them. Some of the cells also include in the bottom right corner the state to which the cache block will have to be changed to after that transition. If omitted, then no state change is performed.

Throughout this document, this last representation is the one chosen to explain the coherence protocols proposed as it facilitates a higher detail level.

Table 2-2. Simplified example of a coherence controller specification using the table-based technique. Shaded cell indicates a potential erroneous transition (depending on the coherence protocol it may occur or not).

Events / States	Load	Store	Others store (Write permission request from others)
I *(Invalid)*	issue request for read permission → S	issue request for write permission → M	
S *(Read-only)*	send data to core	issue request for write permission → M	losing read permission → I
M *(Read-write)*	send data to core	send data to core	losing read and write permission → I

2.4 Snooping coherence protocols

Snooping protocols can be understood as the first coherence protocols that were commercially used [23]. Their basis is that all cores in the system see the requests from others through a shared medium interconnection. If they do have a copy of the requested block, they perform a set of actions according to the policies of the designed protocol. These executed actions depend on whether the protocol is an *update* type or an *invalidation* type [21]. The former updates every copy of a data block whenever this is written. This means that this approach needs to broadcast

every new written value to every other cache in the system holding a copy of the modified block. The bandwidth and energy requirements for this type of systems become quickly unattainable when the number of processors increases. It is because of this poor scalability that current systems use the second type of protocol design, in which all the copies of a block that will be written have to be invalidated before being able to write it. Thus, the single-writer policy seen in the first section is ensured and only the written blocks are transferred after a subsequent miss or a replacement.

When the number of cores in the system is low, the interconnection network used to connect all of them might be a bus or a similar shared medium technology. In this way, as every request from any core is sent through this bus, the rest of the cores may snoop them and be aware of the actions taken by other coherence controllers. In this type of protocols, the bus interconnection network works as a serialization point for every request sent to the same address, providing a total ordering to all of them.

2.4.1 Baseline snooping protocol in a CMP

Figure 2-3 shows an example of how a simplified snooping protocol works. The diagram shows two private caches (P_0 and P_1) and the last level cache (LLC). The private cache coherence controllers have three different stable states for their data blocks: I, S and M, while the LLC controller only needs two different states: V(alid) and I(nvalid). Initially, the LLC is holding the only copy of the block in a valid state (V). Firstly, P_0 misses a load in its private cache so its controller issues a *GetS* requesting a copy of the block, which travels through the bus. When P_1 controller snoops the request, it just ignores it because it does not have a copy of the block. When LLC sees the request for a block that it has available, it sends a copy to the requestor. When P_0 receives that copy, it writes the value in its L1 cache and its core may proceed with the load. Secondly, P_1 executes a store miss. It sends a *GetX* request through the bus in order to get a copy of the data block and to invalidate the rest of the copies in the system. P_0 controller snoops the *GetX* request for that block which is located in its cache and invalidates it. LLC snoops the same request and sends a copy of the block invalidating its own copy too. After P_1 receives the data block, it finishes the write request. The same process starts again with another load miss at P_0. In this case, the data block is only at P_1 so it is responsible for sending a copy of the block to the requestor. P_1 also has to send a copy of this modified block to the LLC in order to have a valid copy located there for future requests. After this, the block in P_1 changes its state to S, becoming a sharer of the block.

Each of the colors in figure 2-3 represents a memory transaction. One way of simplifying the coherence protocol is to consider each transaction as atomic. This means that if there is a

request for a certain block in flight, there cannot be any other request for the same block until the first one is solved. If we consider that transactions are atomic, the store miss of P_1 may not issue a *GetX* until P_0's request has finalized, i.e. its response is seen on the bus. This premise is easy to fulfill when the interconnection network used is an atomic bus and not straightforward for other types of networks – pipelined (non-atomic bus), split transaction bus or point-to-point networks. For cases where the interconnection network does not act as the ordering point, it is necessary to include some mechanism in the coherence protocol and use the information provided by the transient states. This will prevent races that would appear because processors could receive requests in different order.

Even though using such a simplified protocol would provide the system with the coherence it needs, it is possible to introduce some optimizations in order to obtain better performance. These optimizations come with the addition of the states *exclusive* (E) and *owner* (O) which will change the system's behavior substantially.

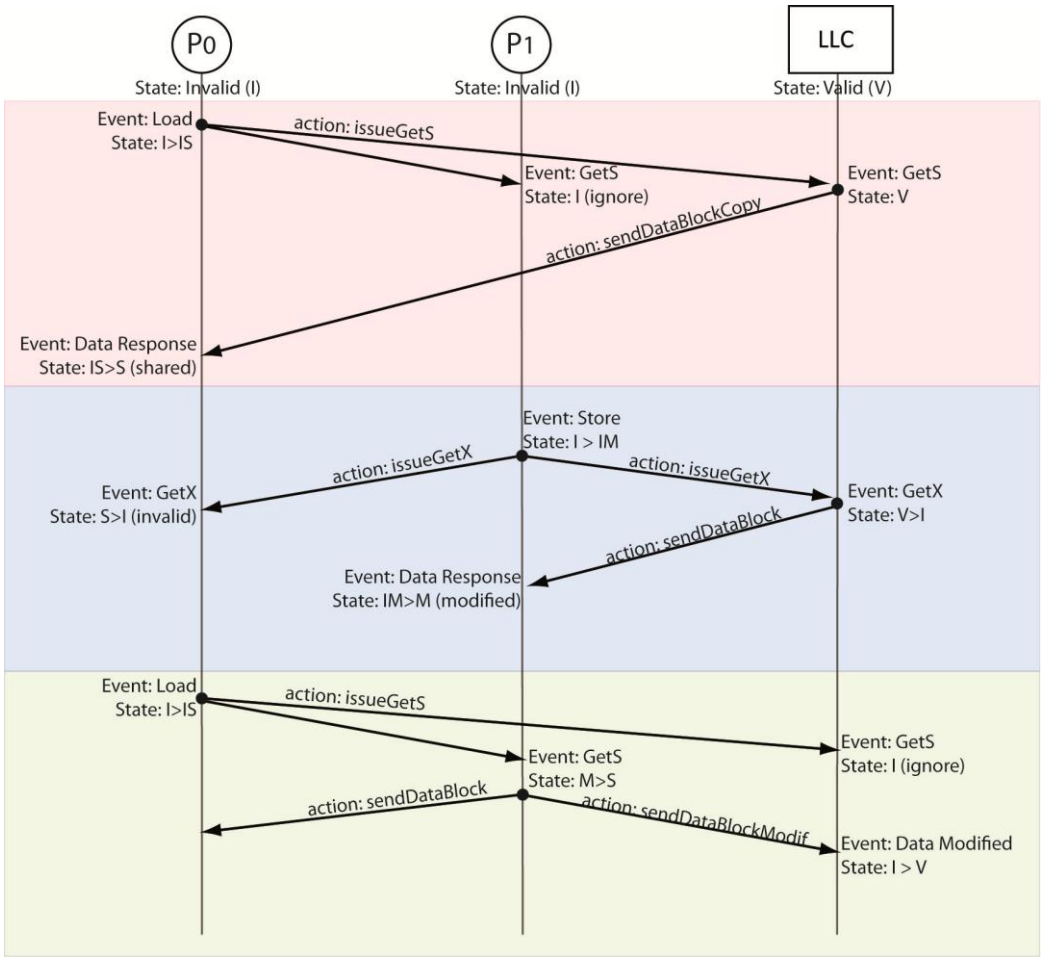

Figure 2-3. Simple MSI snooping protocol example. Shows three different transactions in the coherence protocol for the same data block: a load miss in P_0 processor, a store miss in P_1 processor and another load miss in P_0 after the modification done by P_1.

2.4.1.1 Optimization with Exclusive state

In many important applications, a core frequently first reads a block and then writes it. If the coherence protocol used is a MSI such as the one shown before, all of these situations will imply two misses: the initial read miss (issuing a *GetS* message) and then a write miss, called an upgrade miss, (issuing a *GetX* message). This (common case) upgrade miss occurs even though the requestor is the only sharer of the block. Under these circumstances, and in order to avoid this extra latency problem, it seems interesting to improve the protocol by adding a new state *exclusive* (E).

In the previous example, if P_0 issues its first *GetS* to solve the load miss, and after that it needs write permission for that block, it would need to issue another request, a *GetX*. This would

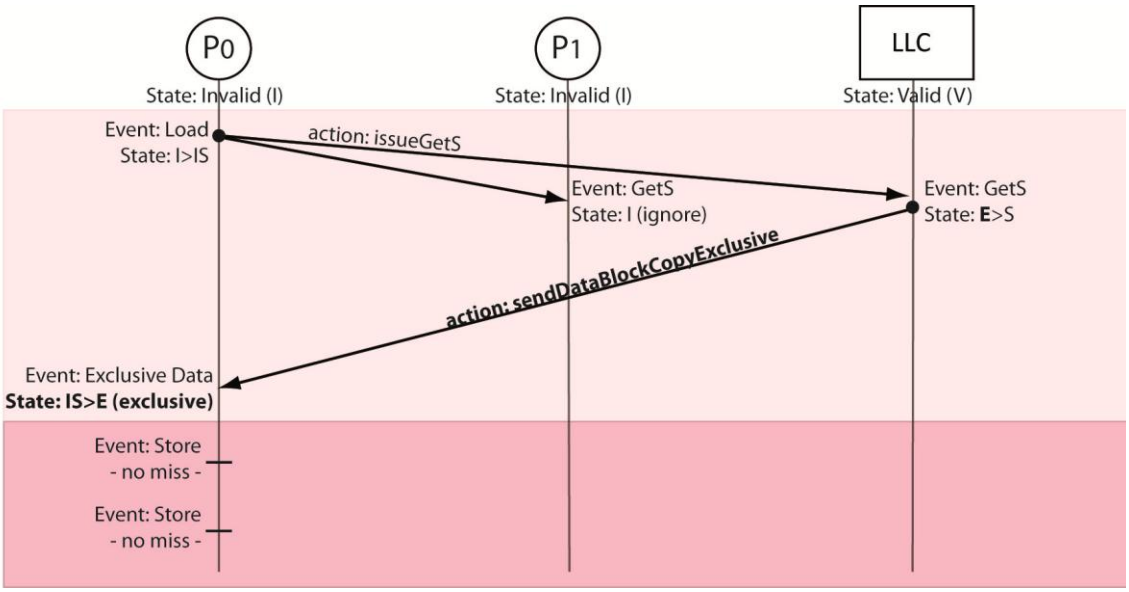

Figure 2-4. Simple MESI snooping protocol example. Shows first a load miss in P_0, and then two consecutive store hits without needing to issue any request for write permission.

invalidate any other copies of the block that in most cases do not even exist as the requestor is the only sharer (like in this case). On the contrary, if the new state *exclusive* (E) is used, this extra message is avoided, because the controller knows that it is the only copy of the block and it can modify it without asking for permission (figure 2-4).

After issuing a read request, in order to know when the coherence controller has to change the block state to S or to E, there are at least two possibilities:

- Modifying the bus signals: wired-OR sharer signal. When a sharer snoops the request, it asserts the *GetS* message so that the requestor knows there are more sharers. [24]

- Adding an extra state in the LLC indicating that there are no more sharers in the private caches. In the previous example, LLC states only indicate whether the data block is valid (V) or invalid (I) with no additional information about the sharers. If LLC knows when there are no sharers, it can send a copy of the data block requested in an exclusive way. So when the requestor receives it, it knows it has to change its state to E (and not to S). This last situation is shown figure 2-4.

2.4.1.2 Optimization with Owner state

Another important optimization that might be added to the MSI protocol is the addition of the Owner (O) state. Its main advantage occurs when a private cache has the block in a *modified* (M) state, i.e. it is the only sharer (it could also be in *exclusive* state (E)) and receives a *GetS* request from another core. In the MSI protocol, a copy of the block must be sent to the requestor and to the LLC controller as the ownership of the block is being transferred to the latter. However, in a MOSI protocol when a cache has a block in M state and receives a *GetS* from another core it keeps the ownership of the block after sending a copy to the requestor. The introduction of the owner state (O) avoids sending an extra update message to the LLC (figure 2-5) and it favors cache-to-cache transfers for the shared data blocks. It also means that the LLC, at least for this reason, does not have to include all the data blocks that are shared in the private caches.

Snooping protocols might be used not only with buses as interconnection network, but with any other type of networks as long as requests are broadcast to all the cores. As will be shown next, there are new coherence protocols based on the snooping coherence concept, but more suitable for cases when cores are connected with an unordered network, such as point-to-point meshes. However, the use of non-totally ordered networks, although it improves the system scalability, will introduce new challenges for the coherence protocol designer.

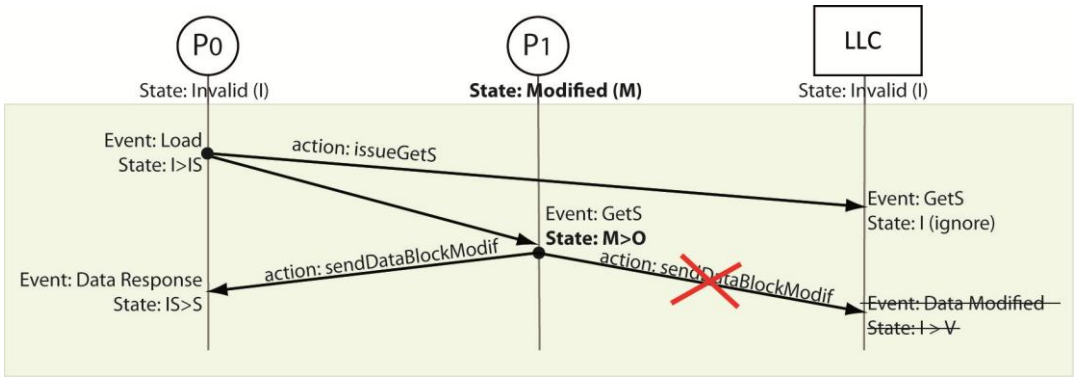

Figure 2-5. Simple MOSI snooping protocol example. Shows a load miss in P_0 and how the ownership stays at P_1 after it receives P_0's request.

2.4.2 Token Coherence

Token Coherence is a type of broadcast-based coherence protocol [17] which differs from the snooping protocols explained in previous sections. It uses *token* counting to ensure that data are read and written coherently and even in unordered networks.

The token counting method consists of assigning a fixed number of T tokens to each of the data blocks in the system. When a processor wants to read a data block, it needs to have a copy of the block with at least one of its tokens. If the processor wants to write in a block, it has to collect all the tokens assigned to that block, ensuring that there is no core either reading or writing. Among all the tokens there is a distinct one called the *owner token*. The core holding the owner token is in charge of replying to other requests with a copy of the data (in a similar way to the owned state optimization). Moreover, when the owner holder needs to replace the data block, it has to send the data value to LLC along with its tokens, while the rest of the token holders just need to replace the tokens they have (without data). By using token counting, token coherence does not need a total ordered interconnection network. Instead, tokens are allowed to move throughout the system as long as these four invariants are always maintained:

- Each block has T tokens in the system. One of them is the owner token.
- Only if a block has all the T tokens may the processor write on it.
- Only if a block holds at least one token and has valid data may the processor read a block.
- If the owner token is sent by a coherence controller, data has to be sent with it.

Token coherence has three (although many others are possible) different performance policies: *TokenB*, *TokenD* and *TokenM*. After a miss, *TokenB* always broadcasts its request to find data directly in any component in the system that might hold a copy. *TokenD* pursues a more efficient bandwidth usage and uses a directory to receive all the private requests. If it does not include the data block requested, then it broadcasts the request to the rest of the components. Finally, *TokenM* tries to combine the low latency of *TokenB* with the bandwidth efficiency of *TokenD* by including information about the sharers next to each entry in the directory. Thus, when the data block is found, the invalidation does not have to be broadcast, but only sent to the current sharers of the block. Among the three of them, the *TokenB* mechanism is the one that provides better performance results and for this reason we will focus our attention on it.

Figure 2-6 shows an example of how *TokenB* works in a system with 4 processors. In this case, the maximum number of tokens is set to 4, as there are 4 processors, i.e. a core needs at least one token to read a block and it needs to collect 4 tokens to write in that data block.

Initially, the LLC controller is the owner of a specific block so it holds the two tokens including the owner. P_0 and P_3 have a copy of that block with one token each. P_1 processor is the one missing in a load and it has to broadcast a *GetS* message requesting a copy of the data block. The owner of the data block is responsible for replying to this request, which in that moment is LLC and so other coherence controllers will just ignore the request (whether they have a copy or not). TokenB favors cache-to-cache transfers by forcing LLC to send all the tokens it has to the private cache levels. Thus, if the data block came with all the tokens, the requestor would know that it had the block in an *exclusive* state and would be able to modify it. However, in our example, LLC sends the copy of the data block with only the two tokens it has (including the owner) and so P_1 automatically becomes the owner after receiving them. In this way, subsequent requests to the same block will find the owner in the private levels reaching it faster without needing to wait for LLC to respond.

After this transaction finishes, P_2 also misses with a load request and needs to broadcast another

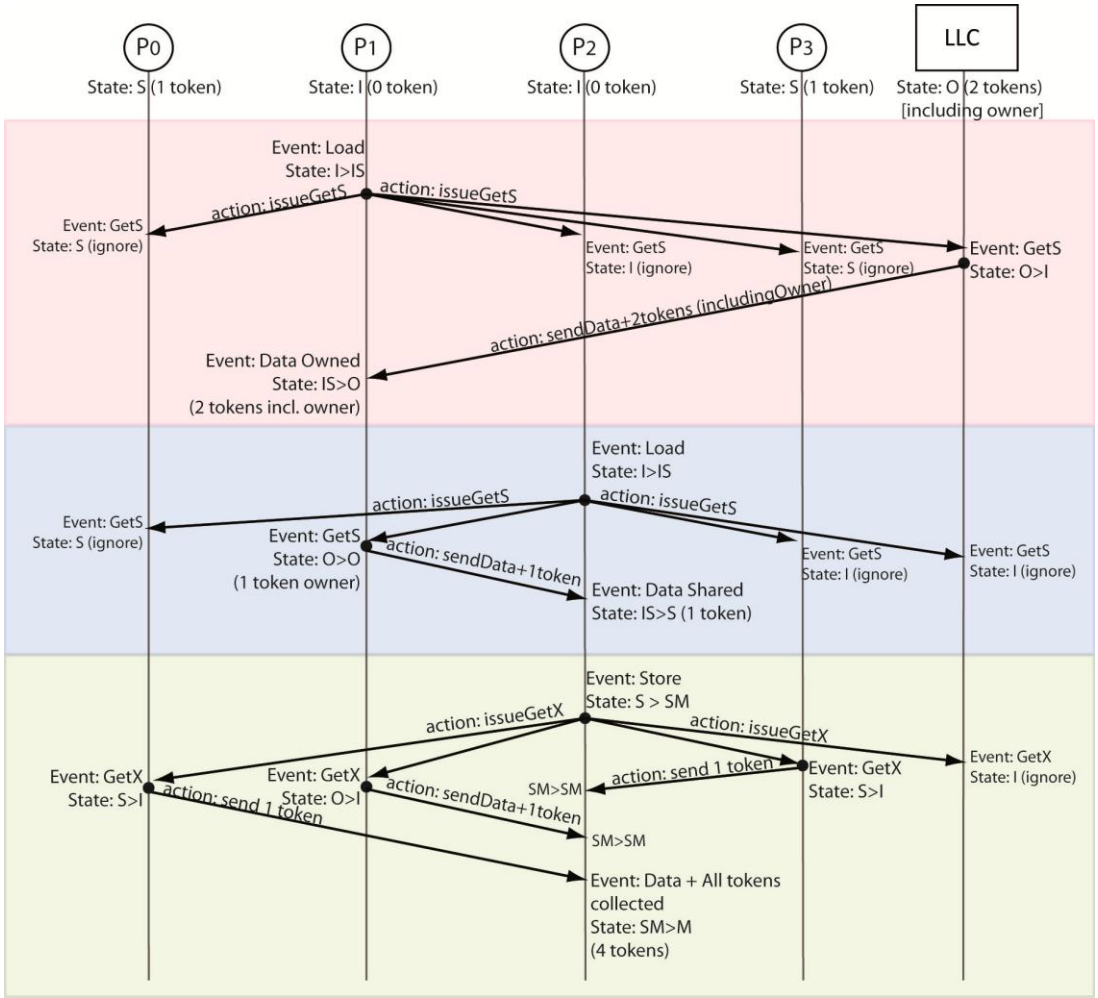

Figure 2-6. TokenB coherence protocol example with two load misses from different processors and a store miss that collects all the tokens for the requested block.

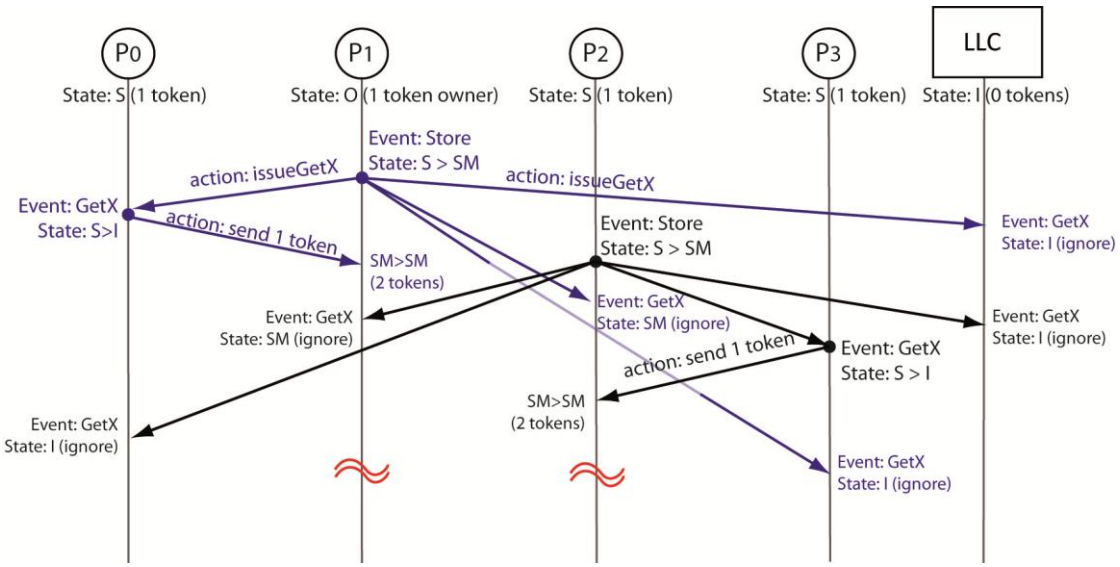

Figure 2-7. Example of a deadlock situation due to two simultaneous requests of the same address.

GetS message. This time P_1, as the owner, is responsible for solving the request, and the rest of the coherence controllers, including LLC, will ignore the request. Finally, P_2 wants to write in the same block, which it has in a shared state, so it needs to send a broadcast *GetX* asking for the rest of the tokens assigned to the data block. This time, all the coherence controllers with a copy of the block will forward their tokens to the requestor. The owner token (P_1) is in charge of sending the copy of the data block in which P_2 will write in.

The main advantage of token-based protocols is that they separate *correctness* from *performance*. The existence of tokens guarantees the invariants mentioned in section 2.1 independently of the token interchange mechanism used. However, the main problem is the fully distributed nature of the system, because *races* may appear. For example, these will occur when simultaneous requests for the same address are broadcast and the tokens needed to perform each operation in each processor do not suffice (for example two *GetX*). In this case, the pending operations will starve, deadlocking the system. Token coherence includes a mechanism based on *persistent requests* to ensure that every read and write request succeeds in order to have a starvation-free system. When a coherence controller detects that it might be starving, it issues a persistent request which tries to solve the starvation situation by broadcasting that information to the rest of the cores. All the nodes in the system remember all the activated persistent requests and they forward all the tokens of the requested block (the ones they hold and the ones that may come in the future). This continues until the requestor collects sufficient tokens to deal with the request and deactivates its persistent request informing all the cores.

In figure 2-7, both P_1 and P_2 issue a *GetX* request trying to collect all the tokens from the same data block. It is possible to see how both requests only get to collect half of the needed tokens (2

tokens) and both ignore the request from each other, because both have pending operations. This originates a starvation situation which will trigger the persistent request method.

When the timeout threshold established is reached, both coherence controllers will broadcast a persistent request notifying that their request was not finished (figure 2-8). It is mandatory to include some priority ordering. For example, if we statically consider that the coherence controller P_1 has higher priority than P_2, P_2 does not ignore the persistent request and it forwards its tokens. When P_1 completes its request, it will broadcast the deactivation of its persistent request. Although not shown in the figure, then P_2's persistent request will become active and the rest of the coherence controllers will forward their tokens.

This starvation problem could also occur when the broadcast request coincides with another operation such as an eviction. For example, in figure 2-9, P_2 has the data block with all the tokens, but needs to replace it. If, at the same time, P_3 processor misses after a store request and broadcasts a *GetX* message, the broadcast message might be ignored by all the components in the system: LLC will not have received the data block with the tokens yet and P_2 has already sent the data block with the tokens so it does not have it anymore. P_3 will wait for a response until the specific *timeout* and if no response comes, it will issue a persistent request. This type of request will find the missed data block in LLC. Notice that this is a simplified example, but after the timeout, data could be anywhere in the system because of another request or the previous situation may be repeated again if LLC has to replace the data block.

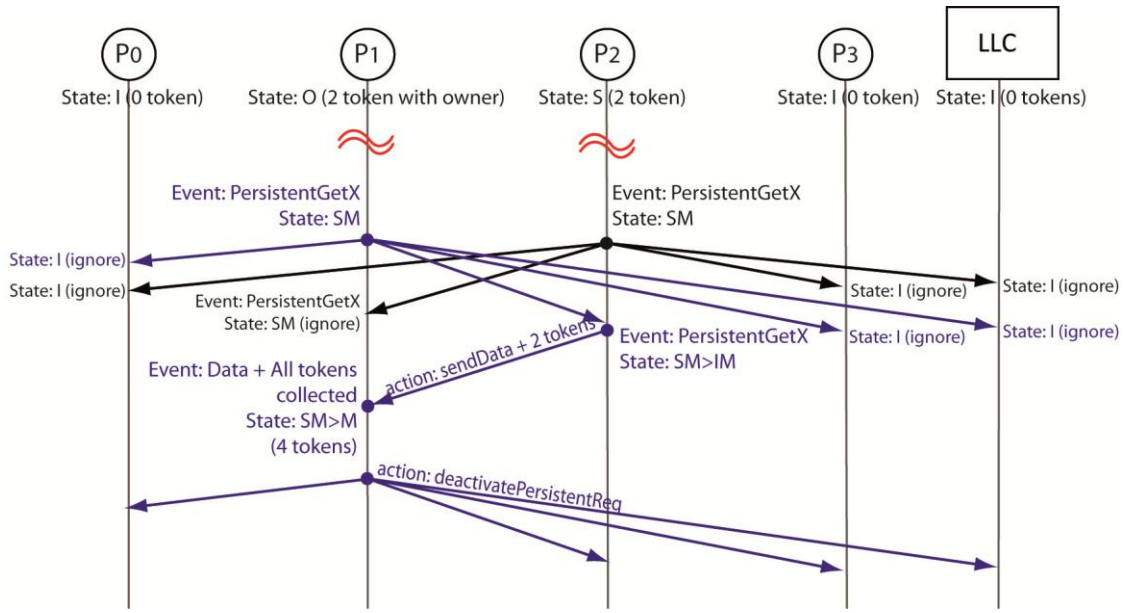

Figure 2-8. Example of a persistent request triggering issued by two coherence controllers in a deadlock situation.

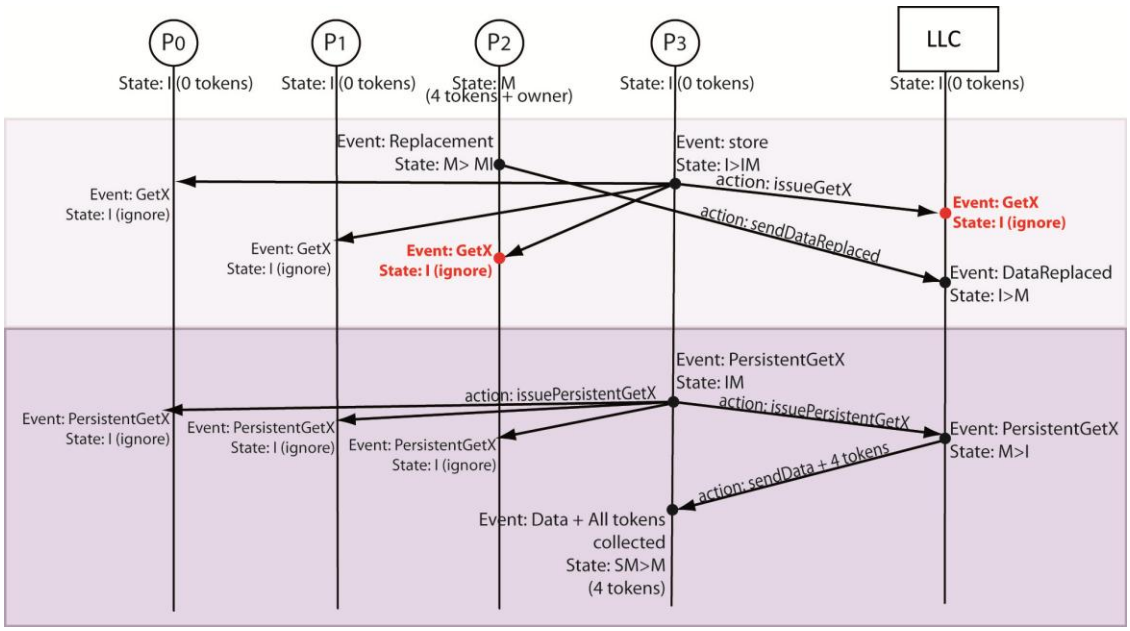

Figure 2-9. Example of a persistent request triggering situation in a TokenB coherence protocol due to a data block replacement and a request issued simultaneously.

The problems related to this mechanism will be discussed in detail in chapter 4, but it should be noted that they could affect the whole system stability since the timeout may lead to the coherence controllers making incorrect assumptions. The controllers might consider that a deadlock situation, like the ones described previously, might happen and so they will issue persistent requests in order to deal with them. However, if the interconnection network or the memory controllers are highly contended and it is for this reason that requests are not being dealt with, the coherence controllers themselves might participate in the creation of more traffic, thus increasing network and memory controllers' contention even more.

Nevertheless, Token Coherence offers a very elegant solution in order to use a snooping protocol with point-to-point interconnection networks, achieving low latency and high performance in optimal situations.

However, as the number of cores increases, the snooping protocol methodology of broadcasting every request causes negative effects in the system. Under specific circumstances, the interconnection network might become a bottleneck because of all the requests that are being sent. For this reason, it is necessary to maintain systems coherent with some other solutions that are more bandwidth conservative and offer more scalability, such as the directory-based coherence protocols introduced next.

2.5 Directory coherence protocols

The alternative developed to overcome the limitations of snoopy protocols is the directory-based coherence protocol. In this type of protocol there is a structure working as a directory with information about the blocks kept in the private caches. This information varies depending on the design of the protocol, but basically it specifies which private caches hold a copy of the block and, if the block is dirty (written), which cache has the modified copy. Therefore, broadcasts are removed and instead, requests are unicast and sent to the directory.

2.5.1 Baseline directory protocols

When a private cache misses a request, it sends a request message to the directory. Depending on the information that the directory has for the requested block, it will proceed accordingly. In every request there are typically two steps (a unicast request followed by a unicast response) or three steps (a unicast request, K forwarded requests and K responses (where $K \geq 1$ and K=number of sharers).

Figure 2-10 shows four of the most common situations of a MSI directory-based coherence protocol. It is important to notice that now the directory state indicates the situation of the block in the private caches: *shared* (S), *modified* (M) or not *present* (I). The rest of the coherence controllers maintain the state of the block they have allocated in the cache.

After a load miss in P_0, it issues a unicast request to the directory. The directory's state indicates there are no sharers for that block (state I) so it just sends a copy of the data block. The way the directory obtains the data block depends on the specific implementation, but to simplify the example, we will consider for now that it is immediate. After sending a copy of the block, the directory allocates a new entry for that address information. It adds P_0 as a new sharer and becomes the owner of the block. After this transaction, P_1 issues a *GetX* after a store miss. When the directory receives its request, it has to send an invalidation message to all the sharers of the block (in this case the only one is P_0) in order to let P_1 modify it. The directory also sends the requestor a copy of the data block and the number of invalidation acknowledgements that it has to wait for before writing in the block. After this request, the directory has to change its state to M, to indicate that the block is modified in one of the private caches and change the ownership of the block to this last write requestor. P_0 invalidates its copy and sends an acknowledgement to P_1 which will finish its request after receiving it.

After the invalidation, if P_0 needs the data block again for another load, it issues another *GetS* request to the directory. This time, the directory is not the owner of the data block, so it forwards the request to the actual owner, P_1, which is the only one with an updated copy of the

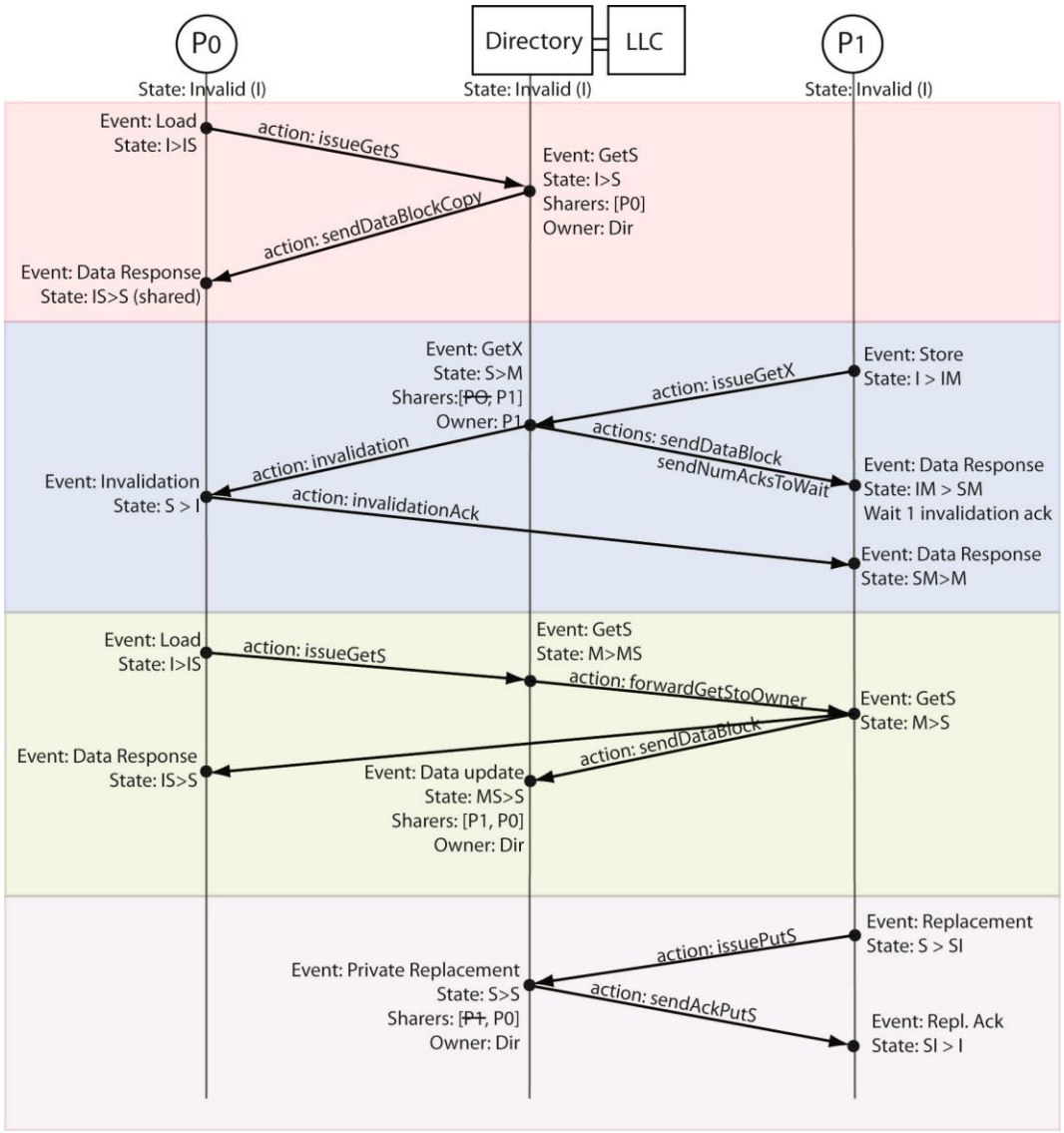

Figure 2-10. Simple MSI directory protocol example. Shows four different transactions in the coherence protocol for the same data block: a load miss in processor P_0, a store miss in processor P_1, another load miss in P_0 after the modification done by P_1 and a data block replacement by P_1.

block. P_1 sends a copy to both, the directory and the requestor and becomes another sharer of the block passing its ownership to the directory.

In this type of protocols, silent evictions are not possible as the directory needs to have accurate information about the block sharers. The last transaction shown in the example in figure 2-10 corresponds to a replacement of the shared block allocated in P_1. It sends a *PutS* request to the directory, asking for permission to replace it. The directory removes P_1 from the sharer list and acknowledges the replacement. If the replaced block has been modified, the *Put* request should include the data being replaced.

In the same way as the snoopy protocols were optimized with the *exclusive* state and the *owner* state, the directory-based ones may also be modified to improve the block access time.

2.5.1.1 Optimization with the Exclusive and Owner states

When adding the *exclusive* state, private caches may silently write in a block when they are the only sharer. This situation causes the directory to be unable to ensure whether the block is dirty or not, because the block could have been requested initially for a load and it could have been written without informing. The owner state allows a private cache with a block in *modified* state (M) to send a copy of the data block to a *GetS* requestor without writing back the modified value to the LLC, and so maintaining the ownership of the block.

The use of the *exclusive* (E) and *owned* (O) states is shown in figure 2-11, which corresponds to the same transactions as in figure 2-10 but with the new states. Initially, P_0 will become the only sharer of the block after its *GetS* so it receives the data block in an exclusive state. Thus, even though it issued a read request, it could modify the block silently. Now when the directory receives the *GetX* request from P_1, the directory is not able to know whether P_0 has written in the block or not, so it has to forward the request to the exclusive sharer. This exclusive sharer will send a valid copy of the data block to the requestor.

There is another difference when P_0 issues its second *GetS*. This request is forwarded to the owner as before, but when P_1 receives it, it will only send a copy of the block to the requestor, but not to the directory, which will only change its state to O indicating that there is more than one sharer for the block and that one of these sharers is the owner. It will keep the ownership assigned to P_1 to forward any future requests.

Lastly, if P_1 maintains the ownership of the block, it is also responsible for updating the modified value. This will be done when it has to replace the block. For this reason, it has to attach the modified data block along with a specific *PutO* request transferring the ownership to the directory.

2.5.2 Directory organization

After reviewing the main aspects of directory behavior, it is also important to analyze its organization. Up to now, it has been assumed that in the directory there is space for all the coherence information needed. However, this is not a realistic assumption, because as the number of cores increases, it becomes quickly unfeasible from a hardware point of view to hold the information about all their private caches. Next, we will analyze what is included in each of the entries of the directory, mainly how the sharers are represented, and the different possibilities when designing the directory.

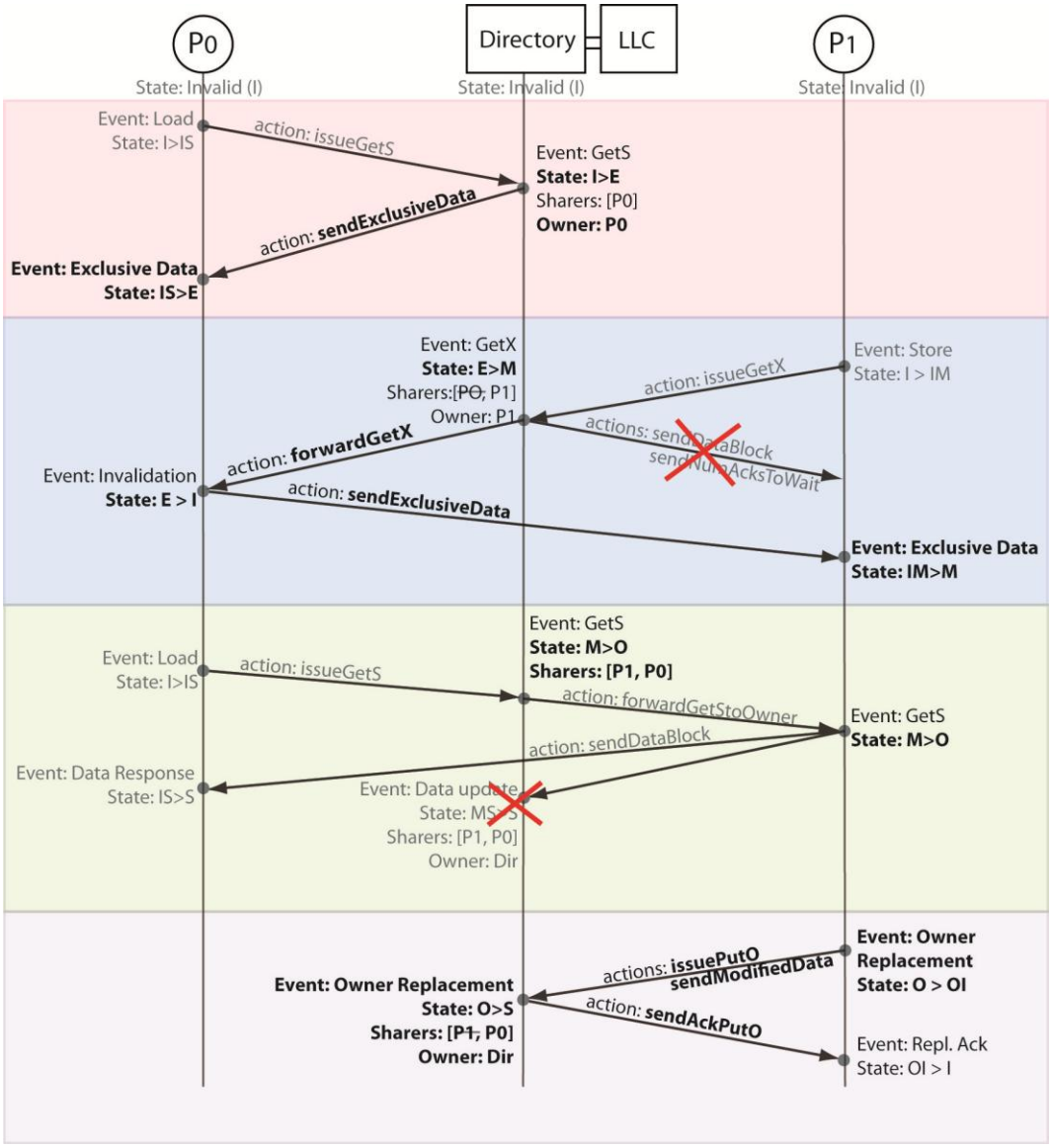

Figure 2-11. Simple MOESI directory protocol example. Shows four transactions in the coherence protocol for the same data block: a load miss in processor P_0, a store miss in processor P_1, another load miss in P_0 after the modification done by P_1 and a data block replacement by P_1. The modifications due to the addition of the *exclusive* and *owner* states are highlighted in bold.

2.5.2.1 Sharers representation

The information associated with a certain address may vary from one coherence protocol to another, because each of them will need specific fields in order to work correctly. However, a directory will always need to have three main basic fields: the block state, the owner of the data block and the sharers of the block. The way of representing these sharers will be an important decision to make when implementing a directory-based coherence protocol, because the space dedicated to this purpose might mean a significant cost overhead. When there is a limited number of private caches, a full-map bit vector may be used [25][26]. This means that for each

of the entries in the directory, there will be a set of P bits, where P is the number of cores in the system.

Nevertheless, this method does not scale as the number of processors increases. For example, a multiprocessor system with 8 cores needs 8 additional bits per address tagged. Considering a directory with 1024 entries, the sharers dedicated space is 8KB (8bits x 1024 entries). If the multiprocessor size increases to a 1024 cores, maintaining a 1024-entry directory (which would be unpractical) the total sharer space will be 1MB (1024 bits × 1024 entries). For this reason, it is necessary to represent the increasing number of sharers, providing the same information but using fewer bits. There are two options to achieve this: to have a coarse directory or a limited pointer directory.

In a *coarse* directory [27] each bit represents a group of private caches. When a bit is set, it means that one or more of the private caches in that group holds a copy. Using the previous 1024-core example, it would be possible to use 16 bits per entry if each of the bits, instead of being associated with one core, represents 64 cores of the whole system. However, this solution implies that when a block needs to be invalidated, there will be extra invalidation messages sent to some caches that may not have the block. This has two negative implications: an increase in bandwidth usage and added protocol complexity.

The *limited pointer* directory is based on a common case observed in parallel applications. Usually a block is shared by a few private caches or by all of them. So, instead of having a complete bit vector, the sharers are defined by using pointers. Each pointer will need $\log_2 P$ bits to point to the right private cache holding the block copy. In a multiprocessor with 1024 cores, to include 3 pointers in each entry will mean 30 bits (10 bits each). However, this representation needs to add some additional mechanism to handle situations in which there are more sharers than those the limited vector is prepared for. Options could include broadcasting whenever there are more sharers than pointers available or invalidating one of the sharers to free space for a new one. A thorough review of the state-of-the-art of the sharer organization will be presented in the next chapter.

2.5.2.2 Directory designs

The possibilities of how to organize a directory are not finite. Originally, the directory controller was integrated with the memory controller, having an entry per existing cache line in the address space. To decrease its latency and the power overhead entailed by each access, designers started to use a separate directory cache structure with a subset of blocks being tracked inside, leaving the rest of the directory information in DRAM [26][25]. However, even with this directory cache, each access entails an unacceptably high latency especially in current

CMPs where the cost of each off-chip access is very high compared to any on-chip access and the bandwidth-wall is always present.

For this reason, in today's CMPs the trend is to introduce as much coherence information inside the chip as possible in order to avoid the off-chip bandwidth limitations and to take advantage of the large LLC sizes inside the chip that keep on increasing. Although in the next chapter the most novel proposals of directory organizations will be presented, it is important to bear in mind two important directory implementations in CMPs.

2.5.2.2.1 LLC in-cache directory

One of the possibilities consists of adding its necessary coherence information to each of the blocks in the LLC. For this directory implementation to work, any existing data block in the lower levels of the hierarchy needs to be allocated in the LLC so it can hold its sharing information. This introduces a new property that has to be fulfilled called *inclusiveness*. The inclusion property indicates that any data block in the private levels of the hierarchy must be present in the upper level (LLC). Moreover, if a block is not present in the upper level, it cannot be present in the lower ones. So, when a request misses in the LLC, it can be assumed that the data block is not in any of the private levels. This inclusiveness property that has to be met by this organization causes three main drawbacks: first, whenever the LLC needs to replace a data block, it is forced to send invalidation requests, called *Recall* messages, to all the existing copies in the private levels; second, the space dedicated in each entry to tracking the sharers must be present in the LLC independently of the block being shared or not; and finally, the aggregate size of the cache is lower if the inclusiveness condition has to be fulfilled.

However this additional storage and the inclusiveness property facilitate the coherence protocol designer's work notably, because the LLC has a real knowledge of the situation of all the blocks cached in the private levels and it knows exactly when it has to access off-chip memory.

2.5.2.2.2 Stand-alone directory

In order to remove the inclusiveness property and the extra storage overhead from the LLC, an especially dedicated structure for directory information can be introduced next to the LLC. This stand-alone directory might be designed to include all the tags of the blocks allocated in the private levels. To achieve this, instead of using the directory state with the sharers of a block, it duplicates all the existing tags in the private caches. Therefore, in order to hold all those tags, the directory needs to have as many sets as there are in all the private caches and its associativity has to be the associativity of the private levels times the number of cores in the system (figure 2-12).

This duplicate-tag directory has a significant implementation cost, especially because of the large associativity needed. It could be suitable for small scale CMPs with limited private cache capacity such as Niagara [28], but for large scale or large private caches the cost becomes unsustainable. This happens because, as the CMP includes more cores, the directory associativity will grow linearly with the core count (adding as many ways per core as the private cache associativity).

In order to limit this large associativity, the duplicate-tag directory may be modified and reduced by assuming that the worst-case scenario will not occur, i.e. all the tags of all the private caches will not be different. Using the example in figure 2-12, the reduced directory would have an associativity A (with A< [C cores*2-ways]), not permitting more than A entries that map to a given cache set allocated in the chip. As there is not a specific place for each of the tags present in the private levels, it is necessary to define which sharers have a copy of that block for each tag in the directory, using a solution like the ones described in section 2.5.2.1 (bit-vector, pointers…). When the directory receives a request for a block whose tag is not present in the directory (which means it is not in any of the private levels) and there are no free entries available, the directory controller needs to replace one of the existing tags, invalidating all its sharers with a *Recall* message like the ones used in the in-cache directories. However, these *Recall* messages do not have the same impact in the system, because they do not mean a replacement in the LLC and so any following request for that data block might find it in the LLC and not in off-chip memory, in a similar way to the in-cache design. Nevertheless, if the size of the directory is not well chosen, these *Recall* requests could be too frequent and make performance suffer because the invalidations remove data that are being used by the cores. A rule of thumb proposed is to establish a directory cache size to cover at least twice the number of cached lines in the private levels [29] in order to prevent unacceptable performance loss. Even so, pathological cases could arise.

	Core 0		Core 1			Core C-1	
Set 0	Set 0 Way 0	Set 0 Way 1	Set 0 Way 0	Set 0 Way 1	…	Set 0 Way 0	Set 0 Way 1
Set 1	Set 1 Way 0	Set 1 Way 1	Set 1 Way 0	Set 1 Way 1	…	Set 1 Way 0	Set 1 Way 1
Set 2	Set 2 Way 0	Set 2 Way 1	Set 2 Way 0	Set 2 Way 1	…	Set 2 Way 0	Set 2 Way 1
Set 3	Set 3 Way 0	Set 3 Way 1	Set 3 Way 0	Set 3 Way 1	…	Set 3 Way 0	Set 3 Way 1
	⋮	⋮	⋮	⋮	⋱	⋮	⋮
Set S-1	Set S-1 Way 0	Set S-1 Way 1	Set S-1 Way 0	Set S-1 Way 1	…	Set S-1 Way 0	Set S-1 Way 1

Figure 2-12. Duplicate-tag directory representation for a C core system with 2-way private levels and S sets.

In any case, even though directory-based coherence protocols were created to overcome the existing problems in snooping protocols, especially concerning their scalability, the truth is that when considering a large number of cores in the system directories, they are not trouble-free. Next, we will finish this chapter with a brief analysis of the main advantages and disadvantages of both types of protocols.

2.6 Qualitative Comparison

Apparently, of the two options seen, the snoopy coherence protocols seem much more simple, manageable and cost-effective than the directories, because of their distributed nature. They do not require any centralized structure, but instead all the nodes in the system have to contribute so that the whole system works correctly. However, in some cases this might increase the difficulties in designing the protocol, because the number of cases that the controller has to handle is much higher. The complexity is even higher especially if the interconnection network used does not naturally provide request ordering (like it occurs in a bus).

Another advantage of this type of protocols is indirection avoidance, since every request is sent to the rest of the cores in the system and not to a centralized structure. This favors cache-to-cache message transfers and avoids sending a unicast request to a directory, which in some cases might be far away from the requestor. However, when the number of processors connected to the bus network is medium-to-large, this acts against the snoopy protocols. All the broadcasts issued by the cores in the system may saturate the interconnection network, which in fact is the main reason for their lack of scalability.

During recent years, some alternatives have appeared based on this type of protocols but trying to overcome their limitations, such as the Token Coherence protocol presented in this chapter. This one does not need a centralized interconnection network like a bus and it can work in any distributed networks, such as meshes or torus. Besides, it has to be remembered that a suitable in-network multicast traffic management is key for avoiding the negative impact that it might have on the system's performance or energy requirements.

On the other hand, directory-based protocols completely solve the excessive traffic problems that snoopy protocols have, because requests are unicast and the directory is the only component in the system which has to solve the problem.

However, this type of protocols moves the scalability problem to the storage needed in the directory where all the coherence information is kept. On the one hand, an important issue is sharer support. As the number of processors in the system grows, the more sharers a data block will have, which increases the difficulties of tracking all of them. On the other hand, the

aggregate cache capacity of the whole system grows as the number of processors and their private caches increase. This means that the *inclusiveness* property that has to be fulfilled by the directory, to ensure that all the private blocks are being tracked, directly affects its size (both in the in-cache and stand-alone designs).

Lastly, after seeing how both types of protocols perform it is reasonable to think that neither option is better than the other. It all depends on the characteristics of the system to which the designer wants to add coherence to. Additionally, and most importantly, the possibility of having hybrid coherence protocols that try to exploit the best characteristics of both types should not be excluded.

2.7 Interconnection networks and coherence

Although the interconnection network and the coherence protocols might seem to be two separate concepts, the fact is that they have a close interrelation. In fact, on many occasions the coherence protocol requires the network to have specific characteristics in order to achieve a correct system. Moreover, the interconnection network can provide an additional support to the coherence protocol adding functionalities to improve the whole system performance.

One of the most important points in common between the network and the coherence protocol is the necessity for message-dependent deadlock avoidance. Even though a network can be considered anomaly free [30] as it is prepared to prevent deadlock, starvation and livelock situations, to ensure correctness of the whole system, it has to guarantee the complete avoidance of this type of deadlock. This situation occurs when there are resources shared among different message types which have some dependencies among them. This situation can be dealt with by adding different physical networks for each type of message [25]. However, this solution becomes too costly and inefficient when the number of message classes is larger and so solutions such as the use of virtual networks are more cost-effective [26][31].

Another example of how the interconnection network can help the coherence protocol so that better performance results are obtained is seen when on-network multicast support is included [32]. This consists of adding the functionality to the routers so that multicast messages are only replicated at the routers where it is necessary in order to reach different destinations. This way, any multicast message is initially sent as a unicast one and when it reaches a router where it would need to take more than one path to reach all its destinations, it replicates. Similar mechanism can be introduced to work in reverse direction, i.e. gathering multiple messages that go to the same destination into only one [33]. As will be shown in the next chapter, these simple mechanisms significantly decrease the average link utilization, decreasing the energy

requirements, especially in snooping coherence protocols where the use of broadcast traffic is common.

Finally, the coherence protocols often need to have the interconnection network to deliver its messages in order, i.e. two messages sent from the same point to the same destination have to be consumed in the same order. A network using non-deterministic routing will not fulfill this protocol requirement, which may result in incorrect behavior of the system.

Summarizing, the collaboration between the interconnection network and the coherence protocol is necessary to obtain a proper function of the whole system, as well as to achieve performance improvements which would not be possible without the contribution from both components.

In the next chapter, before introducing the two proposals of coherence protocols provide in this thesis, we will analyze the state-of-the-art of cache coherence. We will review how coherence has been handled until now. We will also analyze the necessities and the obstacles that have appeared, while describing some of the most relevant proposals offered at present. Lastly, we will forecast the future of cache coherence in CMPs.

Chapter 3. State of the art of coherence

Cache coherence has been a major concern since the first multiprocessor systems. Nowadays multiprocessors always include cache memories within their hierarchy in order to reduce the average latency when executing load and stores and the global traffic to and from memory. However, the existence of private levels induces the appearance of coherence problems since cores do not have the same point of view of the whole additional shared memory space. Current mainstream solutions add specialized hardware to the system, which takes care of the cache coherence problems, discharging the programmer of the duty. This dedicated hardware has been always aimed at performance optimization of the whole memory hierarchy; from the addition of special bits which give specific information about the private levels' data to the implementation of structures with a particular target.

Throughout this chapter, we will provide a general overview of previous work that is most relevant for the development of this thesis. First, we will see the beginnings of cache and the first solutions for coherence issues (section 3.1). Second, we will analyze the main problems that cache coherence protocol designers face nowadays and how their concerns are being solved (section 3.2). Lastly, we will forecast the possible evolution of different aspects of coherence (section 3.3).

3.1 Cache coherence in the past

As was mentioned in the last chapter, from the beginning there were two clear ideas of how to solve the coherence problem: with messages traveling through a shared-bus that is observed by all controllers; or with implementations independent of the network, with a centralized structure where the necessary information to maintain coherency is stored. In the first type, the limited scalability of the shared-bus is the main obstacle to be overcome as it becomes a bottleneck. The only way to do so is by limiting the listening controller's use of that bus. Initially the introduction of cache memories made it possible to reduce the overall traffic by allowing each processor to access its own private cache. Replacement policies were also modified. Instead of updating the main memory each time a block is modified (write-through), only when the block had to be removed from the private cache was the new value in memory updated (write-back).

Write Once Protocol [34], proposed by James R. Goodman, was the first snoopy cache coherence protocol. This is classified as a Write Invalidate protocol since all other caches must invalidate their copies of a block before it can be modified by any other single processor. This protocol was devised for single-board computers based on Intel Multibus [35]. Memory blocks

could be in four different states: *valid* (may be *shared*), *modified*, *exclusive* and *invalid*. New systems such as Synapse N+1 [36] doubled the number of buses to increase the available bandwidth and to be able to introduce 28 tightly-coupled processors of the 80s. Another novelty of this design was the inclusion of a single-bit tag in each cache block of main memory to distinguish when it had to reply to a miss in that block, avoiding possible race conditions.

Another protocol created for a RISC multiprocessor, known as *Berkeley protocol* [37], included two important developments regarding past work: cache-to-cache transfers and the avoidance of updating a block in memory when it was going to be shared between multiple caches. The *Illinois protocol* [38] began to use the source of the requested data to determine the data status in the other caches. Thus, if a data block came from memory, it was assumed that no more copies of the same data were in other caches. If instead the data block came from another cache, it was a shared block. This information (preamble of the exclusive state) significantly improved system performance since invalidations of write hits on unmodified private blocks could be entirely avoided.

The *Firefly protocol* [39] (for the experimental DEC Firefly multiprocessor) was the first scheme in which multiple caches were allowed to contain writeable blocks. When modifying a non-shared block, the protocol followed a write-back strategy. For shared data, an update strategy was performed and the new written value had to be sent to memory, instead of having to send invalidations. The system had a special bus line (*SharedLine*) used to detect sharing copies. When the caches that shared the block snooped the new data value on the bus, they activated the *SharedLine*. Thus, the initiator processor knew that subsequent writes in that block needed to be broadcast. If instead, none of the caches activated the *SharedLine*, the data became exclusive, and any value could be updated in memory when it was victimized. The alternative to Firefly was the multiprocessor *Dragon* [40] whose operation was the same as the above also allowing multiple writers without invalidations, except for the difference that a new written value was not sent to main memory but was only sent to other caches with a copy of that block. This implied the need to add a new state to indicate that although a block was shared its value had to be updated in main memory. These solutions permitting multiple writers have better performance than any invalidating option, but it is always at the expense of greater bandwidth and energy requirements.

During this period, snoopy protocols, such as the ones mentioned above, became more attractive. However, directory-based protocols seem an appealing solution when thinking about larger systems than those used so far. As was mentioned in the previous chapter, the main advantage with this type of protocols is that the location of all copies in the system is known so there is no need to send a broadcast asking all the nodes in the system whether they have the

data. During the early years of directory-based protocols, an important restriction was to forbid having any modified data in several caches without updating it in main memory [41]. In this scheme a central directory is kept with an entry for each block in main memory. By using a set of commands between the caches and the directory, whenever a cache modified the state of its blocks, the directory updated its information. Thus, it could know at any time which blocks were shared or private. When receiving a request, the directory checked whether any cache had changed the block and if so, it first updated the block in main memory and then deleted it from the caches before responding to the received request. If it was not modified in any of the caches, the directory invalidated all the copies sending the data block to the requestor. In [42] a similar directory organization was proposed although it added some filters to avoid unnecessary invalidation messages. It included a *private* flag to each cache block so its holding cache could know that it was the only one with that block (similar to the *exclusive* state). It also added, to each main memory block, as many *present* flags as possible caches that could hold the block. Thus, it was possible to know which caches had a valid copy of each block. Lastly, a *modified* flag was included in each main memory block to know when the content of the block in main memory was different from the copies present in the caches. These last two flags, allowed the update of the value in main-memory to be delayed (i.e. avoiding the write-through mode).

Already in the early days of directory-based protocols, the amount of storage needed to maintain the information about all the copies was starting to be a problem, especially because it was proportional to the size of the main memory. For this reason, successive works proposing reductions of the directory size began to emerge. For example, in [43] each memory block used 2 bits to maintain 4 different states: no copies of the block, unmodified block in a cache, unmodified block in an unknown number of caches, and modified block in exactly one cache. This bit reduction meant that the coherence protocol needed to use broadcasts to perform either invalidations or write-back requests to retrieve the data because it did not have accurate information about each data block.

The late 90s brought the 64-bit processors with large cache and memory sizes and fast floating-point units, making massively parallel computers the most popular choice. Shared-memory computers were rare and were modest in both number of processors and speed. Their absence was in part due to a widespread belief that neither shared-memory software nor shared-memory hardware, were scalable. Even for shared memory machines, the use of non-sharing memory programming models was recommended [44]. In addition, many existing shared memory programs had low performance on massively parallel systems, because they were written under a naïve model which assumed that all memory references had the same cost. This assumption

was wrong because the remote references required slower communication than local ones, i.e. Non-Uniform Memory Access (NUMA).

However, shared memory offered important advantages such as a *uniform address space* and *referential transparency*. A uniform name space allows the construction of distributed data structures, which facilitates fine-grained sharing and frees de programmer for any resource limits. Referential transparency ensures that addresses and access primitives are identical for both local and remote objects. Trying to take advantage of these benefits, proposals began to appear for both shared-memory models [45] and machines with that structure, such as SGI Challenge [46] based on a split transaction bus.

Directory-based protocols were working as an alternative to the bandwidth limitation that meant using a bus for the interconnection network. However, the complexity of managing races, the increasing number of transient states and the centralization of the structure did not help to improve scalability of this type of protocols. Additionally, the gap between the computing capabilities offered by microprocessors and the ones provided by supercomputers was decreasing, while their advantages in price-performance ratio were increasing. This led to the use of microprocessors as computing engines for multiprocessors. Thus, it was most cost-effective to acquire higher performance with them, instead of developing powerful uniprocessors.

An important turning point was the proposal of the *Dash Multiprocessor* [25]. The coherence protocol was a directory-based one and both the memory and the directory structure were

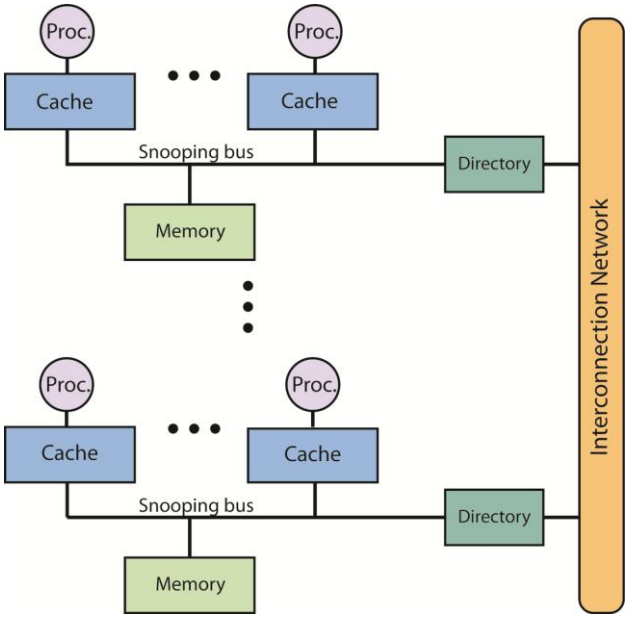

Figure 3-1. The Dash Architecture. The interconnection network connects all the clusters. The directory includes the pointers to the clusters caching each memory line.

distributed avoiding bottlenecks in their accesses and offering more scalability. It was possible to keep on using a single shared memory space and open up the possibility of using generic interconnection networks such as *Omega* [47] or *k-ary n-cubes* [48] used in non-cache coherent machines. As figure 3-1 shows, the multiprocessor was composed of several clusters connected by a snooping bus and all interconnected with a generic network. After a miss in a private cache, the request was sent within the same cluster (local cluster). If it could not be dealt with locally, it was sent to a higher logical level (home cluster) where the directory and the physical memory of the requested address were. Note that Dash already included a two-level memory hierarchy.

A step forward in this direction was the *SGI Origin* implementation [26] whose architecture is shown in figure 3-2. This was also a non-uniform memory access (NUMA) machine, because although the memory was fully addressable, it was distributed among each of the nodes in the system (up to 4GB each). Moreover, each of the nodes contained two processors both connected by a bus. However, this bus was not used as a snoopy bus and requests did not have to be snooped by all the processors in the cluster before being sent to remote levels (in contrast to what the DASH machine did), thus reducing the average latency. SGI Origin had support for the *exclusive* state allowing the read-modify-write accesses. Its protocol also permitted the processor to replace a clean-exclusive cache line without notifying the directory. Additionally, Origin did not need any network ordering and messages were allowed to bypass each other in

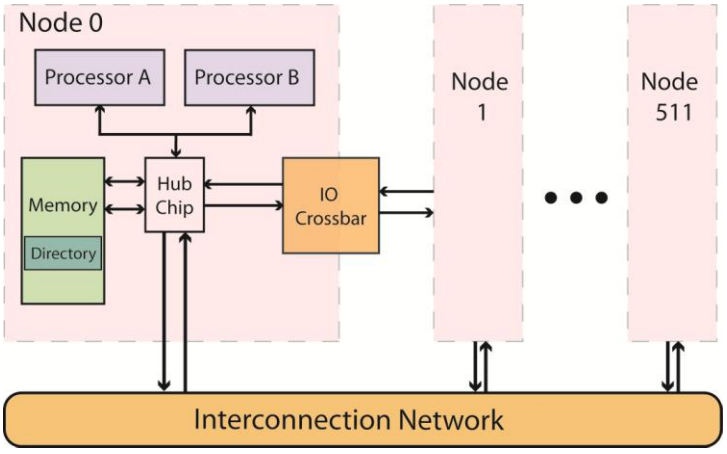

Figure 3-2. The SGI Origin Architecture. Each node includes two processors, up to 4 GB of main memory and the directory. It also includes a portion of the IO subsystem.

the network, because the protocol was able to detect and deal with the out-of-order message deliveries. For this reason, Origin could use adaptive routing, or router micro-architecture improvements such as DAMQ, to deal with network congestion. The directory of this machine stored the sharers in 16 or 64 bits, so it was designed to hold up to 1024 processors; when there

were more than 64 processors, the sharers were stored either with a full-bit vector or a coarse-bit vector, choosing dynamically depending on where the sharers were located.

In parallel with the development of directory-based coherence machines, some interesting and smart solutions were implemented trying to overcome the lack of scalability of the bus. A good example is the Sun Microsystems Starfire E10000 [49]. Sun removed the physical bus and used a scalable network to implement a logical bus. The network was a tree with the processors located at the leaves. When a processor made a request it was sent to the root of the tree from where it was broadcast achieving ordering without requiring a physical bus. This system also added the possibility of allowing the data that did not need ordering to be sent through a crossbar that was connecting all processors, thereby increasing the available bandwidth.

As technology progressed, coherence protocols were modified to obtain optimizations for Symmetric Multiprocessors (SMP). However, as mentioned before, in the mid-90s, the complexity of processor design was increasing as designers were trying to achieve the maximum instruction-level parallelism possible, increasing with it the costs and design time. Commercial applications began to suffer from high memory latencies, so new design alternatives attempting to take advantage of the existence of multiple threads started to appear. This opened two lines of work: simultaneous multithreading (SMT) [50] and chip multiprocessors (CMPs) [51]. The former attempts to maximize the processor resource utilization by executing multiple instructions of multiple threads concurrently whenever it is possible, whereas the latter tries to increase the performance by introducing simple processors in the same chip, trying the take advantage of the thread level parallelism (TLP). In SMT, the first-level cache capacity has to be enlarged, because the amount of data needed by a processor that can execute many threads concurrently can be significant. On the other hand, in CMPs each core is completely independent of the others, so each of them can have its own high-frequency private cache and thus all be accessed in parallel. Since the available on-chip bandwidth can be easily increased, in some cases it is possible to use a write-through policy to maintain the coherence protocol simple.

In the early 2000s, there were some significant works such as the *Compaq Piranha* system [52] or the *IBM Power 4* [53]. Piranha was a prototype that integrated, on a single chip, eight Alpha processors with two levels of memory hierarchy, whose coherence was controlled by a duplicate tag directory. IBM Power 4 was the first commercial non-embedded multicore. It included two cores within the chip and a memory hierarchy with three levels. The two lowest levels were within the chip, L2 being shared between the two cores (physically, three slices connected via a crossbar to the L1s), while the third level, L3, was located off-chip, although the directory was maintained inside the chip.

In the period from 2000 to 2005, as well as a tendency to introduce higher clock rates (through deeper pipelining such as in Pentium 4) and simultaneous multithreading (IBM Power 5), there was a clear movement toward multicore systems following the steps of IBM Power 4, with the *AMD Opteron* [54], *Intel Core-Duo* [55] or *Sun Niagara* [28]. In these early CMPs there were not so many problems with coherence scalability issues since the systems were small. The Opteron used the standard MOESI protocol for cache coherence with a L2 cache acting as a victim cache of L1 (they were mutually exclusive). In the Intel Core Duo case, the snoopy protocol was adopted, for two reasons: (i) it was closer to the design of the single-core Pentium M and (ii) it required less logic (no directory), and therefore less leakage power was dissipated [56]. It included the same MESI protocol as in all other Pentium M processors, but with some optimizations added for faster communication between cores in the same chip (in particular when the data was located in L2). An important modification made was to allow the system to distinguish different situations in which the data were shared by just the two CMP cores or with the rest of the system.

In contrast to the complex Intel Core Duo superscalar processors, the Sun Niagara (figure 3-3) was a CMP which used simple fine-grained multithreaded processors (eight processors and 32 threads per chip). The main reason for this different design is the applications that Niagara was targeted toward: Web and/or database management systems. The interconnection between processors and L2 cache banks was a crossbar switch. The coherence was handled at the L2 level via a duplicate-tag directory scheme. Although this type of directory has a significant cost, as analyzed in the previous chapter, its use is possible thanks to the small size and associativity of the L1 caches (4-way 8KB and 16KB the L1-D and the L1-I respectively) and the reduced

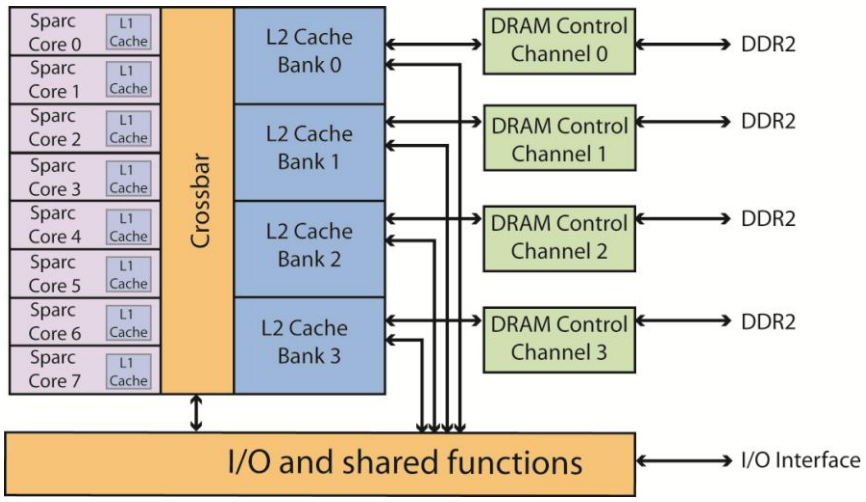

Figure 3-3. The Sun Niagara Architecture. Each SPARC core contains private L1 caches. They share a 4-way banked 3MB L2 cache and they are connected with a crossbar interconnection network.

number of cores in the CMP. After a load miss occurs in the L1 D-cache and a hit in L2, the L1 tag address is entered in the directory and a data block copy is forwarded to the requestor. For a store, the first task is to update the L2 line, then the directory is checked and the corresponding L1 lines are invalidated. The thread that initiated the store can continue without waiting for the coherence actions to take place. In the case of multiple concurrent stores, the updates are delivered to the caches in the same order, thus making sure that transactions are completed in order. L2 is write-back and write–allocate; when a L2 miss occurs, all L1 lines mapping onto the L2 victim are invalidated, thus implementing an inclusive scheme [57].

The technological and market evolution had pushed toward increasing the number of cores per CMP. Although the solutions developed to solve the problems that classic multiprocessor systems could also be used for CMPs, they have some distinctive issues of their own that need to be handled following different approaches. Perhaps, the scarcity of the off-chip bandwidth and the power consumed by the chip can be considered the most important ones. In contrast, CMPs also have positive features: abundance of on-chip bandwidth, which allows a better communication between the different cores and the levels of the memory hierarchy inside the chip.

The debate [58] about the Intel Core or Sun Niagara core types dilemma still remains open, i.e. complex processors or higher number of simple processors. In any case, the complexity of CMPs has grown enough to convert the scalability problems into a matter of numerous studies, with disparate results in many cases. In the next section we briefly analyze the coherence evolution in CMPs over the last few years.

3.2 Cache coherence today (in CMPs)

Future processor chips will contain large numbers of cores and so it would seem that cache coherence might not be scalable under such conditions. Several works suggest that on-chip hardware coherence, as well as shared memories, will not exist in the near future and propose other methods to maintain coherence from a software level. *DeNovo* [14] for instance presents a coherence protocol for an architecture based on a disciplined software model. In the 48-core IA-32-processor [15], shared memory coherence is maintained through software. Data cache lines are modified with a new status bit used to mark the content of the cache line as Message Passing Memory Type (MPMT). This additional bit is determined by the page table information found in the core's TLB which must be setup properly by the operating system. In both of these solutions, the programmer is in charge of managing caches in order to maintain coherence. In the middle ground between these two alternatives, *Cohesion* [59] presents a hybrid memory model which combines hardware and software coherence. There is not a single shared address

space, but instead there are regions where coherence is supported by hardware. [60] propose using the synchronization instructions to maintain coherence instead of relying on the programmer. The readers in the system self-invalidate their blocks using a mechanism called Selective Flushing for which every L1 invalidates any data that is being used when any synchronization point is reached (lock, barrier or wait/signal synchronization).

All these studies support the idea that the coherence hardware is not scalable for future machines. However, some papers such as [16] refute the conventional wisdom that coherence does not scale well to many cores. They show several ways to scale on-chip cache coherence with bounded costs by combining known techniques. This statement is analyzed taking into account different possible problems from the point of view of scalability: *traffic*, *latency*, *storage*, *inclusiveness*, and *energy*. We will use these problems to introduce some of the proposals that have appeared in recent years on each of them, banishing much of the existing reluctance towards coherence hardware scalability.

3.2.1 Traffic and Latency

Traffic and latency are two of the main issues that have to be taken into account when analyzing coherence in current multicores. As the number of processors increases it may seem that the amount of traffic is also increased supra-linearly. However, during recent years, several proposals have appeared trying to reduce latency and global coherence traffic by introducing *interconnection network participation* into the coherence protocol. The first work to implement cache coherence in the network layer was a proposal by Mizrahi et al. [61]. In their work, the entire data cache was migrated into the network routers. However, in the domain of on-chip networks, it is not feasible to cache real data within the network fabric as the access time will critically affect the router pipeline. Thus, in 2006, an *in-network cache coherence* proposal [62] appeared, which adds information to the network routers to construct a virtual coherence tree that connected all the sharers of each of the shared blocks in the system. The idea was similar to Kaxiras and Goodman's GLOW protocol [63]. When a request is travelling through the network to the home node, it may find information about the block without having to reach its destination (reducing the request latency). The data message used to deal with the request constructs a new branch of the virtual tree for further requests from other nodes. Besides the additional hardware needed in the router, which appears to be replicated for every block in more than one router, the in-network proposal only allows one outstanding request per core in order to avoid races. This, for aggressive out-of order cores, does not seem a feasible option. However, the idea of having virtual trees used to connect all the sharers is also used in [64] making it possible to completely avoid broadcast requests and sending them only to the right

nodes. In this case, the trees are constructed based on the sharers of coarse-grained regions rather than for each sharing cache line, so the storage overhead is less than in the in-network coherence proposal. A *region* is a continuous portion of memory comprising a power of two blocks. The idea of regions is that if a block is not being shared among various cores, there is a high probability that the region that the block belongs to is not shared either. Based on this assertion, it is possible to keep the sharers for a region rather than for a cache block without losing excessive accuracy. The usage of regions to reduce bandwidth was also present in *Coarse-Grain Coherence Tracking* [65] and in *RegionScout* [66].

The same trend of adding hardware into the interconnection network in order to reduce the total amount of traffic in the system is followed in the *INCF* proposal [67]. This proposal adds filters inside the elements of the interconnection network to reduce the impact of the broadcasts. By keeping information in each of the routers of the regions that are not shared by any of the cores reachable through each router, it is possible to avoid sending messages and snooping caches that will certainly not have a copy of the block. This means important savings in both bandwidth and power.

As far as these in-network solutions are concerned, they only exploit their characteristics when the data blocks are being shared between cores. With no sharing degree, the amount of traffic used depends on the specific coherence protocol employed, so new solutions to reduce this basic coherence traffic are necessary, especially for snoopy protocols which need a broadcast request for each of the misses in a private cache.

One way of reducing this broadcast traffic comes by varying the granularity of the memory hierarchy blocks. This is what *Amoeba-Cache* does [68], trying to exploit spatial locality. Based on previous work which proposes dynamically modifying block sizes [69], Amoeba-Cache unifies tags and data in the same array and uses two separate bitmaps to distinguish all the words. One bitmap is used to distinguish the tags from the data, and another one is used to know which data is valid and which is not. With this type of cache, the memory utilization is better and therefore the miss rate obtained will be lower, and the bandwidth used (both on-chip and off-chip) will also be less. However, it should be taken into account that such solutions mean an additional complexity in the coherence protocol, making design and verification very difficult. Similar approaches are presented in *Region Tracker* (RT) [70] where its design starts with a conventional cache and replaces the tag array with a structure that facilitates region-level lookups and management at fine-grain level. Its performance results improve previous region works such as *RegionScout* [66] and *Coarse-Grain Coherence Tracking* (CGCT) [65].

PATCH [71] also tries to reduce the total coherence traffic, although employing a different strategy. With a directory-coherence protocol, *PATCH* uses token counting to avoid explicit

acknowledgements and the prediction of request destination to adaptively reduce the bandwidth usage. Any lack of progress in the system is avoided by a mechanism called *token tenure* which uses timeouts to detect when the tokens located at any core should be returned to the home node because another core needs them. However, even though this work's results show less traffic for specific configurations, it does not take into account that the use of timeouts may bring down the interconnection network and degrade the system performance if the timeout value chosen is not adequate, because all the cores could start to return their tokens.

Closely related to the amount of traffic is the latency to finish any transaction and so there are several works attempting to reduce it in very different ways. The system latency is affected by how the systems handle the following four situations: a hit in the private cache; a miss in the private cache that finds the requested data in the LLC (*direct miss*); a miss in the private cache that finds the requested data in another private cache (*indirect miss*); and a miss that means an access to off-chip memory. The coherence protocol has no influence on how long the first situation takes, which means a tag lookup in the private cache to get the requested data. Nor does it have any influence in the fourth situation.

At first glance, the coherence protocol only affects the time it takes to solve the indirect misses, because direct misses are solved by tag lookups in the shared cache. However, the number of direct misses can also have an extraordinary influence on the caches' miss-rate, because reducing the number of times off-chip and direct misses occur obviously reduces the average system latency. Among the numerous works directed to reducing the individual miss latency, those specifically designed for ring-based interconnection networks can be highlighted [72][73][74]. Ring interconnections offer a viable intermediate solution between a clearly non-scalable network and the much more difficult design and verification complexity of other packet-switched networks. Rings use short point-to-point wires with distributed control. They require simple routers with less area and design overhead and have some ordering properties that are exploitable by the coherence protocol. In fact, they are used for example when connecting several IBM Power4 and Power5, in the on-chip interconnect for the IBM/Sony/Toshiba Cell [75], the Scalable Coherence Interconnect (SCI) [76], the Intel Nehalem [77] and subsequently [78][6]. However, given the limitations of this type of interconnection network, the adoption of more complex packet-switched networks seems inevitable to support the increase in the number of cores in forthcoming CMP designs.

The average access latency can also be reduced by lowering the miss rate of the caches. If we avoid replacing data that are being used, subsequent requests for the same data will be dealt with sooner. Modification of the replacement and insertion algorithms is an approach that can decrease this latency and thereby improve the system performance. Two recent proposals in this

sense are *ZCache* [79] and *Cuckoo* [80]. *Zcache* is a cache design based on previous research on skew-associative[1] caches [81] that allows much higher associativity than the number of physical ways. Each way is indexed by a different *hash* function and a cache block can only reside in a single position on each way, corresponding to the hash value of the block's address. Basically, the idea is to increase the number of replacement candidates, but not the number of cache ways. Therefore, hits, which are the most common case, require a single lookup and when a miss occurs, the *zcache* performs a replacement in multiple steps.

The *Cuckoo* proposal is similar, but applied to directory conflicts. Instead of over-provisioning the directory capacity to avoid the impractical highly associative requirements (see section 2.5.2.2.2), the *Cuckoo* directory uses an *N-ary* Cuckoo hash table. This table is a structure with low associativity (3- or 4-way) whose address bits are passed through different hash functions, one for each way (figure 3-4). Its implementation is similar to a set-associative structure. Its lookup operation is identical to the skewed-associative cache, but the main difference is the *insertion* procedure. Whereas the skewed-associative cache selects a victim from one of the ways, the *Cuckoo* organization uses displacement to iteratively move entries until a non-conflicting location is found. There is no discussion about the increase of the protocol complexity when applying this technique.

Finally, to reduce both the total amount of traffic and the average latency, besides adding new

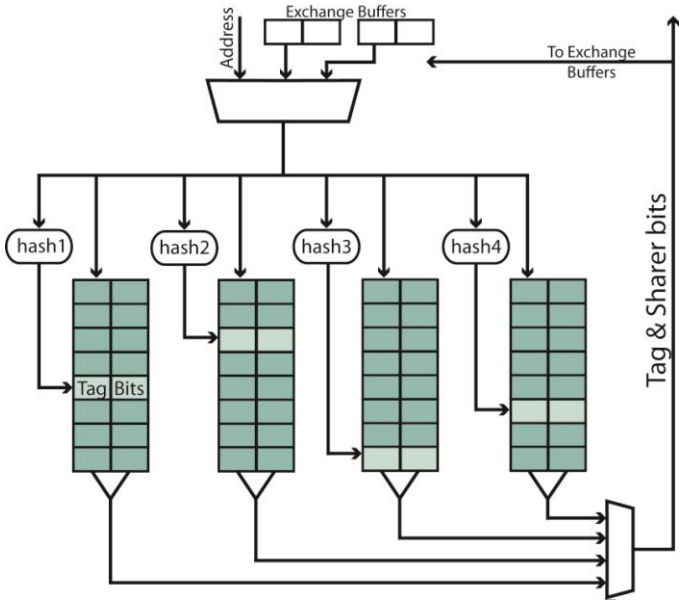

Figure 3-4. Hardware for a 4-way Cuckoo directory.

[1] Skew-associative caches index each way with a different hash function.

features to the coherence protocols and constructing new cache designs, we already mentioned in the previous chapter the importance of mechanisms for handling *multicast traffic*. It is very different to send a multicast message to P different destinations as P unicast messages, than sending only one message which is replicated whenever it has to. The first way means Pindividual messages travelling though the network; the second one initially means only one message that it is replicated as it reaches routers where different paths have to be taken to reach its P destinations. The end-to-end traffic is the same, but the link-traffic is very different. Enright et al. in [32] demonstrated that multi-destination traffic has a serious impact on CMP system performance and the main reason derives from the increased latency of messages. Replicating the messages in the source node causes a waste of bandwidth due to the reiterative resource use of unicast packets that belong to the same multicast message. Moreover, unicast decompositions for multi-destination packets increase the waiting time at their injection queues in each node because of the unavoidable need to sequence the use of the output links. As an example of the enormous difference, figure 3-5 shows the average latency evolution of the two main multicast schemes (path-based and tree-based) compared to the unicast approach for 16 nodes interconnected by a 4-ary 2-cube topology under random traffic with 10% of broadcast messages. The figure shows how, without multicast support, the CMP is able to support about half of traffic. For this reason, this is a fundamental network property and especially necessary in broadcast-based protocols [82].

Closely related to the amount of traffic used is the way the sharers of a block are stored in the system. As more accuracy is achieved in the stored information, fewer messages will be needed

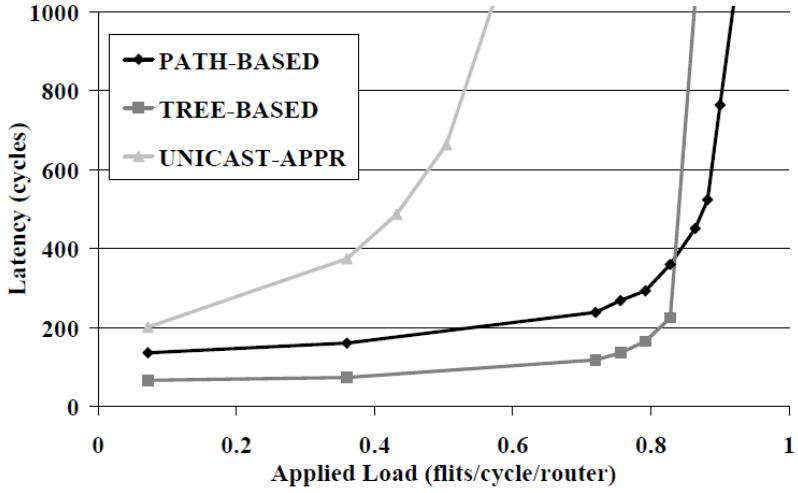

Figure 3-5. Latency evolution for different multicast mechanisms. (Source: [83]**)**

for the coherence protocol reducing the total amount of coherence traffic. However, as will be shown in the next section, the precision of this sharers' information is not trouble-free.

3.2.2 Sharer Tracking

The way all the block sharers are stored is a relevant problem for CMPs. As was mentioned in the previous chapter, there are different structures for maintaining the sharers of a block in the system. The ideal situation is to keep exactly which private caches include a copy of each of the data blocks in the system. However, as the number of cores increases, and so does the number of private caches, it becomes unfeasible to hold all this information with straightforward representations (duplicate-tag directories or full-map directories), because they do not scale with a large number of cores. For these reasons, in recent years, a very active research topic has been to propose new ways of storing the sharers as exactly as possible but in a scalable way.

On the one hand, to save some storage overhead, sharers may be tracked **inexactly**. Besides traditional solutions such as coarse-grain bit-vectors [27] or limited pointers [26], there are other more complex proposals to keep the sharers tracked although in an inexact way. Thus, in *SPACE* [84], it was observed that many memory locations are always accessed by the same processors, i.e. they share the same *sharing pattern*. This means that a large proportion of entries in the directory have similar or the same bit-vectors. In order to take advantage of this situation, *SPACE* includes the used sharing patterns in a separate table and introduces in each of the entries of the LLC a pointer to the correct pattern. Thus, there can be more than one cache line pointing to the same entry in the table, so not needing as many bits. As this table cannot be infinite, *SPACE* dynamically coalesces patterns that are similar to each other to make room for new ones. Moreover, as soon as a pattern is no longer pointed by any cache line, its entry is released for another new pattern. These two actions may lead into two problems. One is that some sharing patterns will cause false positives, i.e. it indicates that a cache is a sharer when it is not. Although *SPACE* tries to merge sharing patterns with the least Hamming distance so that the changes are minimal, false positives will occur anyway. This leads to a higher complexity of the coherence protocol, because the coherence controllers need to be prepared to receive requests for data that they do not have allocated and they need to indicate this to the directory with specific messages. Additionally, bandwidth usage and network contention will increase if false positives occur too often. The second thing that may happen is that, if the table size is not correctly dimensioned, sharing patterns are coalesced too many times and never deleted because they are always pointed to by some cache line. This would mean that the probability of having patterns with all their bits set increases and so the accuracy of the sharing information is completely lost.

Another different approach to track the sharers is the *Tagless Coherence Directory* [85] which removes the tags from the directory by relying in a grid of Bloom Filters [86] to track all the sharers of a block, with one column for each core and one row for each cache set. Thus, instead of having a directory with tags attached to sharer vectors, like a conventional directory does, *Tagless* uses a specific number of hash functions to find out which cores share a copy of the requested address. As happened in *SPACE*, Bloom filters also cause false positives, i.e. the directory says a cache is a sharer when it is not, increasing the protocol complexity. Another problem with Bloom filters is the difficulty to know when an element from the filter has to be removed. *Tagless* directory has one filter per core and set, so to delete the sharer it can use the normal cache eviction information and reevaluate all the hash functions to clear the necessary filter bits. Although the proposal offers a scalability analysis up to 1024 cores, posterior works [80] show that beyond that number of cores, the energy used on each read or update operation becomes too high due to the bit-width, reaching values of the duplicate tag directory.

A work based on both previously mentioned SPACE and Tagless approaches is *SPATL* [87]. As in the Tagless approach, tags within individual sets are combined in a Bloom filter. However, rather than containing sharer vectors, the individual buckets in the bloom filter contain pointers to a table of sharing patterns. As in SPACE, only the sharing patterns actually present due to current access to shared data are represented in the sharing pattern table. This combination enables directory compression with graceful degradation in precision for both inclusive and non-inclusive cache organizations.

Another way of tracking sharers, although sometimes not exact, is to vary the granularity of the data tracked, as *Spatiotemporal Coherence Tracking* (SCT) [88] does. SCT classifies data according to the requests that the cores have sent (read, write or evictions) maintaining different granularity for each of them. For shared data it is better to maintain a fine-grain granularity in order to know the exact sharers with a copy of the data blocks. On the other hand, private data may be detected and grouped in regions using a coarse-granularity to track sharers, reducing the number of entries required in the directory for this type of data.

On the other hand there is the option of attempting to track the sharers **exactly**. To achieve this, the memory hierarchy can be used to track the exact sharers in a *hierarchical* way. If sharers are structured in multiple levels of sparse directories, thousands of cores may be tracked with not a very high storage overhead [89]. There are two main drawbacks in this type of directories: several lookups can be on the critical path (more latency and so worse performance) and the increased complexity of multi-level protocols.

More recently, *SCD* [90] has used the idea of organizing sharers in a hierarchical way, without needing to have multi-level directories. The sharers representation is dynamic depending on

how many sharers the directory tag needs to track. For few sharers, it uses a single-tag limited pointer representation. When there are more sharers than pointers available, it uses a multi-tag format with hierarchical bit-vectors. One entry in the directory is used as a root bit-vector which indicates the sets of the cores that share the line, and as many entries as needed are used as leaf bit-vectors encoding which are the exact sharers of each set of cores. With the use of a highly-associative *zcache* [79] and this method, it is possible to track thousands of sharers without needing thousands of bits per tag in the directory.

3.2.3 Inclusiveness and exclusivenes

At the same time as providing solutions to the problem of how to represent the sharers of any cache block in the private levels, there are also many works focusing on the different designs of the memory hierarchy levels as we will see next.

A system is *inclusive* when higher levels of the cache hierarchy include all the tags and data from lower levels. An *exclusive* system is exactly the opposite: every tag and data in the lower level is not in the higher one and vice versa. The intermediate option is a *non-inclusive* system in which it is not possible to ensure the data that is allocated in each level according to what it is in the others. Figure 3-6 shows a simple representation of the differences among the three designs. In the *inclusive* scheme, after a miss in all the levels of the hierarchy, the data block is allocated in both, LLC and non-LLC ❶. Whenever there is a replacement in LLC, the coherence needs to send a *recall message* invalidating all the copies of the block in order to maintain the inclusion property. Evictions of clean data blocks (not modified) from the non-LLC do not have to be written in the LLC, as they are already allocated there, and only dirty evictions need to be updated. The *non-inclusive* design differs from the inclusive one just in that it is not necessary to send a recall message when the LLC needs to free an entry with a replacement ❷. When the *exclusive* design brings a data block from memory after a miss in all its levels, it only allocates the block in the lowest level, without leaving a copy in LLC ❸ since the block cannot be present in both exclusive levels. As well as not having to send recall

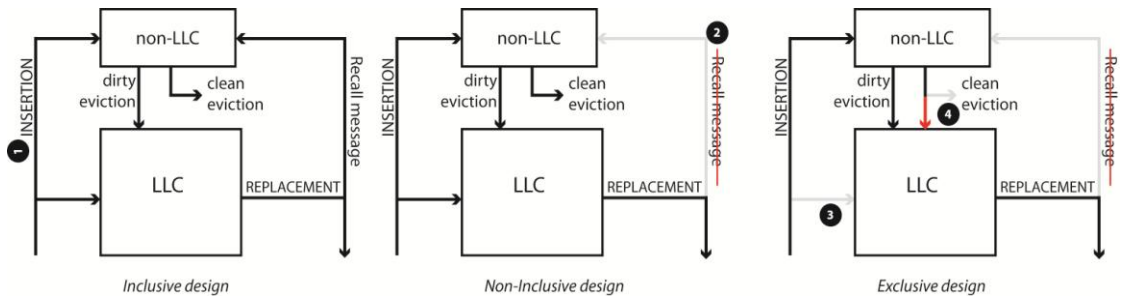

Figure 3-6. Representation of the differences between the inclusive, non-inclusive and exclusive design (light arrows represent unnecessary actions).

messages whenever there is a replacement in LLC, the *exclusive* scheme also needs to allocate in the LLC, not only the dirty evictions like the previous approaches, but also all the clean evictions, since they are not present in LLC when being removed from the non-LLC❹.

In order to analyze the impact that these designs might have on the performance, it is important to consider the size relation between the levels of the cache hierarchy. When the introduction of more transistors onto the chip was used to increase the LLC size, the existing ratio of the non-LLCs to the LLC became lower, i.e. LLC was growing much more than the non-LLC levels. However, with the appearance of CMPs, the number of cores inside the chip has increased and so has the global capacity of the private levels (the sum of all of them). For this reason, the ratio of non-LLC to the LLC has started to increase making the relation between them higher, i.e. non-LLC capacity increases. This behavior can be seen in the graph in figure 3-7. Until 2006, the figure shows that the ratio of cache capacity decreases as the processor design was focusing on increasing the LLC. From 2006 onwards, since the appearance of the first multicore designs, this ratio has stopped decreasing and it has even begun to increase again with the introduction of a L3 cache as LLC.

Of the three options, although *inclusion* makes coherence protocol designer's life easiest, it has least aggregate capacity. This occurs because the global capacity of the system is the same as the highest level which includes the others. For this reason, performance might be severely affected in some situations in which the LLC size is close to the sum of lower levels, i.e. non-LLC/LLC size is close to 1. On the other hand, in an *exclusion* design the global capacity is the sum of all levels and so it has more aggregate capacity. However, this scheme uses more bandwidth than inclusive caches, because of the continuous cache writes of replaced blocks that have to travel from lower to upper levels (both clean and dirty replacements). The *non-inclusive* approach makes better use of the bandwidth available, but obtains higher miss-rate as well as it adds complexity to the coherence protocol design, because the number of cases to be considered is many more than in the previous ones. The three designs are used in some of the levels of

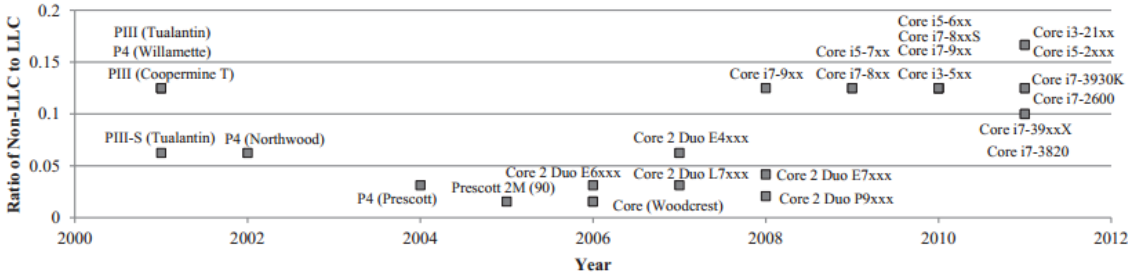

Figure 3-7. Ratio of cache capacity of non-LLCs to the LLC for Intel processors over the past 10 years. (Source: [91]**)**

current CMPs. For example, Intel Nehalem [11] has an inclusive L3 (including everything that is present in privates L1 and L2) while L2 is maintained as a victim cache of L1, both caches becoming exclusive (if a block is present in one of them, it is not in the other). The AMD Phenom II [92] and VIA [93] processors both use exclusive L1 and L2 caches although the former implements a non-inclusive L3 cache and the latter maintains the L3 exclusive too.

Since the three options have advantages and disadvantages, it seems clear that one possibility is to try to combine several of them to achieve their advantages. In the *TLA* approach [94] the inclusive scheme is chosen, but with a modified LLC replacement policy to add non-inclusive characteristics to the hierarchy. The proposal detects the cache blocks allocated in the LLC which are highly accessed by the cores in their private levels, avoiding replacing them in order to maintain the inclusiveness property. The paper suggests three possibilities to know the temporal locality of the blocks in the cache. One consists of sending hints to the LLC to update its replacement state (*Temporal Locality Hints*). Another one is to invalidate lines in the private caches before they become LRU in the LLC, so maintaining the block allocated there. Thus, it is possible to analyze whether the block is re-requested at some other time and so derive from that its temporal locality (*Early Core Invalidation*). The last option is to query the private caches about lines in order to know whether they can be evicted or not (*Query Based Selection*). The first solution means a high usage of the bandwidth available and it might saturate the LLC with coherence messages used to modify its replacement victims. The second solution decreases the bandwidth usage, but invalidates lines according to the access pattern perceived by LLC, which might not match the processors' pattern. This will lead to the invalidation of useful lines and so an increase in the miss ratio in the private caches. The third option improves both previous designs by decreasing the bandwidth usage and avoiding the invalidation of useful lines. However, querying the private caches about their block usage will mean an increase in the control messages the cache will have to manage leading to higher response latency.

The same idea of hints sent to the LLC is used by Chaudhuri et al. in [95], which proposes a replacement algorithm for inclusive and exclusive caches where the private caches analyze the access patterns of their allocated blocks. In an inclusive scheme, private caches may send hints to the LLC, enabling it to have more information about its blocks. This way, the LLC can replace according to the probability of a block being used again by a core. In an exclusive scheme, that pattern information from the private caches may be used to avoid extra writes in the LLC. Thus, when a private cache replaces a block which has a high probability of not being accessed again, it is not sent to the LLC for allocation. This is known as *selective cache bypassing* [96][97]. Although inclusive architectures cannot benefit from this technique, because bypassing inherently breaks the inclusion property, in [98] Gupta et al. propose a

possibility to circumvent this limitation. The LLC includes a bypass buffer and the bypassed cache lines skip the LLC while their tags are stored in it. When a tag is evicted from the bypass buffer, it invalidates the corresponding cache lines in upper level caches to ensure the inclusion property. The key insight is that the lifetime of a bypassed line should be short in upper level caches and it is most likely dead when its tag is evicted from the bypass buffer. Therefore, a small bypass buffer is sufficient to fulfill the inclusion property and to obtain most performance benefits from bypassing.

There are other novel methods which dynamically decide whether to use exclusion or non-inclusion schemes according to the application that is being executed. This is what *FLEXclusion* does [91]. By monitoring traffic, this proposal can select exclusion mode when it is necessary to have more capacity and, in contrast, when it is necessary to decrease the bandwidth usage, it changes to a non-inclusion scheme. Although these methodologies do not increase performance significantly, qualitatively the complexity added to the coherence protocol seems to be non-negligible. In any case, it is important to have new methods to achieve even more scalability in cases where a traditional inclusive or exclusive cache does not seem to be a suitable option.

3.2.4 Energy overheads

One of the most recurrent aspects that appear when discussing future coherence mechanisms in CMP is energy. Like in the majority of current computer architecture works, given the constrained power envelope of a CMP, the energy characterization of any new proposal is fundamental. However, there are some occasions where obtaining this characterization is done by trying to emphasize the proposals themselves, sometimes using quite simplistic models that lead to partial conclusions or even incorrect ones.

The most common pitfall is to analyze the energy consumption of a proposed element isolated from the rest of the system. This means that only the new hardware added in the proposal is analyzed and if the power consumed is lower than the same piece of hardware proposed previously by others for the same task, the conclusion is that it consumes less energy. However, if this comparison is not well done, i.e. with the correct normalized values, usually the conclusions will not be completely accurate. It is important to always bear in mind the power consumed by the whole system while it executes any task, because even though any new element might consume more power, the final energy consumed by the full system might be lower, because the time it takes to finish that same task is reduced.

In any case, almost all relevant works about coherence protocols include energy-efficiency analysis. One way of reducing the energy consumption of a directory-based protocol is to diminish the number of directory accesses (snoop filters) like *Jetty* [99] and *TurboTag* [100] do,

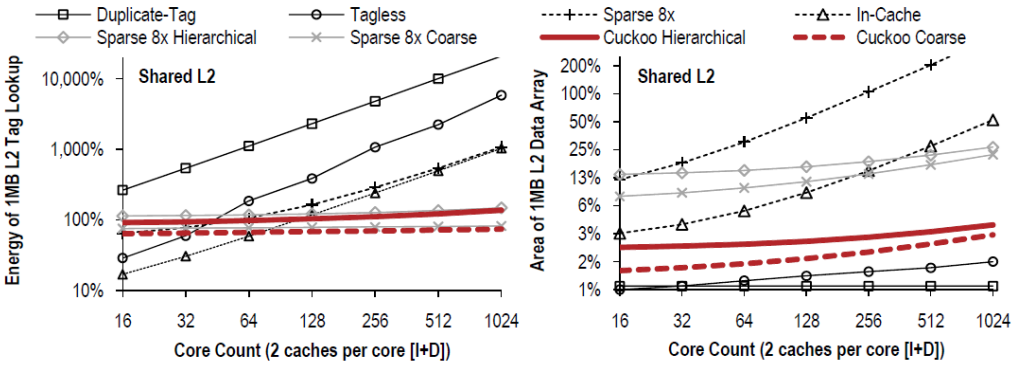

Figure 3-8. Power and area comparison of directory organizations. (Source: [80]**)**

which introduce filtering mechanisms to eliminate unnecessary directory lookups. These needless lookups come from the basic idea that the majority of accesses to the directory find no sharers, because data is not shared among the cores, meaning wasting power if its coherence is checked. There are other similar energy saving proposals like [101] which avoids tracking non-coherent memory blocks, although this scheme needs the operating system collaboration for detecting the private data blocks.

Some scalability comparisons have been done for different directory organizations considering up to a thousand cores [80]. Although their results were obtained under numerous assumptions and simplifications, some interesting conclusions can be highlighted.

Figure 3-8 shows that there are already solutions which scale up to thousands of cores. Moreover, the solutions used up to now, such as the duplicate-tag or in-cache designs, are not suitable, and other hierarchical or sparse solutions are necessary. It is also important to bear in mind that even though proposals could be highly scalable from the area point of view, they are not when power consumption is considered [85].

3.3 Forecasting cache coherence in future CMP

Summarizing the above, the trend of future microprocessor architectures is clear: multicore. Whereas some experts predict processors with a thousand cores or more by the middle of the next decade [102], others have doubts [103]. In any case, an important question that continuously arises is whether current architectures will scale to such high numbers of cores and whether they will be manageable from the programmer's point of view. For some the answer is no, at least with a plain architecture as appears in several works. For others, if several conditions are introduced to overcome the limitations that appear, the answer is yes.

To improve performance when having a large number of cores, it is necessary to address the limitations imposed by the communication between cores and the off-chip memory bandwidth.

When the computation is spread across multiple cores on the chip, the instruction distribution and the communication of intermediate values will increase the execution time, due to latency and communication resource contention. Applications that require a large amount of traffic have to be especially aware of this, because each of their operations ties up many resources and can consume a significant amount of energy. Therefore, even for CMPs with far fewer than a thousand cores, there is a set of necessary conditions that has to be accomplished in order to be able to use hardware coherence in the coming years. Among the most important ones, we could highlight the following.

First, it should be noted that avoiding coherence hardware does not eliminate the problem, but basically it transfers it to the programmer-compiler pair. Obviously there will still be multicore solutions with no cache or shared memory, but they will not be mainstream options. A fact supporting this statement is the evolution of the operating systems; different scalability issues are being solved [104] and even the new multi-kernel proposals are dealt with assuming the existence of hardware coherency [105].

Secondly, one of the aspects that seems essential is the existence of different levels in the subdivision of a large number of cores. Works like [102][106], which analyze the scalability of a thousand nodes, divide the cores into a number of smaller clusters that are interconnected by one or more interconnection networks, minimizing some of the problems that crop up when treating the entire system as one plain device.

Third, it seems clear that such a large number of processors requires either introducing a very large amount of on-chip memory or solving the problem of the bandwidth wall. One possible solution may be brought about by emerging memory technologies (STT-RAM [107], CBRAM [108]...) or using *more-than-Moore* technologies such as 3D Stacking. These technologies will increase the on-chip size by several orders of magnitude. Therefore, to maintain their efficiency it will be necessary to increase the number of levels in the hierarchy, handling their higher complexity.

Fourth, even without taking into account coherence in multi-socket systems, the large amounts of on-chip memory storage will make it impossible to store information about "all" the sharers in a precise way. This will require us to rely on some broadcast-based management when a load or store miss triggers a request with an inaccurate destination.

Fifth, the interconnection network will have characteristics suited to the requirements of a multicore system. This means that they will have appropriate topological properties, including the suitable broadcast management support mentioned before. That is, the interconnection network must minimize the communication cost efficiently handling multicast messages, i.e. no

serialized multiple unicast messages must be generated when sending multicast or broadcast messages.

Sixth, the miss latency for actively shared blocks will be high, but it is important to leave it independent of the network diameter and only dependent on the number of sharers. For this reason, solutions such as ATC-ACKwise [102] are indispensable, which propose storing up to a limited number of sharers in order to know how many cores, but not which cores, have the data stored. Solutions such as the ones proposed in this thesis, based on tokens, also perform the same functionality.

Seventh, it should be noted that although the system grows, the locality of applications will continue to exist and its exploitation will still be essential to improving performance. This suggests finding solutions that use hierarchical protocols that take advantage of this locality. For example, make fast cache-to-cache transfers by using broadcast mechanisms, while remote accesses are made through directory type structures, to avoid flooding the interconnection network with unnecessary messages and wasting power.

Finally, it seems imperative to limit protocol complexity. While it is true that increasing the number of processors does not necessarily mean increasing the protocol complexity, it does increase the time required to verify their correctness. Therefore, it would be convenient to think in terms of hierarchical coherence protocols that can be formally verified with an amount of effort independent of the number of cores [109].

Nevertheless, and considering the commercial tendency, it can be said that CMPs future has a multicore chip landscape, with not a very large number of processors inside the chip, maybe few tens of them, but very powerful such as IBM Power 8 [110] or Intel Skylake. The tendency to introduce even larger amounts of cache inside the chip is also clear. IBM is already introducing eDRAM [4][110] and it seems that Intel is already planning to do the same. In the long term, it will be possible to increase the number of cores inside, reaching hundreds to thousands of cores. Possibly, it will be necessary to organize them in a hierarchical structure as was mentioned before, but the coherence protocols will have to be able to manage such large numbers of cores without becoming an obstacle for the whole system. Bearing all this in mind and believing that hardware cache-coherent systems will be the best design choice for future CMPs, the coherence protocols presented in the next chapter, LOCKE and MOSAIC, each try to solve the problems that may appear in the near future and in the long-term future respectively and benefit from the specific characteristics of each type of system.

Chapter 4. Reactive coherence for medium-scale CMPs: LOCKE

The previous chapter detailed a clear tendency for implementing multicores in the future. However, in the short term, the number of cores that will be introduced inside the chip does not seem to be reaching large-scale values and companies opt for small to medium scale sizes but with more individual power in each core [4][6]. To maintain the coherence of these types of system, their main characteristics should be exploited in order to reduce the global latency of the whole system. One of these characteristics is the high bandwidth availability inside the chip, on the contrary to the limited bandwidth that exists on the off-chip interconnection networks, where it is scarce because of the discrete nature of the communication system elements. The use of scalable point-to-point interconnection networks and the scalable cache hierarchies designs implemented, such as NUCA [111], make this bandwidth profuse inside the chip. If we add to these characteristics the appearance of 3D stacked systems [13] and the utilization of low-swing links [112], the excess in bandwidth is substantially increased and the energy cost of moving data faster is reduced.

For all these reasons, when designing coherence protocols in small to medium size architectures, we will always have to think about using this on-chip network bandwidth availability and try to avoid any extra latency in the form of indirections as much as possible. Currently there are a substantial number of CMP coherence protocol proposals that share this point of view [17][71][113] and most of the ideas use broadcasting as the mechanism to overcome indirection at intermediate ordering points. The impact of the shortcomings that these protocols might have can be much less than is commonly assumed. Namely:

1) *The multicast traffic required for on-chip cache requests will increase network consumption.*

   It is true that power consumption is affected by multicast traffic, but the final effect depends on the network characteristics. As we saw in the previous chapters, if the network has hardware support for multicast messages [32][114], its impact could be reduced because each network resource is used at most once per request. This happens because the message is only replicated when it has to go through different paths to reach its destinations. When no multicast support is included, one message will have to be sent for each of the destinations and so each resource will be used many times. According to [32], using multicast support could save up to 70% in the network Energy Delay Square Product (ED2P).

2) *Excessive network cache bandwidth consumption could increase contention and significantly increase on-chip latency.*

   Although this may potentially ruin the rationale of snoop-based coherence protocols, a correctly dimensioned design for the cache hierarchy capable of decoupling the number of cores and the on-chip cache bandwidth will prevent it. Under these circumstances, on-chip communication bandwidth will scale in proportion to core count and/or its aggressiveness.

3) *Extra cache tag lookups produced in these protocols will increase cache energy consumption.*

   If we take into account the growing leakage in each technological advance [9], the area devoted to cache, and the substantial benefit in terms of performance obtained by snoop-based coherence, the increased tag snoop energy might be quickly amortized by the benefits in static energy.

Under this scenario, the LOCKE coherence protocol is proposed for small to medium architectures. As a starting point it uses the token coherence framework [115] seen in chapter 2, but enhances responsiveness and stability in several ways as will be shown next. LOCKE can establish the position of all the tokens by using explicit acknowledgements for each token movement. Thus, every request will locate either the necessary tokens or a pending acknowledgement. Its requests may be quickly forwarded to the in-flight tokens' destinations, improving the latency especially when accessing contended data. Moreover, LOCKE does not require any starvation avoidance mechanism, such as the persistent request method, since it is a reactive coherence protocol where requests always have information about where to find the requested data.

It might appear that this acknowledgment traffic will increase bandwidth utilization and maybe the added contention could potentially increase network latency or energy consumption, but as will be demonstrated in this chapter, this might not be the case. The effectiveness of the token location mechanism compensates for its extra bandwidth consumption, improving the energy-performance tradeoff of both token coherence and directory-based coherence protocols.

To check LOCKE'S effectiveness we have used a full-system simulator which includes a precise interconnection network simulator along with a wide variety of workloads ranging from multithreaded server and numerical applications to multiprogrammed workloads (Appendix A). On average LOCKE outperforms a conventional directory and a token coherence protocol by 16% and 28% respectively for a 16-core CMP.

The rest of the chapter is organized as follows: section 4.1 focuses on analyzing the responsiveness and the instabilities of token coherence protocol, motivating the necessity of LOCKE coherence protocol. Sections 4.2 and 4.3 will describe the coherence protocol proposal itself with its different methods to solve false and true racing requests, which will be described in sections 4.4 and 4.5. Last, section 4.6 will provide the performance results obtained with LOCKE and demonstrate the improvements obtained.

4.1 Motivation

In order to understand why a novel coherence protocol like LOCKE is attractive, it is important to bear in mind the limitations of the Token Coherence protocol, in which it is based. Although this protocol's main characteristics were reviewed in the coherence protocols chapter (chapter 2), we will study in detail the instabilities that its responsiveness mechanism undergoes under specific situations.

4.1.1 Token Coherence responsiveness

As a reminder for the reader, Token Coherence protocol deals with racing requests by counting tokens. In this way, data races are avoided by forcing different ongoing memory operations to require an incompatible number of tokens. In starvation-prone circumstances, each contending processor eventually issues what is called a *persistent request*, which will statically determine the winner and force the loser or losers to return the tokens to the frontrunner processor. When this one finishes its operation, the next processor obtains the tokens required to perform its pending memory transaction. Under most working conditions racing requests are not frequent, so this serialization will have a negligible impact on performance.

However, many racing requests will come from the synchronization instructions, especially in workloads like multithreaded ones [116] where it is their key operation. The passive approach used by token coherence to resolve this kind of situations, which is limited by the time established to issue the persistent request, could delay synchronization resolution unnecessarily. Additionally, persistent requests not only serialize potential data races, but also address the temporary lack of knowledge about token location. This lack of knowledge arises when some of the tokens required to perform a specific memory transaction are unavailable at the end point of the messages from a broadcast request. For example, this happens when a block is evicted from a cache and a request overtakes the in-flight data block in the interconnection network. In these circumstances, the request will not be fulfilled because it will not reach the needed tokens either at the origin or at the destination (for a specific example, see figure 2-9 in chapter 2). The outcome of this situation is similar to a temporary racing request, denoted from now on as a

false racing request. By contraposition, we denominate the concurrent and simultaneously incompatible operations issued over the same block by different processors as *true racing requests.*

4.1.2 Token Coherence Stability

True and false request races are dealt with using the persistent request method by keeping track of the time involved in each pending memory request. If the time is greater than a fixed threshold, a persistent request is sent. In order to maintain the scalability of the hardware, structures are required to perform persistent requests and to provide a distributed and fair arbitration scheme. Token coherence establishes that only one ongoing persistent request per core is supported. For this reason, to minimize the performance impact that this might have in processors with multiple outstanding memory operations, the original request is reissued one or more times before sending the persistent request. The timeout chosen to trigger this process can be established statically, looking at the on-chip miss access latency, or dynamically, averaging the latency of recent memory transactions. If the time of a particular ongoing memory transaction is above this limit, it seems reasonable to suppose that there might be another core accessing the same block. The request is reissued and if the timeout is once again exceeded then a persistent request is sent.

Although the persistent request mechanism seems to be very simple, contention effects can negatively impact its performance. When applying a significant load on the network, the communication latency of each individual message increases as a result of the unavailability of resources in use by other messages. At medium loads the total latency could increase by a few cycles, but when the load is higher, the effect could be substantially larger. Worst of all, this variation could be highly dependent on the traffic pattern and the applied load, which can vary abruptly throughout the workload execution.

In a low contention situation, network latency is closer to base latency and persistent requests work as expected. Nevertheless, if a spike of traffic suddenly appears, contention increases and so does the latency of all pending memory transactions. If the effect of the contention is over the persistent request timeout, a chain reaction might be triggered. The positive feedback between reissues and persistent request and network contention creates a storm of persistent requests in which almost any memory operation is reissued or even solved by a persistent request. Under this unstable situation, the system performance drops dramatically. To illustrate this phenomenon, we will focus our interest on two particular applications (NUMERICAL and SERVER) running in 16 aggressive out-of-order cores in the CMP such as the ones described in table 4-5. All the parameters of the system, including the network, are correctly dimensioned,

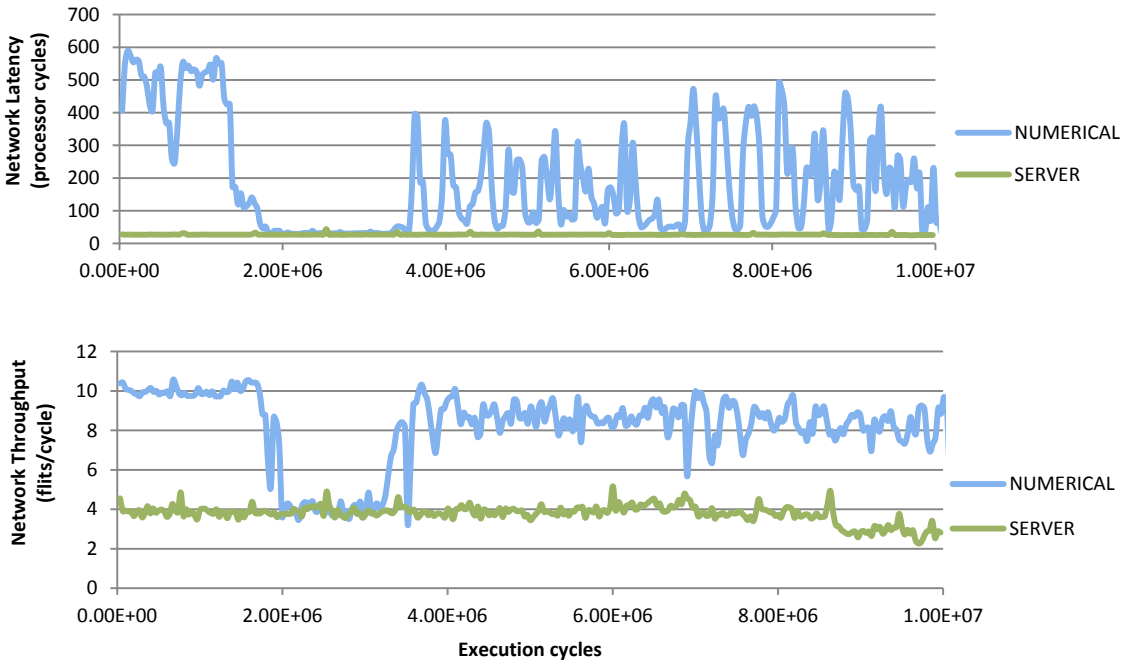

Figure 4-1. Network dynamic evolution with a 16-processor system.
(a) Average latency (includes injection queue delay); (b) Throughput.

i.e. they are chosen in order to obtain an optimal cost/performance ratio over a large set of applications. The sharing degree of the two applications is quite different, in the NUMERICAL it is low and in the SERVER it is high. For this reason, the number of persistent requests in the former should be lower than in the latter. However, for an optimal time-out threshold and one reissue before sending a persistent request, the proportion of memory transactions resolved by persistent request is more than 10% in NUMERICAL and less than 0.1% for SERVER.

This behavior, which apparently seems contradictory according to the sharing degree of each application, may be explained looking at figure 4-1. It shows the network latency (a) and the applied load (b) during 10 million processor cycles for both applications. In contrast to SERVER, the NUMERICAL application is very interconnection-network demanding during short intervals due to the access to highly contended blocks. During these phases, the latency spikes due to on-network contention effects. These effects are exacerbated by the one-to-all traffic pattern of the application. During these spikes, reissue and persistent request frequency increases, not because of true racing requests, but because packets are delayed within the network. This triggers more reissues and persistent requests, which further increase contention. Even using dynamically predicted thresholds, we are unable to capture the sudden variations in latency. In fact, dynamic estimations could accelerate system instabilities even preventing the complete execution of the workload. The described effect is not a rare anomaly and similar behavior can also be observed if off-chip bandwidth is saturated. All in all, without a solution for this problem, choosing this protocol to be used in a general purpose system might be unsafe.

For all these reasons, if we want to use token coherence in these architectures and take advantage of all its good characteristics, it is necessary to have a new coherence protocol. This also includes token counting, but maintains its behavior independent from the interconnection network situation and shielded from any negative contention effect by removing any timeout from its functioning.

4.2 Conceptual approach

LOCKE uses token counting to maintain coherence invariants, but it includes additional characteristics to avoid false racing requests and a smart mechanism to actively resolve true racing requests, making a passive starvation avoidance mechanism unnecessary. In order to do this, LOCKE is based on precise knowledge of where any token is or will be located in the near future. Thus, if the protocol can track all the tokens, no false racing requests are possible. On the other hand, LOCKE solves true racing requests with a starvation-free self-inhibition mechanism that serializes data access of simultaneous incompatible memory transactions.

Before seeing LOCKE behavior in detail, we will review the conceptual approach of its design when a false and a true racing request occur. In a token counting coherence protocol, the tokens assigned to a specific data block can be found either stored in a line in any cache or they can be moving from one place to another, i.e. being sent to a requestor or being replaced from a higher level of the hierarchy. These token movements are the main cause of false racing requests, because of the lack of knowledge about the tokens location during specific time intervals. For this reason, LOCKE requires an acknowledgement to be sent to the token sender so that it knows when the tokens have reached their destination. The second type of racing requests, true racing requests, occur because of simultaneous operations initiated by different processors which are incompatible. For this situation, LOCKE includes a self-managed mechanism to serialize all the incompatible operations without suffering any deadlock.

Figure 4-2 shows a simplified sketch of how LOCKE works when a false racing request occurs in a simplified 3 processor system, each of them with its private cache and a LLC shared among all of them. The initial situation we will consider is P_0 having the data block with all its tokens allocated, i.e. block is in state E (exclusive). First, P_1 issues a read request for that block sendinga broadcast message to the rest of coherence controllers in the system ❶ (messages to the LLC are omitted in the figure, but its controller would work like any other). When the request reaches P_0 where the requested block is, P_0 sends a copy of the data block with one token to P_1 ❷ and holds the line in a specific state that determines that it is waiting for an acknowledgement for that token movement. While that token is going to its destination, P_2

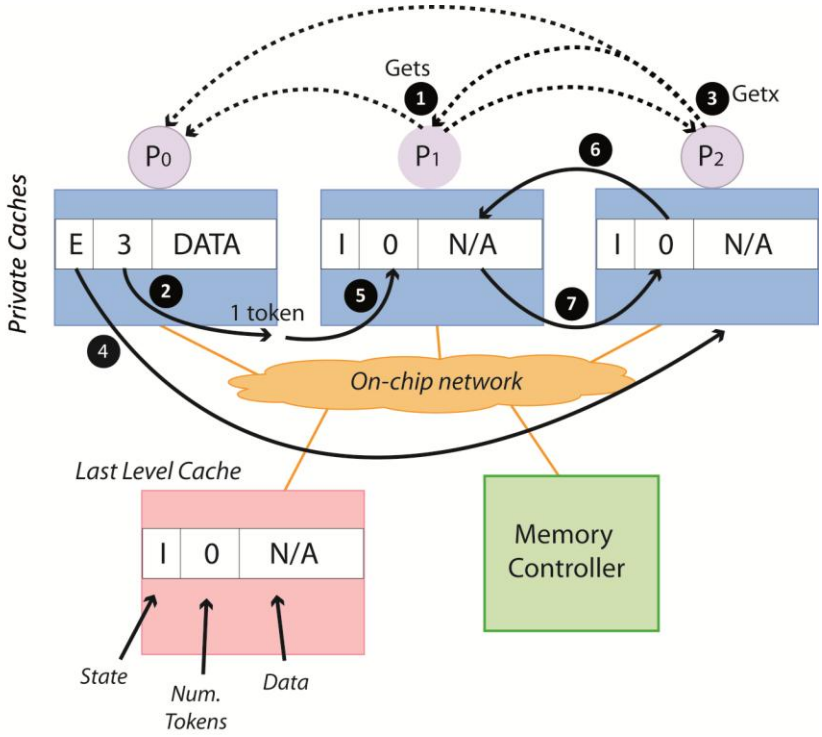

Figure 4-2. Sketch of a *false racing request* handled with LOCKE.

issues a write request, sending another broadcast to the rest of the controllers ❸. When this write request reaches P_1, it has still not received the data block so it ignores it. When it reaches P_0, it replies to the write request sending all the tokens it has (all of them except one) and informs P_2 that it is waiting an acknowledgement message from P_1 ❹ meaning that P_2's request may reach P_1 before it has received the data block (as happens). On the other hand, after all this, P_1 receives the data block with the token ❺ and so it sends a reception acknowledgement message to P_0, completing its read operation. When P_2 receives the pending acknowledgement information from P_1, it sends a special unicast read request to P_1 asking for the lost token ❻. This time, P_1 does not ignore the request because it has the requested data block and so it sends the token to P_2 ❼, invalidating its own block in order for P_2 to be able to finish its write request. This last movement will also wait for the reception acknowledgement message in case there are more requests for that same block and to maintain LOCKE's invariant of always knowing where all the tokens are.

The other situation for which LOCKE has to have some mechanism is the true racing request that might appear for incompatible and simultaneous requests. Figure 4-3 shows a basic sketch for two simultaneous write requests (*GetX*) from P_1 and P_2. We will consider that the initial situation this time will be that P_0 has two of the three tokens of the requested block and the LLC has the remaining one. Both write requests are made at the same time (❶ and ❷), but the request from P_1 arrives first to P_0 and the request from P_2 arrives first to the LLC. P_0 will forward its two tokens to P_1 ❸ and the LLC will forward its token to P_2 ❹. From this moment,

both requesting processors will have a subset of the tokens needed to perform their write operations. If no mechanism to handle the situation was added, the system would enter a deadlock, because none of the private cache controllers would release their tokens until their operations are done. However, LOCKE is able to solve the problem when P_1's write request arrives at P_2 ❺ (we will consider for now that P_1's priority is higher than P_2's). At the very instant that P_2 knows that there is a simultaneous write request, which is incompatible with its pending operation, and that the other write request has more priority that its own, it self-freezes. This means that it forwards all the tokens it has collected up to that moment to P_1 ❻ and it would forward any other token that arrived later. In this specific example, after the token is forwarded from P_2, P_1 can finish its request because it collects the three tokens to finish its write instruction. Although not shown in the example[2], when P_1 finishes its write request, it broadcasts a completion message to indicate to all the possible frozen processors that they can retry their pending requests.

The key point of the whole protocol is that either the tokens or their pending acknowledgement are found by all the requests that are broadcast to the network. This means that no false racing requests will occur, because there will never be a lack of knowledge about when tokens are moving from one cache to another. If a true racing request happens, where the problem is the

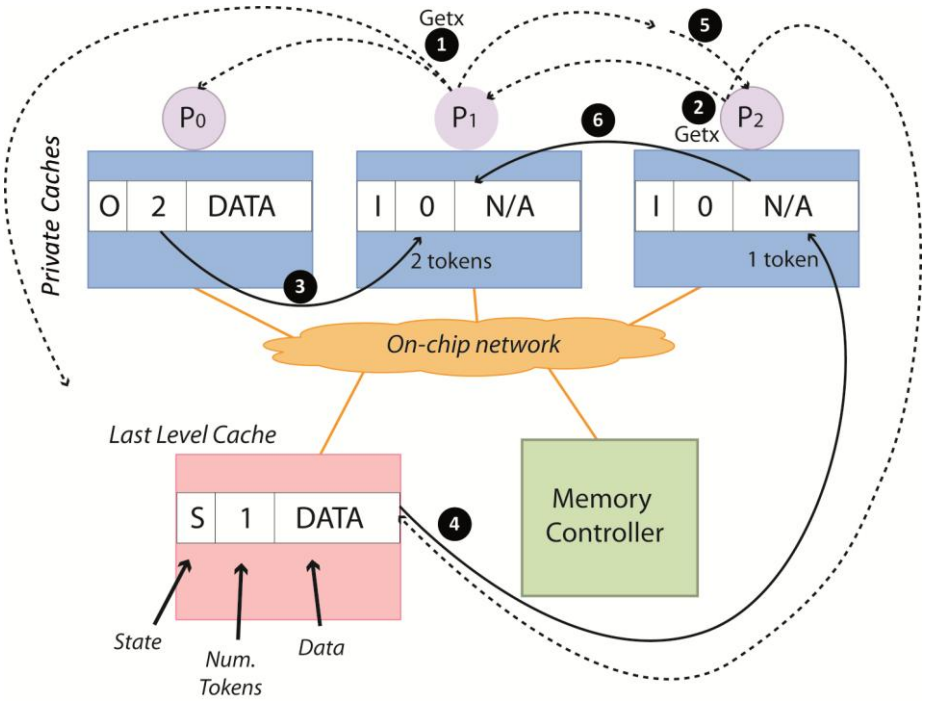

Figure 4-3. Sketch of a *true racing request* handled by LOCKE.

[2] The sketch also omits the acknowledgement messages sent due to the token movements.

incompatibility of several requests, LOCKE includes a mechanism to serialize the controllers and avoid starvation situations.

Obviously, the two sketches represent very specific situations so the reader can understand LOCKE's main behavior. In the following sections, the whole coherence protocol will be described in detail.

4.3 Design details

LOCKE coherence protocol includes the standard stable states of a MOESI protocol, with 3 transient states (IS, IM and SM). It also adds 4 new control states for its characteristic behavior (PS, PO, PX and F). The first three of these are used to indicate when there is an acknowledgement message pending. The F state is needed to manage when there is more than one processor trying to write simultaneously, i.e. when a true racing request occurs. Table 4-1 provides a brief description of each of the states needed in the protocol.

The stable (M, O, E, S and I) and their transient states (IS, IM and SM) maintain their generic meaning as table 4-1 describes (for more details see chapter 2). We will focus on the four control states specific of LOCKE.

Table 4-1. Description of LOCKE states.

States	Description
I	Block not present or invalid
S	Block with shared data including some token/s
O	Block with owned data including some tokens and the owner token is one of them
E	Block with exclusive data including all the tokens
M	Block with modified data including all the tokens
IS	Controller issued a GetS and it is waiting for data
IM	Controller issued a GetX and it is waiting for data
SM	Controller issued a GetX, it is waiting for data and holds some tokens
PS	Controller sent shared data and it is waiting for acknowledgement
PO	Controller sent shared data and it is waiting for acknowledgement, but keeps the owner token
PX	Controller sent data with all the tokens it had and it is waiting for acknowledgement
F	Frozen. Controller has a pending store operation but with less priority than another one

The first three of them (PS, PO and PX) are states used when the controller has sent some kind of data with tokens and it is waiting for an acknowledgement message. The difference between them is the type of data sent and what is kept in its own cache. The PS state indicates that shared data was sent, not leaving any data or token allocated in the origin; the PX state shows that data and all the tokens available were sent (including the owner); the PO state indicates that shared data was sent, but the controller keeps the owner token. This differentiation is made to distinguish when the requests received from others while waiting for an acknowledgement have to be answered or not. The fourth characteristic control state of LOCKE is the F state, for *frozen*.

Table 4-2. Basic events of the private coherence controller

Events	Description
Load	Processor wants to load a block which is allocated in the private cache with read permissions.
Store	Processor wants to store a block which is allocated in the private cache with write permissions.
Replacement	The coherence controller needs to evict the line to make space for a new one.
GetS	Another controller has sent a read request.
GetX	Another controller with less priority than this controller has sent a write request.
FreezeGetX	Another controller with more priority than this controller has sent a write request.
SpecialGetS	Another controller has been told that tokens were coming to this controller and it wants to perform a read operation.
SpecialGetX	Another controller has been told that tokens were coming to this controller and it wants to perform a write operation.
DataShared	Received data block with one token.
DataOwner	Received data block with the token owner (and maybe more tokens, but not all).
DataAllTokens	Received data block with all the tokens.
Ack	Acknowledgement message indicating the tokens sent were received so they do not have to be tracked anymore.
Retry	Message indicating the need to retry our pending request with another controller, because ours arrived while tokens were in movement.
Complete	Complete message of another coherence controller that has finished its write operation.

A line changes its state to F when there is an unfinished write request, but the controller has detected another higher priority write request for the same address.

Using the table-based technique, table 4-4 shows a simplified transition table of the private cache controllers using LOCKE coherence protocol. To help understand the table and as a support for the reader, table 4-2 and table 4-3 describe the events triggered and the actions taken respectively.

Table 4-3. Basic actions of the private coherence controller

Actions	Description
sendGetS	Broadcast a GetS request to all the coherence controllers.
sendGetX	Broadcast a GetX request to all the coherence controllers.
replaceData	Send the data block replaced to the LLC.
send1Token	Send data block with 1 token to a read requestor.
sendAllTokens	Send data block with all the tokens to a write requestor.
update	Update the incoming data block and number of tokens in our cache.
sendAck	Send an acknowledgement message to the sender of a data block with tokens indicating that the tokens have arrived to their destination.
inforTokensDestination	Inform a requestor where the controller has sent the tokens.
inforOwnerDestination	Inform a requestor where the controller has sent the owner token.
sendSpecialGetS	After receiving information about the owner token destination, send a unicast SpecialGetS to that destination.
sendSpecialGetX	After receiving information about tokens' destination(s), send a multicast SpecialGetX to that/those destination(s).
askToRetryToMeLater	Send a message indicating the need to retry the request as a unicast once again.
askToRetryBC	Send a message indicating the need to retry the request as a broadcast once again.
retryWithBoss	Send a message indicating the need to retry the request with the coherence controller that we keep as the boss (writer with most priority).
bounceToBoss	Bounce data block and tokens received to the coherence controller that we maintain as the boss (writer with most priority).
bounceData	Send back the data block and tokens received to the sender.
bounceToL2	Bounce data block and tokens received to the LLC.

Table 4-4. Simplified transitions table for a private cache coherence controller using LOCKE protocol.

Colored cells indicate control actions: stalling the request in green, ignoring the incoming message in blue and an error transition in red.

	Load	Store	Replacement	GetS	GetX	Freeze GetX	Special GetS	Special GetX	Data Shared	Data Owner	Data AllTokens	Ack	Retry	Complete
I	send GetS	send GetX	*False racing requests*				askToRetryBC	askToRetryBC	bounceData	bounceData	bounceData			
S	load	send GetX	replace Data / PS		sendAllTokens / PS	sendAllTokens / PS	askToRetryBC	sendAllTokens / PS	updateData sendAck	updateData sendAck / O	updateData sendAck / M			
O	load	send GetX / SM	replace Data / PX	Send1Token / PO	sendAllTokens / PX	sendAllTokens / PX	send1Token / PO	sendAllTokens / PX	updateData sendAck	updateData sendAck	updateData sendAck / M			
E	load	store	replaceData / PX	send1Token / PO	sendAllTokens / PX	sendAllTokens / PX	send1Token / PO	sendAllTokens / PX						
M	load	store	replace Data / PX	send 1Token / PO	send AllTokens / PX	send AllTokens / PX	send1Token / PO	sendAllTokens / PX						
IS							askToRetryBC	askToRetryBC	updateData sendAck / S	updateData sendAck / O	updateData sendAck / M		send SpecialGetS	
IM						send AllTokens / F	askToRetryBC	askToRetryBC	updateData sendAck / SM	updateData sendAck / SM	updateData sendAck sendComplete / M		send SpecialGetX	send SpecialGetX
SM				askTo RetryLater		send AllTokens / F	askToRetryLater	askToRetryLater	updateData sendAck	updateData sendAck	updateData sendAck sendComplete / M		send SpecialGetX	send SpecialGetX
PS					Inform TokenDest	inform TokenDest	askRetryBC	inform TokenDest	sendAck bounceL2	sendAck bounceL2 / PX	sendAck bounceL2 / PX	I		
PX				Inform OwnerDest	Inform TokensDest	inform TokensDest	inform OwnerDest	inform TokensDest	bounceL2	bounceL2	bounceL2	I		
PO				send1Token	iInform TokensDest sendAllTokens / PX	inform TokensDest sendAllTokens / PX	send1Token	inform TokensDest sendAllTokens / PX	updateData sendAck	updateData sendAck	updateData sendAck	I		
F				retry WithBoss	retry WithBoss	retry WithBoss	*True racing requests*		bounce ToBoss	bounce ToBoss	bounce ToBoss	F	send GetX	send GetX / IM

Both events and actions are a reduced list of the two sets. Different specific situations (according the received message and the state of the block) and all the corner cases that occur in a CMP have to be managed by the coherence controller and the protocol has to be prepared for **all** of them. However, with table 4-4, the reader might obtain a general idea of how a private coherence controller with LOCKE protocol works.

Two different groups have been differentiated in table 4-4 by cells with thicker lines: those directly related to the false racing requests and those about the true racing requests. Cells that are empty indicate special transitions. Green and blue represent generic control actions. Green cells indicate that the controller has to stall the incoming request, because it has another one pending and that has not finished. Blue indicates that the incoming message is ignored, which usually happens because it is not needed any more. Red cells represent transitions that cannot happen (if they occur the coherence controller is not properly implemented).

In the 'false racing requests' transitions group of table 4-4, it is possible to see that, after receiving a request for a data block (*GetS*, *GetX*, *FreezeGetX*, *SpecialGetS*, *SpecialGetX*) or when having to replace it (*Replacement*), the cache sends that block with the tokens requested with the actions *sendAllTokens*, *send1Token* or *replaceData*. However, the state of the line does not change to invalid as it should be when data and tokens are removed from a cache, but instead it changes to one of the LOCKE control states (PS, PX or PO) to indicate that there is a pending acknowledgement for that address. In this way, if another request arrives at the cache controller and the line is in any of these states, the cache controller can inform about the next destination of the tokens sent (actions *informOwnerDest* and *informTokensDest*). Hence, it is possible to ensure that any broadcast request will find either the tokens or information about where they are going to next, so avoiding any false racing request

The 'true racing requests' are managed by using the F state. If we look at table 4-4, when the cache controller is in a transient state with a pending write operation (IM or SM) and receives a *GetX* which has more priority than its own (*FreezeGetX*), it changes the line state to F, self-freezing until the other request is over. The way priority of each request is set will be seen in the following section. While in the F state, any data received is bounced to the higher priority requestor, which in LOCKE is called the *boss* of the true racing request (action: *bounceToBoss*. The event *Complete* is triggered when the complete message broadcast by the coherence controller that it is maintained as the boss is snooped. From this moment, the controller is unfrozen and can retry its write request operation.

Similarly, the shared LLC controller will also expect an acknowledgement message whenever a data block with tokens is sent and it will also send it whenever a replacement block is received. However, in order to reduce the total amount of traffic due to the replacements, LOCKE allows

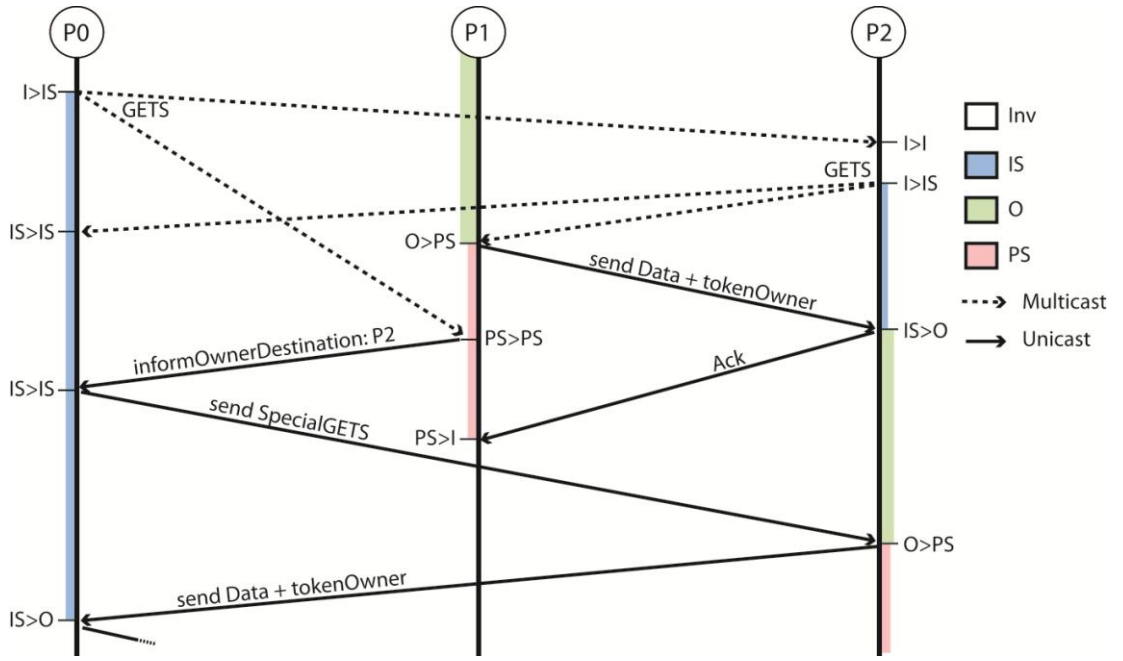

Figure 4-4. Token location with explicit acknowledgement: P_0 issues a GetS operation transitioning the block to IS and P_2 issues another GetS operation for the same block. The request from P_2 arrives first to P_1, which has the owner token. P_1 sends the data with the owner token to P_2, transitioning its own block to PS. This state will be maintained until the explicit reception acknowledgement from P_2 arrives at P_1. When the block is received at P_2, the block goes to the stable state O and the acknowledgement message is sent. In the meantime, the request from P_0 arrives at P_1 which informs it that P_2 has the owner token. P_0 reissues a unicast to P_2 requesting a copy of the data.

private caches to replace shared data blocks (with no owner token) just by sending the tokens and so avoiding sending the data block. This means that the LLC needs to be prepared for receiving these *data-less replacements*. For this reason, LLC needs an additional state A, which means that the data block is allocated, with invalid data but with one or more tokens. Except for this small design detail, the rest of the LLC controller will work the same way as seen for the private controller in table 4-4. Full LOCKE specification tables for all the coherence controllers can be found in [117].

Next, we will show with detailed diagrams how false racing requests are avoided and true racing requests are dealt with.

4.4 False Racing Requests: Token location

As was mentioned before, in order to determine token location and thus avoid false racing requests, any block movement is monitored at the originating location, keeping a label of the destination of the block. The label for the pending transaction is kept until a message reception acknowledgement is received from that destination. Thus, when a coherence controller generates a request, all the tokens needed or the flag of some pending acknowledgement will be found. Note that, in contrast to directory-based protocols, LOCKE's acknowledgement messaging is outside the critical path.

As seen in the previous transitions table, if the request corresponds to a write operation (*GetX*), any token will be forwarded to the requestor. On the contrary, if the request corresponds to a read operation (*GetS*), only the controller with the owner token will reply. If the request arrives when the tokens required are in-flight, the requestor is notified with the final destination of the tokens. Thus, the requestor may reissue a unicast request to the one holding the necessary tokens. The intermediate node always notifies the requestor if the transaction is a *GetX*, but only notifies if the owner token is in-flight when the request is a *GetS*. Note that this is the situation depicted in the example in figure 4-4. Processors P_0 and P_2 simultaneously try to perform a *GetS* operation for the same block, and P_1 holds only the owner token for that block. P_2's request reaches P_1 first, so P_1 sends its data with the owner token to P_2. When P_0's request reaches P_1, it finds the pending acknowledgment flag so P_1 notifies P_0 to retry its request to P_2. If this same situation happens when using a Token Coherence Protocol, a false racing request would occur. The side effect of this mechanism is the generation of extra unicast traffic for acknowledgement packets and reissuing the *GetS*. As we said before, in contrast to directory-based coherence protocols, acknowledgments operate outside the critical path of any memory transaction. In this example, the hit latency of processor P_2 will not be increased because of the mechanism.

4.4.1 I-trees

Unfortunately, the previous scheme is starvation prone. To exemplify this, figure 4-5 shows the same initial situation as in the previous figure 4-4, but this time, P_0's request is delayed long

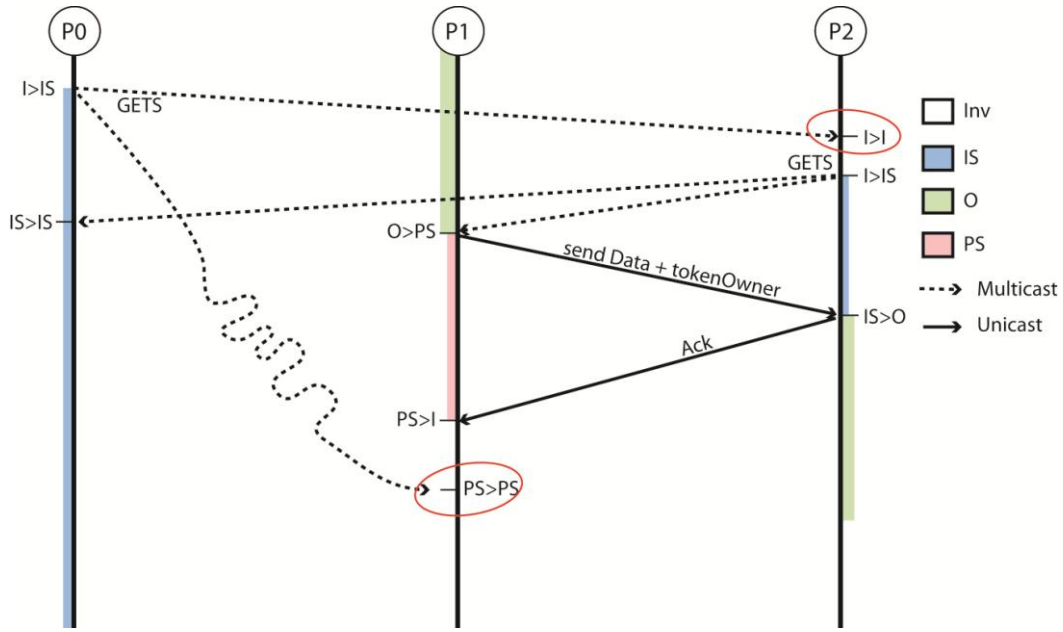

Figure 4-5. Starvation with request overtaking: With the same initial state depicted previously, the P_0 multicast request message arrives at P_2 before it issues its own GetS and most importantly, it arrives at P_1 after the acknowledgement reception from P_2. Both processors P_1 and P_2 ignore P_0's request.

enough so that it arrives at P_1 when the acknowledgement message from P_2 has already been received. In this situation, P_1 does not notify P_0 that P_2 has the block and the owner token. Moreover, P_2 is unaware of P_0 being interested in that block because P_0's request arrived at P_2 before this processor issued the *GetS*. If both of these things happen, P_0's transaction starves.

In order to prevent this anomalous situation, we need an approach to order both requests on the interconnection network. The most scalable way to perform this ordering is to use a *fixed multicast tree* for each set of addresses. If we force all the requests to a specific address to follow that tree, then no request or acknowledgment race is possible because the messages involved cannot be overtaken. To balance network resource utilization we could define different multicast trees per address. The routers should include the mechanism to use the right tree according to the address accessed. Using the least significant bits in the address we could select which one to follow. Figure 4-6 shows a possible distribution in an 8-processor CMP with a non-uniform cache architecture (NUCA) using a 4×4 mesh interconnection network and four multicast trees. We will denote the multicast trees as *I-trees*. To minimize base latency effects, each *I-tree* trunk can pass through the last level (LLC slice where the address could be located). Note that one of the destinations for the broadcast request will be an L2 slice. For example, addresses mapped to slice 0, 4, 8 and 12 will use the I-tree for addresses whose last 2 bits are 00.

In any case, a marginal latency increase will be observed if source and destination are in the same column and the LLC slice is not. It should be noted that this tree has no relation to home indirection in directory (there is no root or serialization point). Any multicast-capable network will require a multicast tree [64].

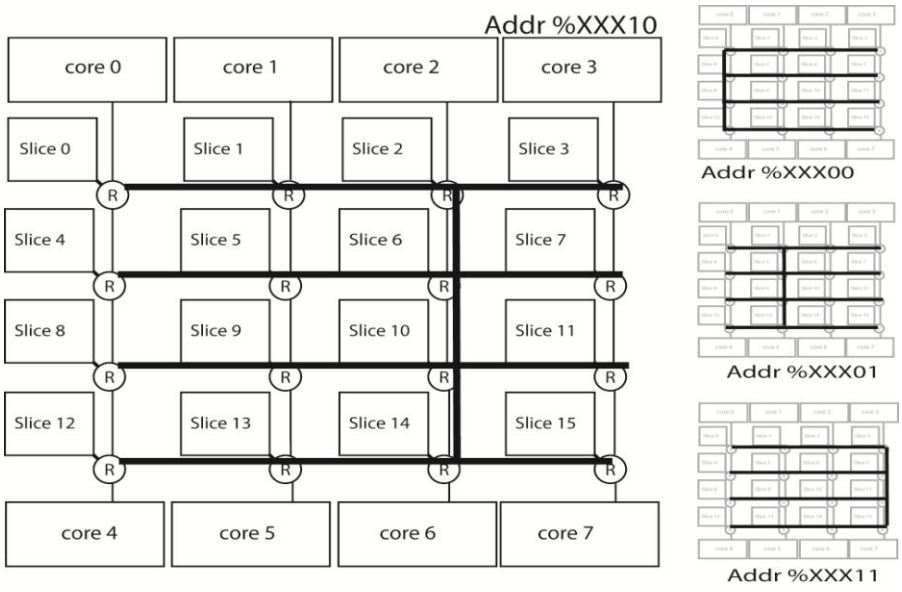

Figure 4-6. Ordering I-tree in a NUCA architecture.

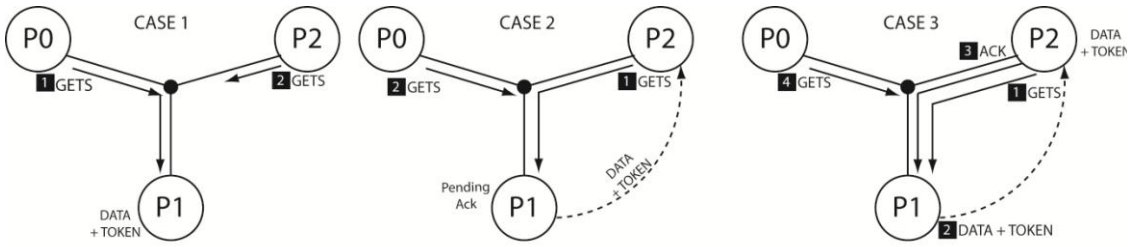

Figure 4-7. Three possible situations when using I-trees considering one common point. Case 1) Data is found in P_1: P_0's request arrives first at the common point, so it reaches data in P_1 first. Case 2) The pending acknowledgement is found in P_1: P_0's request arrives second to the common point, so it reaches P_1 after P_2's request, finding the pending acknowledgement. Case 3) Data is found in P_2: P_0's request reaches the common point after the acknowledgement from P_2, meaning that data is in P_2.

For example, in figure 4-6, if core 0 requests data that is located in core 1 L1 cache, it will take only one hop in the network to reach it. In the worst case, if data is located in core 4 L1 cache using the I-tree of the figure it will take 7 network hops to reach it, while in an optimal multicast tree it will take 3 hops. Although the average impact on on-chip latency overhead will depend on data distribution and network contention, the average distance increment for multicast messages is less than 10%. Moreover, the rest of the traffic (responses, acks, etc.) will follow minimal paths.

What we are able to do by using these trees is to always have *a common point for every communication between three points*. If we take the previous example in figure 4-5 and consider that the three processors are connected with an I-tree, they would have a common point among them in some place of the network, as is shown in figure 4-7.

Under this circumstance, when P_0 broadcasts its request and reaches the common point, three different cases may occur:

- If P_2's request is still going to that common point without reaching it, it will find the token needed in P_1 (case 1).
- If P_2's request has already passed that common point, it will reach the token first, and so P_0's request will find the pending acknowledgement mark (case 2).
- If P_2's acknowledgement has passed that common point, P_0 will find the data needed at P_2 (case 3).

These three possible situations are the only ones that can happen and that it is why it is possible to ensure that when using LOCKE any request will find either the tokens in their location, or the pending acknowledgement mark in its origin.

4.5 True Racing Requests: Arbitration

4.5.1 Self-inhibition

If the location of all tokens needed to complete a transaction is known then only true racing requests have to be resolved. When two or more processors are trying to perform simultaneous but incompatible operations, LOCKE deals with the situation using scalable self-regulated arbitration. The option adopted is to assign a priority order to each processor and operation and to allow the resolution of the race without breaking the coherency invariants. The different coherency controllers apply this policy in a fully distributed way, so guaranteeing system scalability.

Two or more simultaneous operations on the same block are incompatible if the total number of required tokens is greater than the number of processors P. If one coherence controller detects the possibility of such a situation arising, it must choose whether to keep going with the operation or to give up. For example, if it wants to perform a write operation in a cache block and sees an incoming write request from another processor trying to write in the same cache block, it has to check each request's priority. Initially and for the sake of simplicity, we will assume that the priority is determined by the processor index. If the current controller has an index smaller than the incoming request, the controller goes ahead with its operation or, if not, it self-freezes the operation.

If the controller decides to temporarily inhibit the outgoing transaction, due to its lower priority with respect to the remote incoming request, it changes the block state to "frozen" and annotates the winner controller for that block. When a block is frozen, any incoming token will be forwarded to the annotated winner controller. The block will remain in a frozen state until the winner notifies the completion of the operation, via a *complete multicast message*. If this happens, the inhibited operation is reissued from the beginning. Figure 4-8 presents an example of this situation. We will assume that P_2 has all the tokens and P_0 has the highest priority. In P_1's controller the block changes to frozen as soon as the request from P_0 is seen. When tokens and data arrive at P_1 they are forwarded towards P_0. On each interchange of tokens the controller has to carefully deal with the acknowledgement signaling. When P_0 completes its operation, it awakens P_1, which reissues its pending GetX.

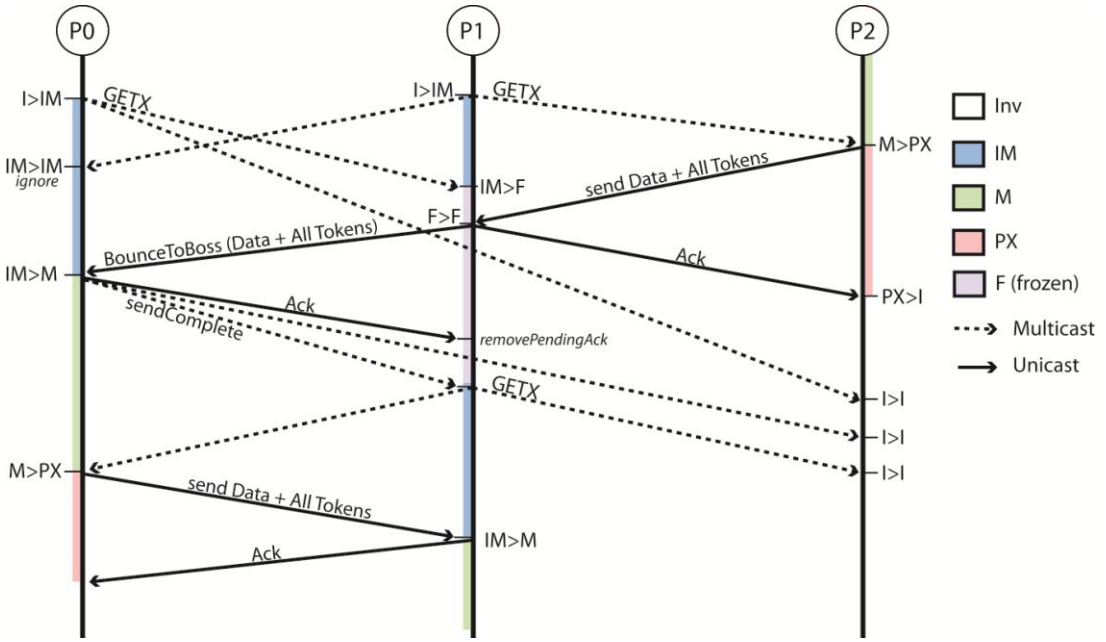

Figure 4-8. Example of write serialization: P_0 and P_1 simultaneously issue a GetX on a block which is in M state at P_2 (i.e., all the tokens are located there). We will assume for now that P0 has higher priority than P_1. P_1's request arrives at P_2 first, so P_2 sends data and all tokens, changing its state block to the transitory state PX until the acknowledgement from P_1 is received. Before receiving the data and the tokens, P_1 snoops a request from processor P_0 which has greater priority than its own one, so it self-freezes its operation and annotates P_0 as the winner at the MSHR (its boss). When data and tokens from P_2 arrive at P_1, they are immediately forwarded to the winner P_0, annotating the in-flight tokens. When P_0 receives the data and tokens it sends an acknowledgement to P_1 (as it corresponds to any token movement) and finalizes its operation. When P_0's GetX operation ends, it broadcasts a complete message. P_1's MSHR hit unfreezes the operation and reissues it.

When a block is frozen, any other write request from another controller, no matter what its priority is, will be ignored. Thus, depending on the timing of the reception of requests, an implicit tree of pending operations is formed. This tree has a tendency to follow the address I-tree shape. Usually, independently of the number of controllers that are trying to perform the operation concurrently, the ordering tree shape is deep. Therefore, the request reissue after reordering is lazy; only one pending memory transaction is reissued after the completion of a write in most cases.

4.5.2 Fair priority ordering with out-of-order processors

Statically assigned priorities could provoke pathological situations, because contended blocks are obtained most often by the same processor. Nevertheless, assuming multiple outstanding requests per core, there is an easy and scalable solution to deal with this if we can guarantee that:

- Two different processors cannot issue an operation to the same block with the same priority.

- The probability of having a different priority ordering at two contended blocks from two different processors has to be non zero.

The first condition guarantees that two different processors will never grab simultaneously a subset of tokens from the same block, i.e. avoiding starvation. The second condition guarantees that, on average, no processor memory operations are favored over others. The most straightforward way to achieve this is to construct the priority of each request as the combination of the processor ID (LSB bits) used to achieve condition one, and a small random number (MSB bits) that would be added to each write request to achieve condition two. The priority is maintained until the request is completed.

Experimentally, it is observed that this approach provides similar performance to an age-based priority (which requires a complex coordinated timestamp-based mechanism) at a fraction of the cost. On average, this approach equalizes processor work balance. The performance reduction observed for a four-bit random number compared to an idealized fully age-based approach is less than 1%. The bottom line is that only 4 bits of overhead per package is required (the requesting node is always included in it).

4.6 Evaluation

In order to validate the advantages of our proposal, we have used two coherence protocols for the given system architecture with the configuration parameters shown in table 4-5. The main parameters of the target system mimic state-of-the-art high-end CMPs such as [5][4][118]. An optimized directory protocol similar to the one used to compare to the token coherence protocol in [119], but adapted to NUCA is used as a baseline coherence protocol. Directory information is distributed across all slices and full mapping. Optimistically, null storage overhead is assumed for this protocol.

Broadcast-based token coherence protocol variation [115] is considered, as a representative counterpart of snoop-based protocols. A fixed timeout to reissue the request is set. Only one reissue is tolerated before triggering a persistent request. The timeout has been selected measuring all the benchmarks with different timeouts and choosing the one with best average performance. For the system configuration used, the selected time-out is 330 processor cycles. A dynamically estimated time-out does not provide performance benefits.

To evaluate the coherence protocol, a 4-way superscalar out-of-order processor architecture is chosen. Having multiple outstanding memory requests makes the coherence protocol more relevant and it will help us check LOCKE under more stressful circumstances.

For the cache hierarchy, we assume NUCA [111] for the last level cache. Although LOCKE is also applicable for a tiled system, NUCA architecture is a better approach because it decouples the number of LLC cache slices from the number of cores, providing much more flexibility to scale available on-chip bandwidth.

Table 4-5. Basic system configuration, 32 nm. technology assumed for energy estimations.

		Configuration 1	*Configuration 2*
Processor Config.	Number of cores	8 @3GHz	16 @3GHz
	Functional Units	4xI-ALU / 4xFP-ALU / 4xD-MEM	
	ROB size	128	
	Fetch/Issue/Retire Width	3/4/3 way	
	Fetch-to-Dispatch	7 cycles	
	Branch predictor	YAGS with 8K entries	
L1 Cache	Block Size	64 Bytes	
	Size	128KB Instruction/Data	
	Associativity	4-way	
	Access Time	2 cycles	
	Max. number of outstanding memory operations	16	
L2 Cache	Block Size	64 Bytes	
	Size (number of banks × size per bank)	8MB (16×512KB)	16MB (32×512KB)
	Associativity	16-way	
	NUCA Mapping	Static, interleaved across slices	
	Bank Access Time	5 cycles	
Memory	Capacity	4GB	
	Access Time	240 cycles	
	Num. Memory Controllers	2 centered	4 centered
	Bandwidth	32 GBs	64 GBs
Network	Topology	4×4 Mesh	6×6 Mesh
	Link Latency and Width	1 cycle – 16 Bytes	
	Router Latency	1 cycle	
	Flow Control	Wormhole	
	Buffering per Router	5.4 KB	
	Routing	DOR	

As far as the interconnection network is concerned, we will add the minimum variation over commonly used router microarchitecture and network topology. As for the router micro-architecture used, it will be similar to the proposal described in [32], using on-network multicast support when required. We use dynamic buffering allocation per virtual channel and 1-cycle low load pipeline pass-through. In each protocol we use the required number of virtual channels

to avoid message-dependent deadlock [120] and network deadlock. Dynamic buffering enables the use of a fixed capacity per router of 5.4KB. DOR routing is used when no *I-tree* has to be followed by messages.

In order to observe the scalability of the proposal, we chose two different system sizes composed of 8 and 16 processors. The eight-processor system layout is similar to the one shown in figure 4-6. For the second configuration, although processor, L1 and router specifications remain unchanged, LLC capacity and bandwidth are scaled up in accordance with the larger number of processors. For this, L2 has 32 slices of 512KB mapped over a 6×6 mesh for a total of 16MB.

In all configurations, instead of using in-order cores, we opted to mimic [4][118] with aggressive out-of-order processors. Although a large number of small cores could make sense for cloud-computing workloads, we focus LOCKE on the CMPs forecast in the previous chapter for the near future: aggressive and powerful processors inside the chip. Additionally, medium size systems with a large number of outstanding memory transactions per processor are much more demanding for the coherence protocol than many simple cores.

Moreover, a detailed description of the methodology used, including simulation stack and workloads description can be found in Appendix A.

4.6.1 Performance and efficiency

Figure 4-9 provides performance with the basic 8-processor CMP (*config1* in table 4-5). On average, DIRECTORY is outperformed by both snooping protocols LOCKE and TOKEN. As expected, some workloads are insensitive, which attenuates average performance impact of coherence protocol. In contrast, in applications with highly contended blocks, such as numerical benchmarks (see appendix), coherence impact on performance is quite relevant. In those cases, LOCKE outperforms other protocols by up to 30%. In applications with high sharing degree but limited contention, such as server workloads, LOCKE outperforms the other counterparts by a smaller but still visible margin. Although, on average, TOKEN performs better than DIRECTORY, some noticeable results such as IS or FT, even in a modest size system like this one, show its performance is poor due to the reasons explained in Section 4.1.2. In contrast, LOCKE exhibits a consistent performance across all the workloads.

End-point traffic comparison of different protocols may not reflect a direct impact in performance or energy profile. First, using routers capable of handling multicast traffic, as in our case, causes a multicast packet with *n* destinations not to use the same effective bandwidth as *n* unicast packets for the same destination [32]. Second, network energy is only a part of the on-chip memory hierarchy which is dominated by cache. Third, Energy Delay Square Product (ED2P) is the most suitable metric to estimate energy-performance tradeoff in high-performance systems such as ours [121]. Therefore, we provide this metric, grouped for each suite of benchmarks and protocols in figure 4-10. As we can see, the cubic influence of performance in ED2P has a major effect, meaning that the ED2P of the network, in spite of producing more traffic, is even smaller for broadcast-based protocols. Additionally, for 32nm technology and a large cache footprint (8MB in this configuration), leakage power, which is constant across coherence protocols, causes the ED2P leakage proportion to grow significantly when the performance is worse. Therefore, snoop-based broadcast coherence protocols have lower average ED2P than directory-based for this type of architectures. Due to the more consistent LOCKE performance, on average it requires 19% less ED2P than DIRECTORY. In contrast, due to performance instabilities, which negatively affect some workloads, TOKEN is only capable of saving 7%.

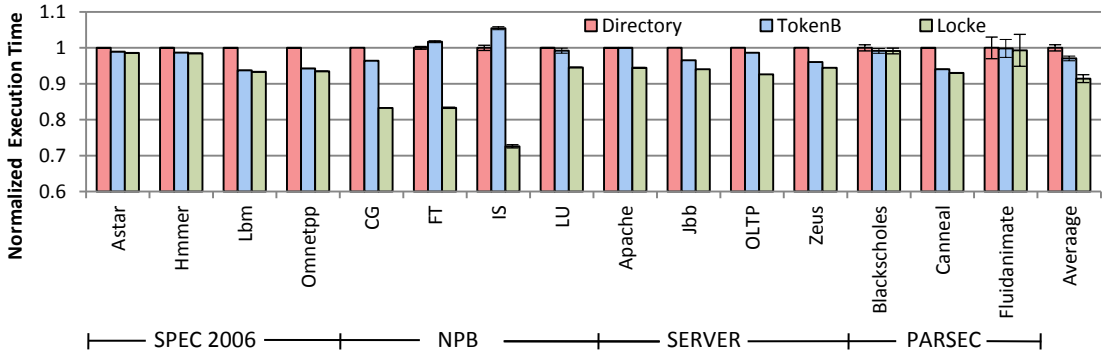

Figure 4-9. Directory normalized execution time in an 8-processor CMP.

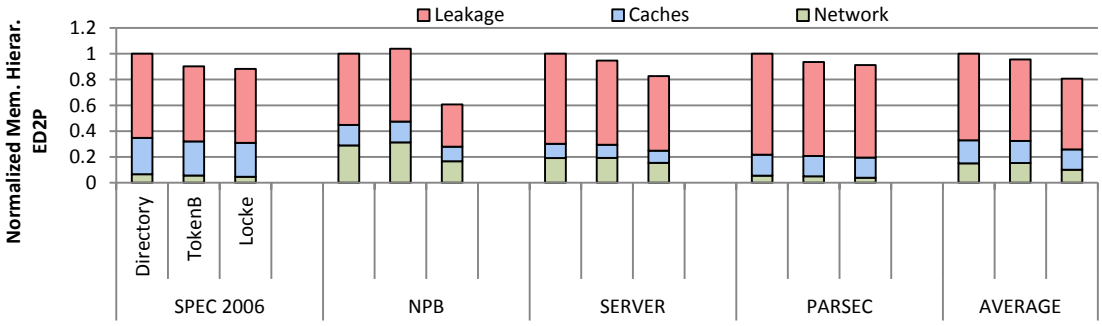

Figure 4-10. Directory normalized memory hierarchy ED2P in an 8-processor CMP.

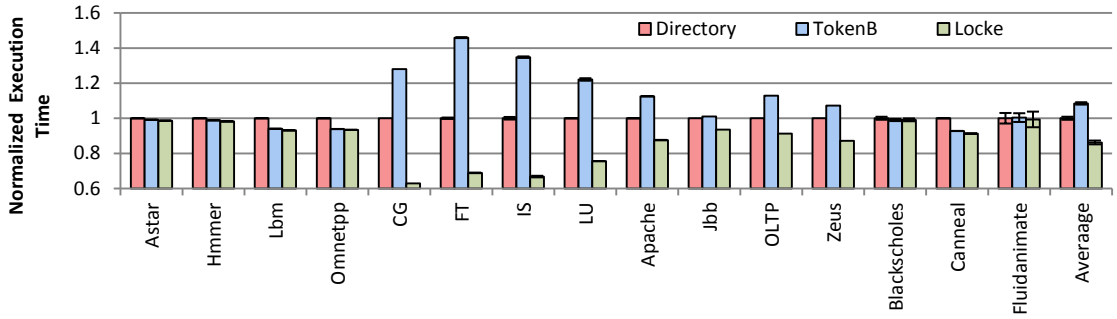

Figure 4-11. Directory normalized execution time for a 16-core CMP.

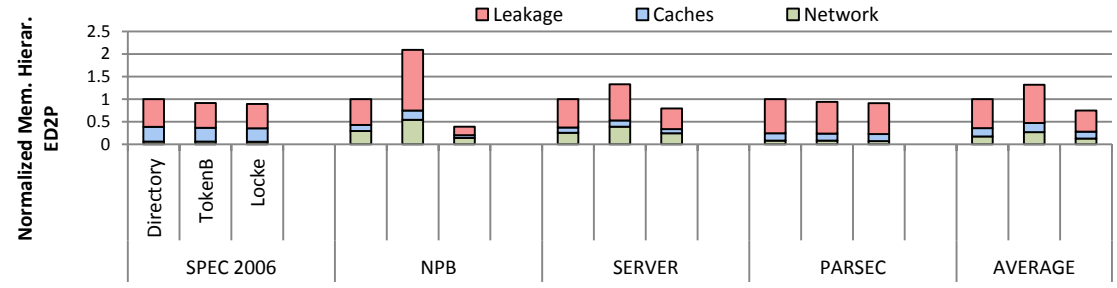

Figure 4-12. Directory normalized memory hierarchy ED2P for a 16-core CMP.

4.6.2 Scalability

Although LOCKE is not a coherence protocol envisaged for use in large architectures (the coherence protocol presented in the next chapter is designed for these type of systems) and its objective is to improve the protocol responsiveness in small-to-medium ones, it is important to analyze whether its performance advantage is maintained when increasing the number of processors. To explore its scalability, we increase the system size to 16 cores (*config2* in table 4-5).

The performance observed in figure 4-11 indicates that LOCKE is able to increase its advantage in comparison to DIRECTORY. Scaling up the network size to accommodate NUCA slices would increase the cost of DIRECTORY indirections. Nevertheless, the increased contention due to larger numbers of multicast destinations seems not to increase the latency in the network significantly for LOCKE. Therefore, the performance advantage of LOCKE over the directory is now greater than in the 8-core CMP (16%). In contrast, TOKEN performs poorly, being noticeably slower than DIRECTORY.

4.6.3 Responsiveness

LOCKE's main objective is to offer an option with good performance and responsiveness, especially in situations with high contention, where other snoopy protocols do not perform very

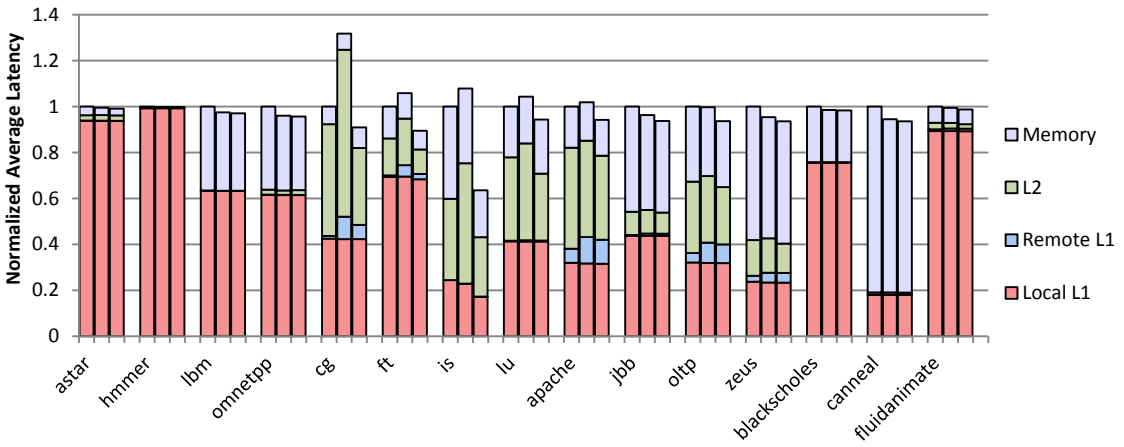

Figure 4-13. Directory normalized average latency for an 8-core system.

well. To demonstrate LOCKE's effectiveness, figure 4-13 shows the average latency perceived by the processor with each of the protocols for an 8-processor system. Although memory level parallelism and synchronization makes it hard to directly translate any performance difference into average access times, those metrics could help to understand the performance differences. As can be appreciated, DIRECTORY-based protocol has a larger memory contribution in some applications. This is a direct consequence of inclusiveness. Whereas snoop-based protocols do not need inclusiveness to track on-chip block sharers, DIRECTORY requires an entry in LLC for any L1 cache block. Consequently, the effective cache capacity is smaller than in the snoopy protocols and so LLC miss rate is raised. This problem is acknowledged as a serious drawback of directory coherence protocols [122][123] (the next chapter will present an efficient directory solution that solves this inclusiveness problem). TOKEN coherence introduces pressure on the network in some applications and the starvation avoidance mechanism increases the on-chip hit latency significantly, making the average access time up to 40% slower in applications such as IS. In contrast, LOCKE seems to consistently outperform other protocols in most applications.

Although, on-chip hit latency provides a good idea about protocol efficiency, it might be interesting to isolate how the protocol reacts when multiple coherence events arise simultaneously for the same block. In such situations, the effectiveness of the protocol is the key to prompt resolution of the situation. Figure 4-14 shows how effective each protocol is when resolving true racing requests in eight-processor systems. As we can see, in most cases LOCKE is the fastest one, being on average 10% faster than DIRECTORY and 60% faster than TOKEN. Token's persistent mechanism to resolve those situations makes it the slowest one, being on average 40% slower than DIRECTORY. With non-conflicting coherence events broadcast-based

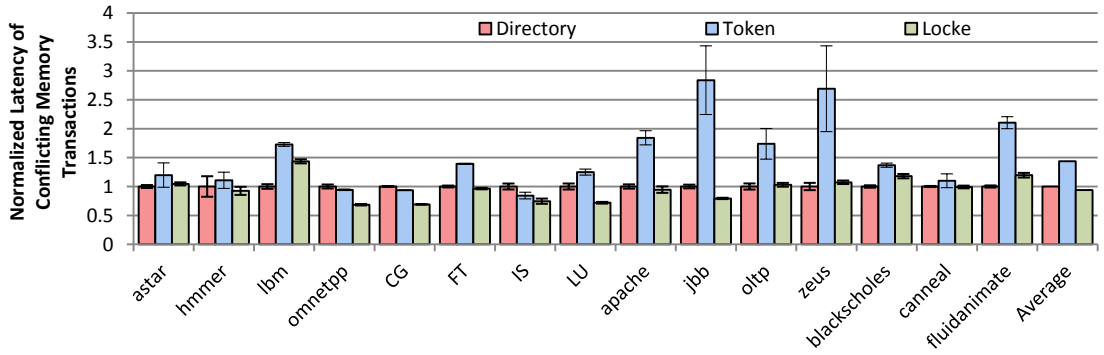

Figure 4-14. Normalized time to resolve conflicting memory accesses for an 8-processor CMP.

coherence protocols are faster that directory due to inclusiveness, which increases on-chip miss rate, as can be appreciated in the memory contribution in figure 4-13.

4.6.4 Network Energy Impact of Multicast traffic

As stated before, it is commonly assumed that multicast traffic has a large impact on network power consumption. This assumption is based on the large increment in control traffic observed at the end-point, i.e. consumers. Nevertheless, when a network has multicast support, i.e. on-network packet replication, this is completely wrong because multicast packets use network resources only once before replication [32][82]. Therefore, unlike unicast-only networks, in multicast-capable networks energy consumption is not proportional to end-point traffic, but to average link utilization. For example, figure 4-15 shows the directory normalized network link utilization for LOCKE and TOKEN for 8-processor and 16-processor CMPs. All the links in the interconnection networks have been considered, including connections from routers to L1 caches, L2 slices and memory controllers.

As we can appreciate, network activity in snoop-based protocols is not much higher than directory protocols, at least for the medium size system considered in this work. Multicast

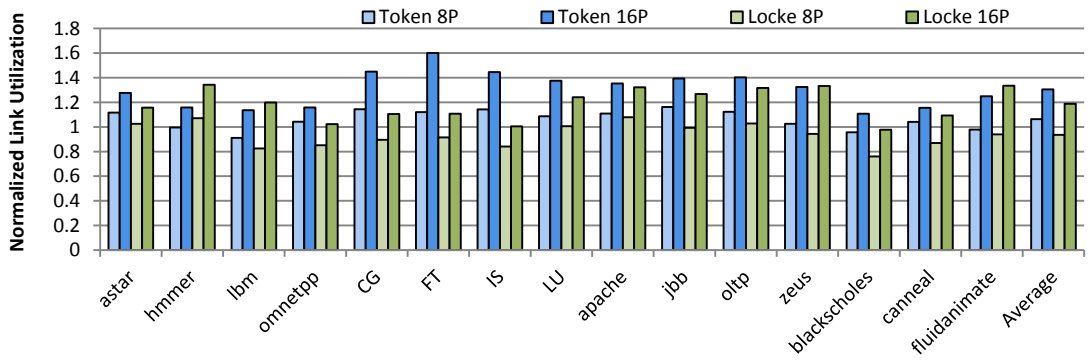

Figure 4-15. Directory normalized average network link utilization.

capable routers have an identical data-path to conventional ones, so normalized link utilization differences will be translated into energy consumption (and negligible implementation cost). In all cases, LOCKE has lower link activity, because the multicast tree used is much deeper than the one used in TOKEN, which tries to reach all the destinations as soon as possible replicating the message earlier. As indicated in section 4.4.1 LOCKE ordering *I-trees* delay packet replication, which increases request base latency, but reduces network load. With particularly demanding applications, such as most NAS Parallel Benchmarks, or bigger system sizes, TOKEN starvation avoidance increases the amount of activity. Even in the largest system, network system activity is only 15% greater in LOCKE than in DIRECTORY. Performance benefits offset this, making LOCKE the most efficient coherence protocol of the three. With small size, the system DIRECTORY generates more network activity than snoop-based protocols due to protocol indirections and the larger number of on-cache misses.

4.7 Conclusions

Throughout this chapter a new coherence protocol has been presented and evaluated. LOCKE successfully exploits large on-chip bandwidth availability to improve cache-coherent chip multiprocessor performance and energy efficiency. Provided that the interconnection network is designed to support multicast traffic and the protocol maximizes the potential advantages that direct coherence brings, we demonstrate that a multicast-based coherence protocol could reduce energy requirements in the CMP memory hierarchy. The key idea presented is to establish a suitable level of on-chip network throughput to accelerate synchronization by two means: avoiding the protocol serialization, inherent to directory-based coherence protocol, and reducing average access time more than in other snoop-based coherence protocols, when shared data is truly contended. LOCKE is developed on top of a Token coherence performance substrate, with a new set of simple proactive policies that speeds up data synchronization and eliminates the passive token starvation avoidance mechanism. Using a full-system simulator that faithfully models on-chip interconnection, aggressive core architecture and precise memory hierarchy details, while running a broad spectrum of workloads, our proposal can improve both directory-based and token-based coherence protocols both in terms of energy and performance, at least in systems with up to 16 aggressive out-of-order processors in the chip.

Chapter 5. Scalable coherence for large CMPs: MOSAIC

Up to now, we have considered small to medium architectures with tens of processors. In the previous chapter, LOCKE's scalability was analyzed and although its results for a small number of cores showed very good performance, it is necessary to be aware that when considering large architectures with hundreds or thousands of cores, the interconnection network will not be able to provide support to all the broadcasts made for every miss happening in the private caches and so relying on broadcast-based coherence protocol will become unfeasible. For this reason, it is necessary to find more scalable solutions that reduce the number of broadcasts made.

Historically, directory-based coherence protocols have been used to address the scalability problem in multiprocessor systems. However, nowadays CMPs present specific characteristics that change the situation. On the one hand, unlike what happened with the bus used for the interconnection network, when using meshes and torus inside the chip, bottlenecks are avoided and bandwidth availability is not the problem anymore. In contrast to off-chip networks, on-chip link bandwidth is profuse and in latency-sensitive scenarios, link availability is usually employed to build wide channels, reducing the serialization penalty of communications. On the other hand, the private section of the cache hierarchy in current systems is quite large, in order to achieve progressive hit-times throughout the different levels of the memory hierarchy [124]. As the memory wall effects become more relevant, more on-chip cache capacity will be required and therefore large private caches will be needed. These large capacities require large storage necessities to keep all the coherence information about all the data copies in the system. As has been mentioned before, this coherence information has to maintain the inclusiveness and hold all the information about the copies allocated in the private levels. Depending on the directory design chosen, this inclusiveness will have a certain effect. In the *in-cache* directory, including all the coherence information in the LLC will mean, on the one hand, that the space that has to be reserved for storing the information in each of the blocks will be increased, although on some occasions it will not even be necessary. On the other hand, the effective capacity of the LLC will be reduced since there will be progressively more blocks that will have to be dedicated to maintaining this information and fewer blocks dedicated to victim cache for private replacements. When a sparse directory design is chosen, the total effective capacity of the LLC is recovered, but the directory size has to be correctly set in order to avoid the negative effects of inclusiveness, needing to send recall messages to invalidate private blocks that are being used because there is no available space in the directory. This "correctly-dimensioned"

attribute of the directory is not easy to choose because it might reach large values, which in some cases could even be unsustainable. For all these reasons, neither of the two solutions, broadcast and directory (neither of the two designs explained), seems the most suitable choice. However, we believe that the hybridization of the two approaches in a single proposal is a reasonable way to overcome their inherent limitations. By designing a new coherence protocol that it is able to exploit the on-chip bandwidth availability, it is possible to eliminate the necessity of inclusiveness of the data present in the private caches.

The coherence protocol introduced next, MOSAIC, is able to take advantage of the bandwidth availability inside the chip in order to avoid the necessity of inclusiveness and still keep the system scalable. Sending broadcasts to reconstruct the directory information whenever it is needed avoids having to maintain inclusive information in the directory, although it requires extra bandwidth. However, token counting enables the LLC to be used as a filter to eliminate most of these broadcast messages, which enables a scalable system to be achieved.

The chapter is divided into 4 different sections. Initially, the conceptual approach of the proposal is shown in section 5.1 in order to give the reader a general idea of how the protocol works. This generic idea will be extended with the design details in section 5.2 and specific and detailed examples will be presented in section 5.3. The MOSAIC proposal will be fully analyzed in section 5.4 and a summary of the possible optimization paths for the future will be explained in section 5.5. Section 5.6 will end the chapter with the conclusions.

5.1 Conceptual Approach

The MOSAIC protocol is focused on reducing one of the main problems that the conventional directory approach has when dealing with a large number of processors and with large number of blocks kept in the private levels: the space needed to hold all their coherence information. The cost of the directory is proportional to the size and plurality of the private levels. In order to break this directory constraint, MOSAIC does not evict blocks from the private levels when there is not enough space in the directory and some coherence information has to be removed to allocate new coherence lines. This means that the blocks can be kept in the private caches, although the directory is not tracking them anymore. Thus, coherence information inclusiveness is completely removed from the directory, allowing some restrictions to be eliminated when deciding the size of the directory.

Without this inclusiveness eforcement property, when a request is received and a miss occurs in the directory, it is not possible to know whether the requested data block is allocated in the off-chip memory, in the LLC and/or in any of the private levels. For this reason, the coherence

protocol needs to have a special mechanism to find out and to locate all the possible copies of the requested data.

In order to be able to collect all the coherence information associated with a requested block, after any subsequent miss in the directory, an on-chip reconstruction of the directory entry is initiated. This reconstruction process starts by checking in the LLC whether the requested block with all the tokens is present. If it is not, a broadcast message is sent to all the private caches asking for information about the requested block. This process will end when all the coherence information associated with that block (i.e. the sharers of the block and their state) has been collected. By using token counting [17], the process is kept simple and negative acknowledgements [122] are avoided. This is possible because only the private caches that have the data block with some tokens have to reply to the reconstruction broadcast message. These replies will include the number of tokens that they have, so by adding all of them the directory will know when it has finished the reconstruction process. It is important to bear in mind that the directory will not store the number of tokens each private cache has and it will only store which of them have a copy (i.e. the sharers) and which one has the owner token.

To explain the whole process in a more graphical way, figure 5-1 presents a schematic sketch of how MOSAIC behaves. The example starts with a *read request* from processor P_0 that, after missing in its private cache, sends a read request to the directory slice ❶. If the directory does not have any information about the requested data block, it checks whether it is present in the

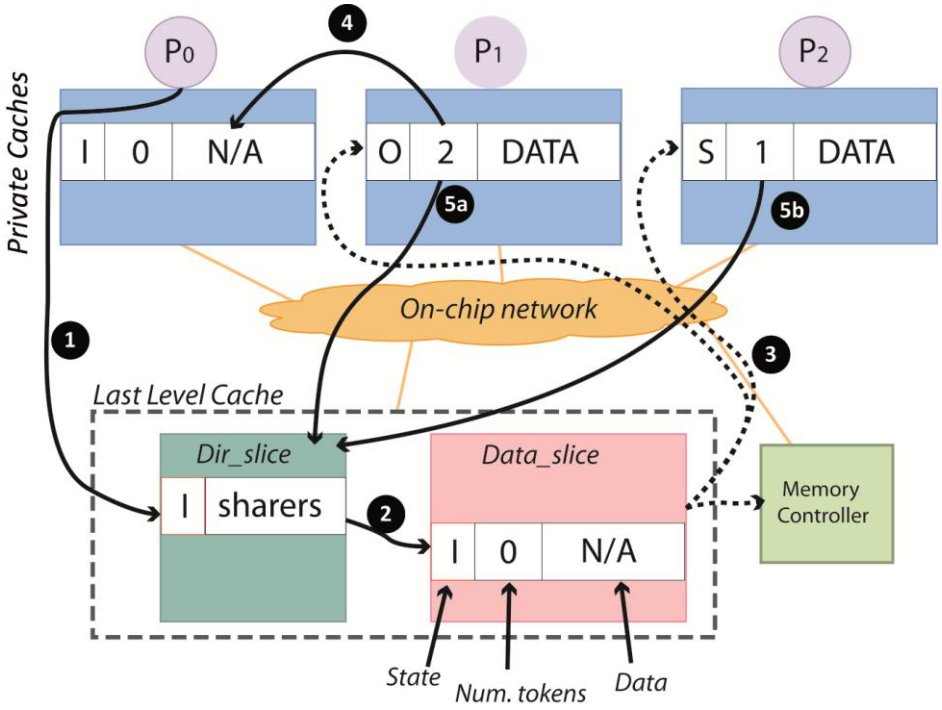

Figure 5-1. Sketch of MOSAIC's concept after a request from P_0 misses in the LLC and in the directory.

LLC ❷, and if it is not, it starts a broadcast reconstruction message looking for the data block needed ❸. This reconstruction message has two objectives: to build the directory sharers information and to solve the request that initiated the whole process. For this last goal, the reconstruction message includes information about who started the reconstruction and for which type of request it did so. Thus, the corresponding private caches will be able to know when and how they have to reply to the requestor. This means that, for example, in figure 5-1, since the starting request is a read request, only the private cache holding the owner token will be in charge of solving it. For this reason, P_1 sends a copy of the data block with one of its tokens to P_0 ❹. To achieve the first goal of the reconstruction process, the directory needs to collect all the information about the requested data block. So it needs to know who is holding any tokens associated with that address and also how many of them they have, in order to know when the directory has finished collecting all the information. In figure 5-1, P_1 and P_2 send the information about their tokens to the directory ❺. For a *write operation*, the reconstruction process is similar with the difference that all of the sharers will forward their tokens to the requestor (invalidating their copies) without sending any message to the directory. The requesting processor, after collecting all the tokens, will notify the directory with a completion message. In any case, once the entry is fully constructed, if the directory needs to evict it, because of lack of space in the directory after a subsequent miss, MOSAIC does not need to invalidate any of the private copies. It may replace the entry silently because it will be reconstructed if necessary.

5.2 Design details

MOSAIC coherence protocol may be used either in a sparse directory or in an in-cache directory. The only difference between using one or the other of them is in the coherence controller that is in charge of constructing the line, which is the element holding all the coherence information and acting as the directory. This coherence controller can be a standalone directory in the sparse design or the LLC controller in the in-cache design.

Each of the entries in the directory, or in the LLC, will hold the coherence information about the address it makes reference to. The main states that might be considered are the ones naming the coherence protocol: Modified (M), Owner (O), Shared (S), Allocated (A), Invalid (I) and Constructing (C). The meaning of the first three and the invalid state are well known (chapter 2), but the new states A and C provide the key implementation details of the MOSAIC protocol. The C state indicates when an entry in the directory is being constructed and the A state defines when a line is fully constructed with all the coherence information attached. However, each of the designs has its own necessities and more importantly, its own possibilities for optimizations.

For this reason, these main states vary a little from one to another. Next, specific design details of each of them will be seen using the table-based transitions method.

5.2.1 Sparse directory specification

In a sparse design, the directory does not have data copies attached to each line. For this reason, having the M, O or S state in those entries does not apply, because the only necessary information is whether the entry is already constructed (A), being constructed (C) or invalid (I).

When the directory controller is constructing a line, the block enters a transitory state (C_S or C_X). To which of them will depend on whether the reconstruction process was started by a read request (C_S) or a write request (C_X). This distinction is necessary because the directory controller needs to recall why the reconstruction process started in order to take specific actions in case a race occurs and to avoid possible deadlocks, as we will see later. This requirement is also mandatory for the Allocated state (A) which is divided into A_S or A_X after a *GetS* or *GetX* request respectively for the same reasons. Table 5-1 summarizes a brief description of each state.

Besides the state of the block, the coherence information that each of the entries in the directory should include is: the *sharers* of that block, the core holding the *owner* token (as it will be in charge of forwarding data if necessary) and a *token-count field* of that block (we will see next why this is necessary). Any existing method to maintain the sharer information such as the ones

Table 5-1. MOSAIC protocol main states in a sparse directory.

States	Description
I	Invalid. Block is not present in the sparse directory.
C_S	Constructing the block after receiving a read request (GetS) from a core.
C_X	Constructing the block after receiving a write request (GetX) from a core.
A	Allocated. Block is fully constructed with all the coherence information about that block.
A_S	Allocated and a read request (GetS) has been received from a core. Waiting for an unblock message.
A_X	Allocated and a write request (GetX) has been received from a core. Waiting for an unblock message.
A_I	Invalidating a block.

shown in chapter 3 or others may be chosen [90][79]. However, a full bit vector will be assumed throughout this document to simplify the protocol complexity.

Table 5-2 shows a simplified version of the transition table of the sparse directory controller working with MOSAIC. When receiving a request (*GetS* or *GetX*), if the block is not present in the directory (state I), this controller initiates a reconstruction process like the one explained in the previous section. Notice that this reconstruction process is different depending on whether the request is a *GetS* or a *GetX* and so the state the entry has to change to is different (C_S or C_X respectively).

During the reconstruction, when the controller receives information about some tokens' location (event: *Token Info*), it adds that sharer to the sharers bit vector and updates the number of known located tokens. When the request triggering the reconstruction is a *GetS*, the cache with the owner token of the block will send a copy of the data with one of its token to the requestor. After that, it will inform the directory about how many tokens it has left. When the requestor finishes its request, it sends an unblock message (event: *Unblock*). The directory will always wait for the requestor's unblock message to finish the reconstruction. This will guarantee that no other request is dealt with until the entry is fully constructed and the request is completely resolved. If the request is a *GetX*, all the caches with a copy of the requesting block will have to forward their tokens to the requestor, which will send the unblock message when it has collected all of them and so its request is finished. In this case, the directory controller will add the requestor as the exclusive sharer of the data (state C_X, event *Unblock*).

If the coherence information needed is in the directory (state A), all the data locations are known so the directory only has to forward the request to the appropriate sharer. If it is a read request (*GetS*), it sends it to the cache holding the owner token; if it is a write request (*GetX*), it sends it to all the sharers of the block.

The directory needs to be informed about all the replacements occurring in the private levels in order to always have updated information about the sharers. Any private cache replacing a block sends a request with the tokens (event: *PUT Tokens*), or if it has the owner token with the data (event: *PUT Data*) to the directory. When there is a token replacement, the directory maintains this tokens in its entry increasing the number of tokens it owns (this is why the entry needs to have a token count field). When receiving a data replacement, if the entry is not constructed (state I) or there is no pending request (state A), data and all the tokens are written back to LLC[3] (action: *write data in LLC*).

[3] The directory writes back the tokens received along with any token it had from another previous replacement. Thus, tokens tend to be regrouped in LLC.

Table 5-2. MOSAIC sparse directory controller transitions table.
Colored cells indicate control actions: stalling the request in green and an error transition in red.

Events / *States*	GetS	GetX	Token Info	Unblock (Last Token Info from Requestor)	PUT Data	PUT Tokens	Silent Replace	Replace with tokens	Ack From LLC
I	begin reconstruction for read / C_S	begin reconstruction for write / C_X			write data in LLC	write tokens in LLC			
C_S			add sharer update num tokens known	add last sharer / A	bounce data to requestor	update tokens			
C_X				add exclusive sharer / A	bounce data to requestor	bounce tokens to requestor			
A	forward request to Owner / A_S	multicast request to all sharers / A_X			write data in LLC	update tokens	invalidate block / I	invalidate block write Tokens in LLC / A_I	
A_S				add new sharer / A	update tokens	update tokens			
A_X				remove old sharers add exclusive sharer / A	bounce data to requestor	bounce tokens to requestor			
A_I					write data in LLC	write tokens in LLC			remove block / I

Replacements that occur while the line is being constructed (C_X or C_S) or when the directory is still dealing with a request (A_S or A_X) have to be handled with care. Write requests are easier (C_X), because when the directory receives a replaced data block (event: *PUT Data*) or replaced tokens (PUT Tokens) it just forwards them to the pending requestor (action: *bounce data to requestor*). Read requests on the other hand are a little trickier, because a lot more possible situations can occur. On some occasions, the directory might be in charge of solving the pending read request with the replaced data, but it cannot be fully sure about this without more information, because it does not know whether the request has already been solved. If the reconstruction request arrived at the owner before it made its replacement, it has dealt with the pending request. If it arrived after the replacement, it could not do so because it did not have any tokens. When the replacement message arrives with all the tokens attached the answer is clear, the request was not solved and the directory needs to do so itself. On the contrary, if the replacement message does not include all the tokens, the directory controller is not able to know whether one of the missing tokens was sent to the requestor or not. The only way to know without sending extra control messages or negative acknowledgements is to finish constructing the whole entry and locate all the tokens. When the reconstruction is over, if the pending requestor did not send any token information, it means it did not receive any response and the directory needs to send one. As the reader may appreciate, all these corner cases require the addition of more states and more events indicating these situations with their corresponding extra transitions. However, they were not included in table 5-2 to avoid extra complexity for the reader and only the most common cases are illustrated. The full protocol specification may be found in [125].

Having the directory and the LLC for the same address side by side [126] gives MOSAIC a great opportunity for optimization. When a request is sent to the directory, the LLC can be accessed in parallel. If the data block is found in the LLC with all the tokens, it is possible to avoid the broadcast reconstruction request although the entry is not present in the directory. This speeds up the entry reconstruction and more importantly, it filters most of the multicast messages sent to the private caches in the CMP. As LLC capacity will be substantially higher than the number of blocks tracked by the directory, this will be the most habitual scenario for actively used private data blocks, which is the common case. Therefore, in most situations the data and all the tokens will be allocated there. If all tokens are in LLC, it is known that no other copy of the block is located in any of the private caches and the directory entry reconstruction will proceed without broadcast. Additionally, it should be noted that actively shared data (such as those associated with frequent state changes, i.e. producer-consumer scenarios) will require frequent accesses to the directory. A plain LRU replacement algorithm in this structure, even with a low associativity, will evict entries tracking private data blocks sooner.

5.2.2 In-cache directory specification

The in-cache implementation of MOSAIC has a substantial number of similarities with the sparse version. Nevertheless, its different structure means the addition of new states and in some cases the possibility of some optimizations. A LLC controller working with a MOSAIC protocol also needs to provide information about the situation each data block is in. For this reason, it is then not sufficient to define only whether an entry is constructed or not, but it is also necessary to indicate the state that valid data is in. Therefore, in contrast to the sparse design, now there are three additional possibilities, which are that a data block can be shared (S), owned (O) or modified (M). The A state is still necessary, because block sharing information may be valid (entry constructed), while data copy is not. The C states are now a group of three different states. As well as distinguishing whether the reconstruction process is started with a read request (C_S) or a write request (C_X), MOSAIC is optimized to react differently when there is an instruction fetch, in which case the entry is in C_I state. A brief description of the main states is given in table 5-3.

Table 5-3. MOSAIC protocol main states in an in-cache directory.

States	Description
I	Invalid. Block is not present in the sparse directory.
C_S	Constructing the block after receiving a read request (GetS) from a core.
C_I	Constructing the block after receiving an instruction fetch (GetI) from a core.
C_X	Constructing the block after receiving a write request (GetX) from a core.
A	Allocated. Block is fully constructed with all the coherence information about that block.
S	Shared. Block with valid data and one token.
O	Owned. Block with valid data and at least the owner token.
M	Modified. Block with valid data and all the tokens.
A_S	Allocated and a read request (GetS) has been received from a core. Waiting for an unblock message.
A_X	Allocated and a write request (GetX) has been received from a core. Waiting for an unblock message.
A_I	Invalidating a block.

Table 5-4 shows the main transitions occurring in the LLC controller. The main difference compared with the sparse directory is the existence of the C_I state whose aim is to optimize the protocol when receiving an instruction fetch. This optimization is possible thanks to having the sharing information next to each data block. For loads (non-instructions), MOSAIC always tries to send the data block along with all the tokens to the requestor in order to facilitate following writes on that block, emulating an *exclusive (E)* state behavior. If the block has all the tokens, the controller may write in it without sending any request (i.e. upgrade miss) and the more tokens it has, the easier it will be to collect the remaining ones. Moreover, avoiding maintaining tokens in LLC favors silent entry evictions in case of replacements. Therefore, when constructing an entry, if the requested data block is present in off-chip memory, it is sent with all the tokens to the requestor. However, instructions will not be written during the execution and they may be part of shared code, so it does not make sense to initially send them with all tokens to the requestor. Instead, when off-chip memory receives a reconstruction request for an instruction, it sends a copy of the block with one token to the requestor and another copy with the rest of the tokens (including the owner) to LLC. In table 5-4, when the entry is in C_I, it may receive a data block from memory (event: *data from Memory*) and when it receives the *Unblock* message from the requestor, it changes it state to O (owner). Thus, if those instructions are later requested by other cores, they will receive a copy with a token simply using a 2-hop process: request to LLC and LLC sends data to the requestor. If instructions were treated like normal reads, they would need 3 hops: request to LLC, then the request is forwarded to the owner and then the owner sends data, with the increase in latency that this would mean.

Another detail to take into account in the in-cache version is that replaced data has to be distinguished in order to know to which state the block needs to change to after writing it back. In table 5-4, only the event PUT Data appears, but note that with only this event, it is not possible to know to which state the controller has to go to when the entry is in the I or A state. Once again, the complete and detailed documentation of the coherence protocol for the in-cache design can also be found in [125].

Table 5-4. MOSAIC in-cache LLC controller transition table.
Colored cells indicate control actions: stalling the request in green and an error transition in red.

	GetS	GetI (Get Instruction)	GetX	Data from Memory	Token Info	Unblock (Last Token Info from Requestor)	PUT Data	PUT Tokens	Replacement
I	begin reconstruction for read / C_S	begin reconstruction for instruction / C_I	begin reconstruction for write / C_X				update data / M/O	update tokens / A	
C_S					add sharer update num tokens known	add last sharer / A	bounce data to requestor	update tokens	
C_I				update Data	add sharer update num tokens known	add last sharer / O	write data bounce data + 1 token to requestor	update tokens	
C_X						add exclusive sharer / A	bounce data to requestor	bounce tokens to requestor	
A	forward request to Owner / A_S	forward request to Owner / A_S	multicast request to all sharers / A_X				update data + tokens / M/O	update tokens	invalidate block / I
S	forward request to Owner	forward request to Owner	send tokens multicast request to all sharers				update data / O	update tokens	replace Tokens / I
O	send Data + all tokens	send Data + 1token	send Data + all tokens multicast request to all sharers					update tokens	replace Data / I
M	send Data + all tokens	send Data + 1token	send Data + all tokens						replace Data / I
A_S						add new sharer / A	update tokens	update tokens	
A_X						remove old sharers add exclusive sharer / A	bounce data to requestor	bounce tokens to requestor	

One of the main disadvantages of the in-cache structure is that, in some cases, replacements cannot be silent. When it is necessary to construct a line and there is no available space for it in the LLC, the coherence protocol needs to replace one block to construct a new one. If the data block is in the A state, the eviction can be made silently, but if it has some tokens, it has to replace this tokens writing them back in off-chip memory. This did not occur in the sparse version of MOSAIC where a construction of a line did not mean an eviction from the LLC.

5.3 Detailed examples

Now that the reader has a vision of the details of the coherence protocol, we can review the conceptual approach seen at the beginning of this chapter, but focusing on precisely describing what happens with the entry states and the rest of the copies in the system.

Figure 5-2 and figure 5-3 show the representation of two consecutive reads in a 4-core CMP with a MOSAIC sparse directory. We have added one additional processor (P_3) to the conceptual approach example in order to be able to see the behavior when there is a second read after the line has been reconstructed. The initial situation is with P_1 having the data block with all the tokens except for one, which is in P_2's private cache with another copy of the data block. P_0 issues a read request (*GetS*) to the directory ❶ because it does not have the data block in its private cache (which might be composed of multiple levels). The directory does not have any entry allocated for the requested address so it broadcasts a reconstruction message ❷ asking for

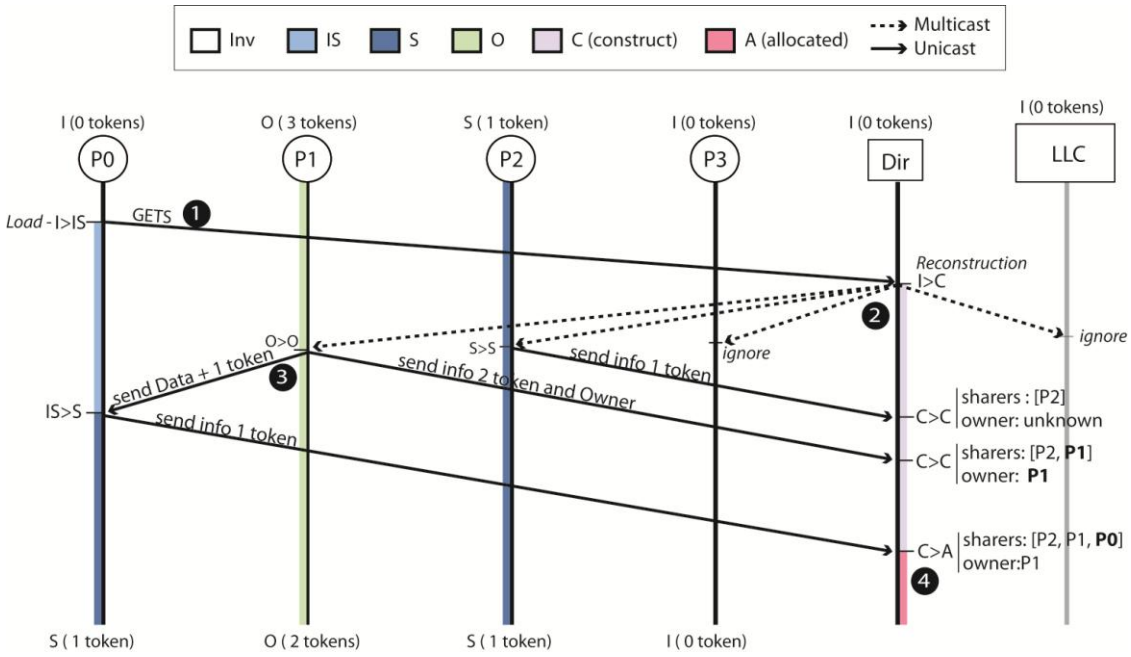

Figure 5-2. Example of MOSAIC coherence protocol when a read request arrives at the directory and no entry for the requested block is allocated. P_0 issues a GetS operation and the directory has to initiate the reconstruction process.

all the token information and indicating that P_0 needs a copy of the data block with at least one token. Processors that do not have any token ignore the request (like P_3) and processors with the data block in a shared state (such as P_2) send information about how many tokens they have. The processor holding the owner token (in this case P_1) is in charge of solving the initial request, so it sends a copy of the data block with one token to P_0 ❸ and sends information about all the tokens left to the directory. While the directory is receiving messages with the token location information, it updates the sharers vector and it increases the number of known tokens that it has received so far. It will also receive information about which processor holds the owner token. Thus, when it knows where all the tokens are and who the owner of the block is, the directory is able to ensure that the entry information is completed. In our example, this occurs when the last token information arrives from the requestor ❹, when the directory can change the state to A indicating that the entry is allocated with all the information updated. Using token counting is a key component in MOSAIC, because it simplifies all the handshaking used to reconstruct directory entries and it avoids the use of negative acknowledgements as well as the necessity of timeouts. Since the directory behaves like a serialization point, concurrent operations initiated by different processors for the same cache block will never end up suffering starvation. In this way, the directory coherence controller avoids these problems without requiring persistent request [17] or added token tracking facilities (chapter 4) [127][71].

After this process, any other request for that address arriving at the directory will find the entry fully reconstructed and it can be dealt with directory, like in a conventional directory protocol. This situation is shown in figure 5-3 where, using the final situation of the previous figure as the starting point, P_3 issues another read request to the directory ❶. This time, when the request

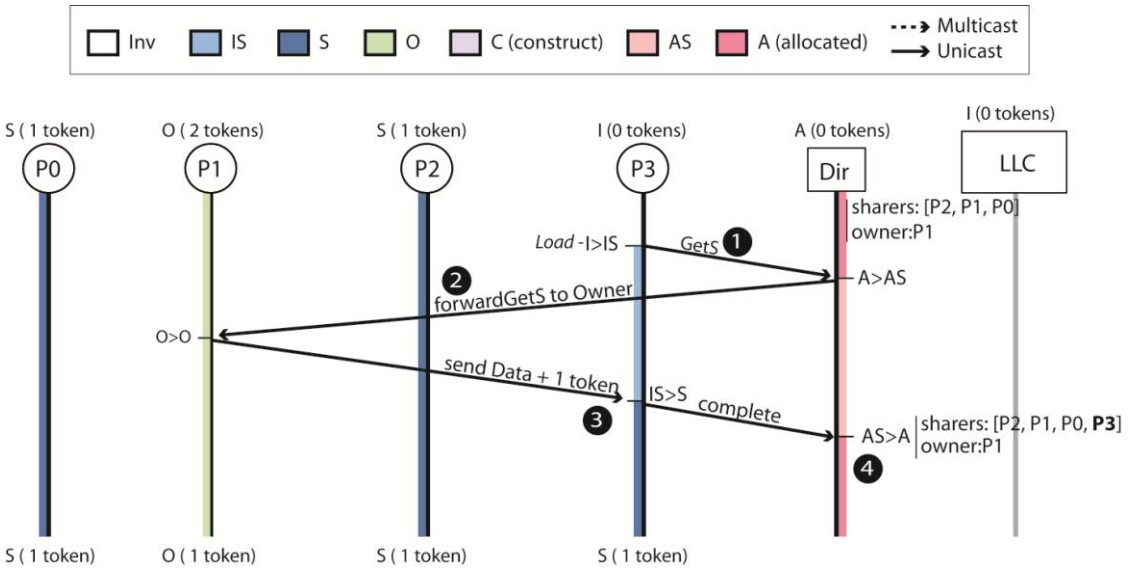

Figure 5-3. Example of MOSAIC coherence protocol when a read request arrives at the directory and it finds the entry for the requested block constructed with all the coherence information.

arrives at the directory, the line is fully constructed and it includes all the necessary coherence information. Therefore, as it knows that the owner of that data block is P_1, it only has to forward ❷ the read request to P_1 and the owner will reply to P_3 with a data block copy and one token ❸. If the request was a write request, this unicast message would be a multicast message to all the sharers to invalidate their copy (P_0, P_1 and P_2 in this case). After P_3 finishes its read request, it sends a complete message to the directory, which will add it as another sharer and set the entry state back to the stable state A ❹.

In order to see how MOSAIC behaves for a write request, we will repeat the example of the reconstruction process in figure 5-2, but with write request instead. Figure 5-4 shows how P_0

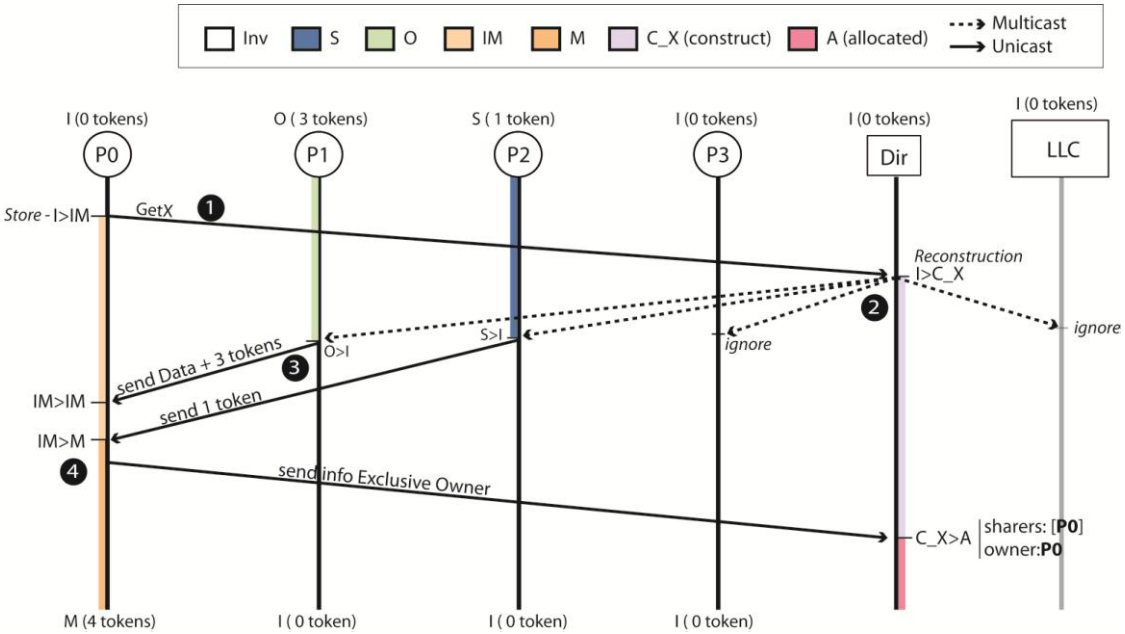

Figure 5-4. Example of MOSAIC coherence protocol when a write request arrives at the directory and it does not find the entry for the requested block constructed with all the coherence information.

issues a *GetX* after a store miss ❶. When it arrives at the directory and it does not have an entry allocated, it starts the reconstruction process as it did for the read request ❷. However, as this reconstruction message indicates that a write request from P_0 started the process, the private caches with allocated tokens will not inform the directory about their location, but instead they will forward all of them to the requestor. Therefore, P_1, as the owner of the block, sends a copy of the data block with all its tokens to P_0 invalidating its copy of the block ❸. P_2 sends its token as well (no data attached because it is not the owner). When P_0 receives the last token assigned to the requested block, it can ensure there are no more copies of that block in the system and it can finish with its pending store operation. It also sends a message to the directory indicating that it is the exclusive owner of the block and so the directory can finish the construction process.

Although the number of different situations is large, (in fact, all state transitions shown in the table definition should be analyzed) we believe that these examples show the essential aspects of the behavior of the protocol. The next step in developing MOSAIC was its performance evaluation under real applications.

5.4 Evaluation

It has been shown in the previous chapter that for sizes of systems such as those with 8 or 16 cores, a broadcast-based protocol like LOCKE might perform better than a directory protocol and in order to check MOSAIC's capability to overcome classic directory limitation, it would be necessary to simulate much higher numbers of cores in the CMP. Nevertheless, to evaluate

Table 5-5. Memory system configuration of 8-core CMP (and 16-core CMP).

Private caches	L1	Size	32KB Instructions and Data
		Associativity	2-way
		Access Time	1 cycles
	L2	Size	64KB
		Associativity	4-way
		Access Time	2 cycles
		Exclusive with L1	
Shared cache	L3	Size	*config1*: 16MB *config2*: 32MB
		Associativity	16-way
		Data Slice Size	1MB
		NUCA Mapping	Static, interleaved by LSB
		Access time	1 MB / 6 cycles

systems with tens or hundreds of cores is unfeasible with current evaluation tools because of the computational effort of this task and the limited availability of scalable workloads. For this reason, in this section MOSAIC's directory properties will be varied, like its associativity and capacity, reaching values that may seem unrealistic, but that will allow us to extrapolate the results to a much larger number of cores. Similarly, studying the evolution of the benefits and drawbacks of 8-core CMP compared to 16-core CMP will also allow us to glimpse the scalability of the idea with a higher number of cores.

The majority of the following analysis will be focused on the sparse directory design, since it has less storage overhead [126][128] and the cache evictions due to directory conflicts have lower impact on performance.

As was done with LOCKE, we will use aggressive out-of-order cores to analyze MOSAIC. We will use the same processor configuration values that we used to evaluate the LOCKE coherence protocol (table 4-5), introducing changes in the memory configuration as shown in table 5-5.

We decided to simulate the same two layouts of the 8-core and 16-core CMP used in LOCKE, which use a 4x4 (figure 4-6) and 6x6 mesh topology respectively. As both figures show, the notable difference is that now the on-chip hierarchy configuration is composed of three levels (figure 5-5). We reduce the size of the first level of cache and we introduce an additional private L2 cache. The LLC is kept as a shared static NUCA [111] connected with a mesh network. The workloads described in the appendix will be used.

5.4.1 Impact of Directory Configuration on Performance

When the number of cores is large, conventional sparse directory protocols have to face limitations in two main factors, *capacity* and *associativity*. MOSAIC's sensitivity to both parameters is analyzed next and its results are compared with those from a conventional sparse

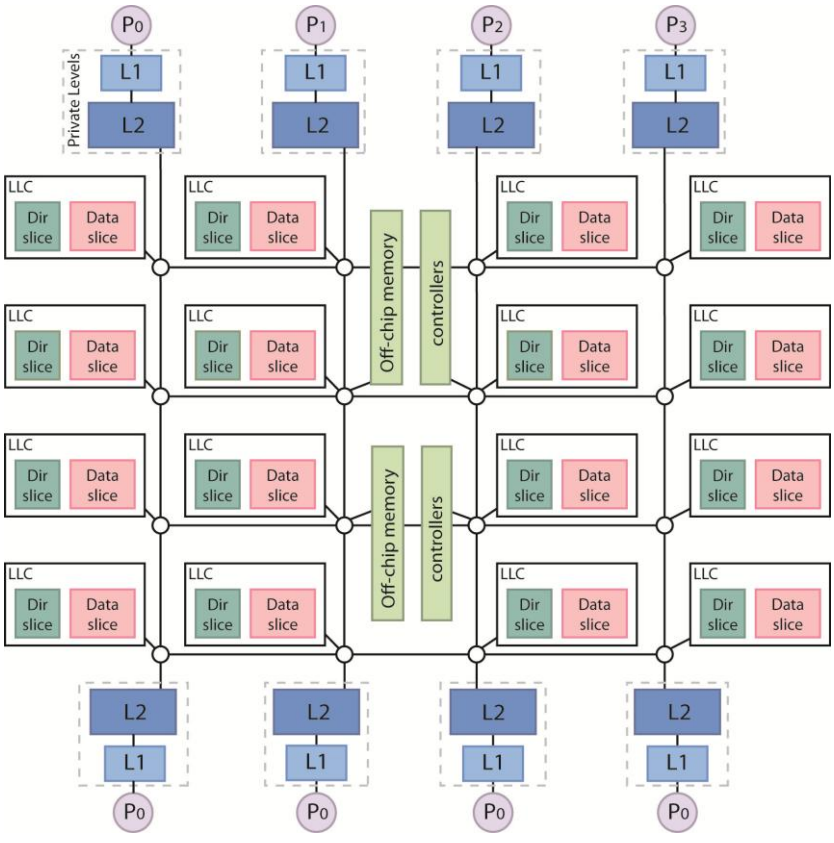

Figure 5-5. Layout of the 8-core CMP simulated with MOSAIC.

directory implementation. The **reference point** in this analysis will be a directory with duplicate tags. Since under this configuration there will be no private cache invalidations due to directory misses, there will be no performance differences between MOSAIC and conventional protocols. We will start with small private caches of a 2-way 32 KB L1 I/D and a unified 4-way victim L2 cache of 64KB. Assuming in both cases a block size of 64 bytes, L1 caches have 512 entries each and L2 caches have 1024 entries each. This means that the number of entries required in the directory to avoid capacity misses (to have space for all the tags) is *2048*#cores*. In the CMP simulated, *#cores* correspond to eight cores (except in the scalability analysis). Therefore, assuming 8 bytes per directory entry (enough to store the tag and the sharing information), the total directory size required to avoid capacity misses will be 128KB (2048 entries × 8 bytes/entry × 8 cores). With the aim of minimizing the access time to data in data slices and avoiding access contention, the directory is distributed in 16 slices (as many slices as the LLC). The slice interleaving of data and directory entries over LLC uses the least significant bits of the address. For the same addresses, the directory slice and data slice are 1 cycle apart. As was mentioned in chapter 2, section 2.5.2, to avoid any conflict misses in the directory, the required associativity will be 64. This large associativity is necessary because on each entry we need as many ways as the sum of both of the private levels' associativity times the number of cores (i.e. *(L1I associativity + L1D associativity + L2 associativity) * #cores*).

5.4.1.1 Sensitivity to Conflict Misses in the Directory.

Initially, the sensitivity of a conventional directory protocol and MOSAIC when the associativity of the sparse directory is reduced will be determined. This will help us undersand how the two protocols will react when the number of conflict misses in the directory increases significantly. In order to perform this analysis, we keep the directory capacity fixed at 128KB and modify the associativity from 64-way to 1-way per set. As the associativity goes down the number of conflicts grows, because even though there is space for all potential blocks stored in private caches, some of them may conflict in the directory.

Figure 5-6. Normalized number of misses in the private levels when sparse directory associativity is changed for a conventional coherence protocol (BASE) and MOSAIC.

Figure 5-6 shows how the base Directory protocol (from now on it will be denoted as BASE) and MOSAIC impact on the cache level behavior when the number of directory conflicts is increased. While MOSAIC is completely insensitive to any associativity modifications, BASE directory has an adverse reaction to that change, causing a large number of misses in private levels due to private cache invalidations induced by directory conflicts. In some applications, such as *Omnetpp* (where the cores are not sharing any data), the misses in those levels are multiplied by two.

The final differences in performance depend on each type of application, i.e. its behavior in private caches using a duplicate tag directory. Figure 5-7 shows these results, indicating that the MOSAIC protocol could be up to 40% faster than the BASE protocol. For the combination of system size and applications used, the most remarkable effects are found in extreme situations when even with capacity to track all private blocks, the performance will fall, on average 12% when restricting the associativity of the directory to 1 way per set. MOSAIC overcomes the

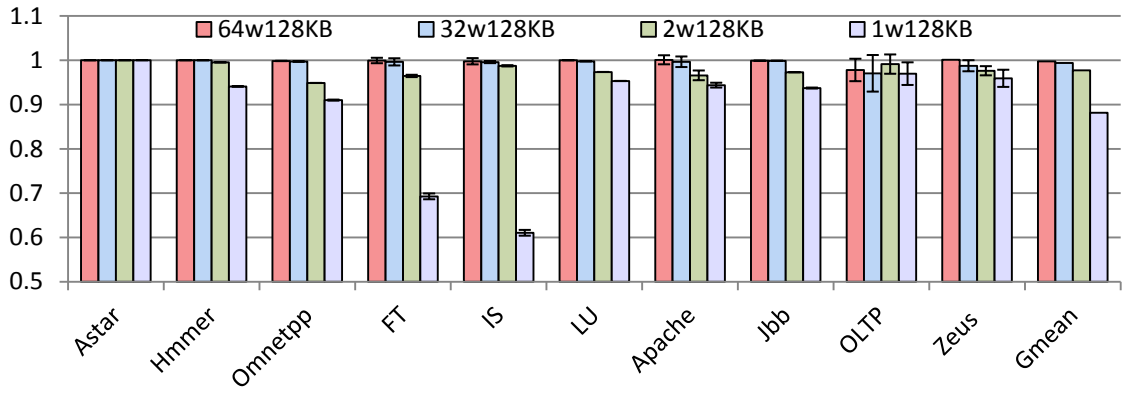

Figure 5-7. MOSAIC execution time normalized to BASE, while varying the associativity of a fully sized sparse directory (i.e.16K entries).

problem of having a limited associativity (major issue in directory protocols [128]) since a simple direct mapped directory is capable of maintaining the performance and even in some cases improving it.

5.4.1.2 Sensitivity to Capacity and Conflict Misses in the Directory

The second effect that might influence performance is the number of capacity misses in the sparse directory. The combination of capacity misses induced by limited directory storage as well as the associativity reduction seen in the previous section will increase total conflict misses. To compare how both effects might impact on each protocol, we reproduce the previous analysis, but reducing the directory capability to *track only an eighth* of the private caches capacity, i.e. up to 2K blocks. Figure 5-8 reproduces the results provided in figure 5-7 with the new directory capacity. In this new configuration, misses in private caches for BASE are substantially higher than in MOSAIC (Figure 5-9). Even with an associativity of 64, after reducing the size of the directory, capacity conflicts have a relevant impact on performance, degrading it up by up to 20%. These capacity misses seem to be more relevant in applications with a higher sharing degree (i.e. commercial workloads [129]) since the number of misses in

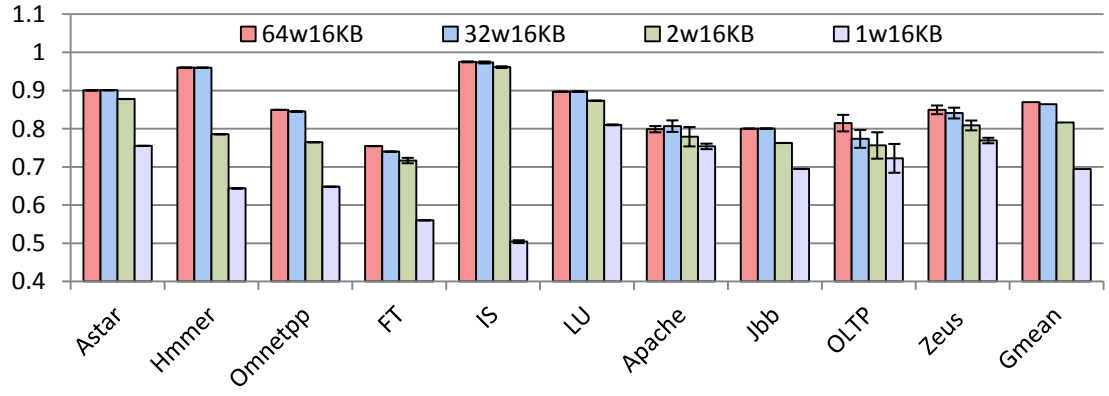

Figure 5-8. MOSAIC execution time normalized to BASE while varying the associativity for a directory with one eighth of fully sized sparse directory (i.e., 2K entries).

the private levels is much fewer, while applications with a reduced working set are less sensitive to this capacity reduction, such as *hmmer* which only shows a 2% improvement.

However, the associativity reduction now has a greater influence on performance, especially on those applications where there is no sharing data at all. This happens because the cores are using completely different data and so they all need different entries allocated in the same set of the directory, but as the number of ways is being limited, there is not enough space for all the tags. For this reason, the directory is continuously replacing the tags allocated when receiving new requests.

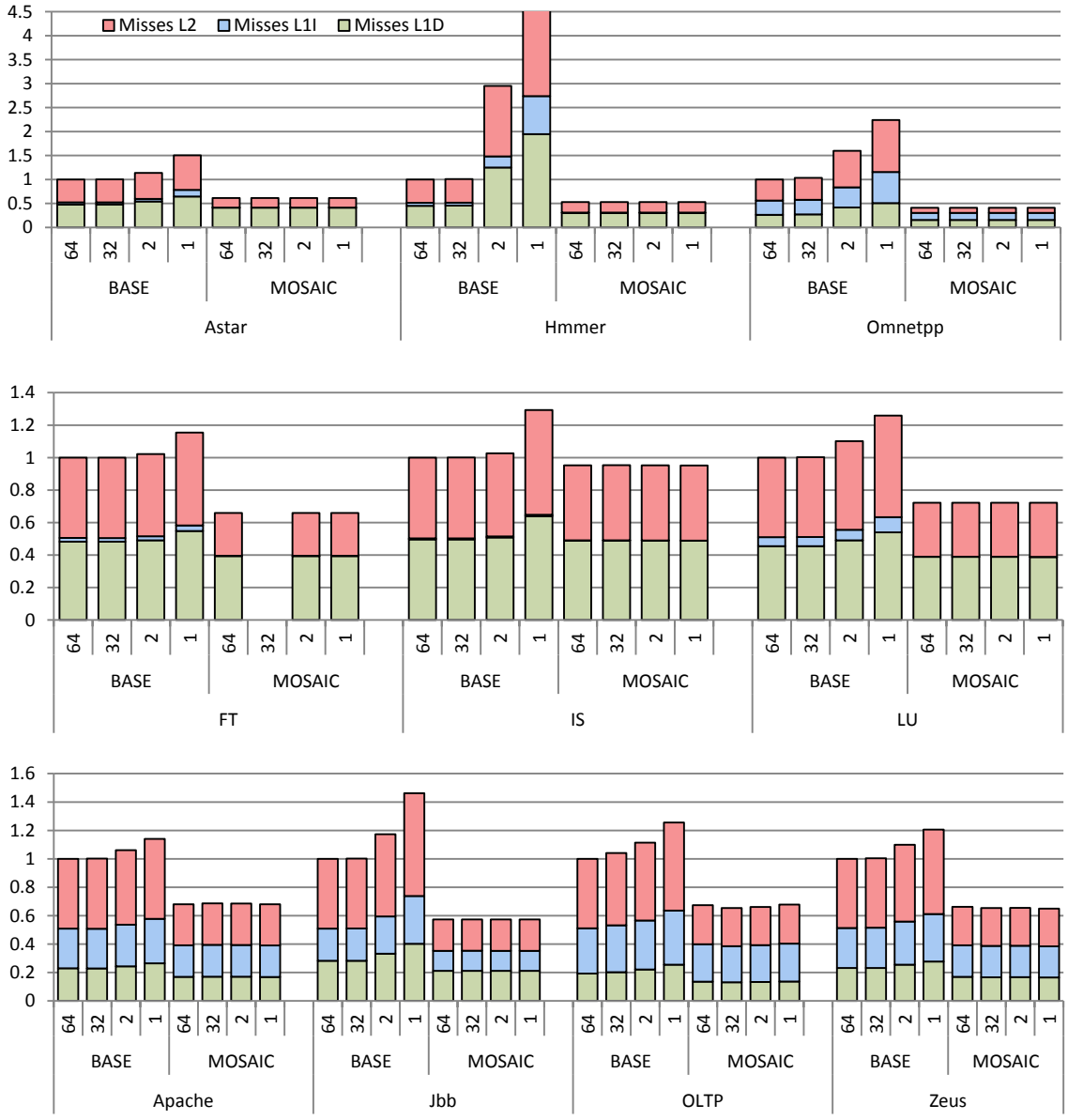

Figure 5-9. Normalized number of misses in the private levels when sparse directory associativity and capacity is changed for a conventional coherence protocol (BASE)

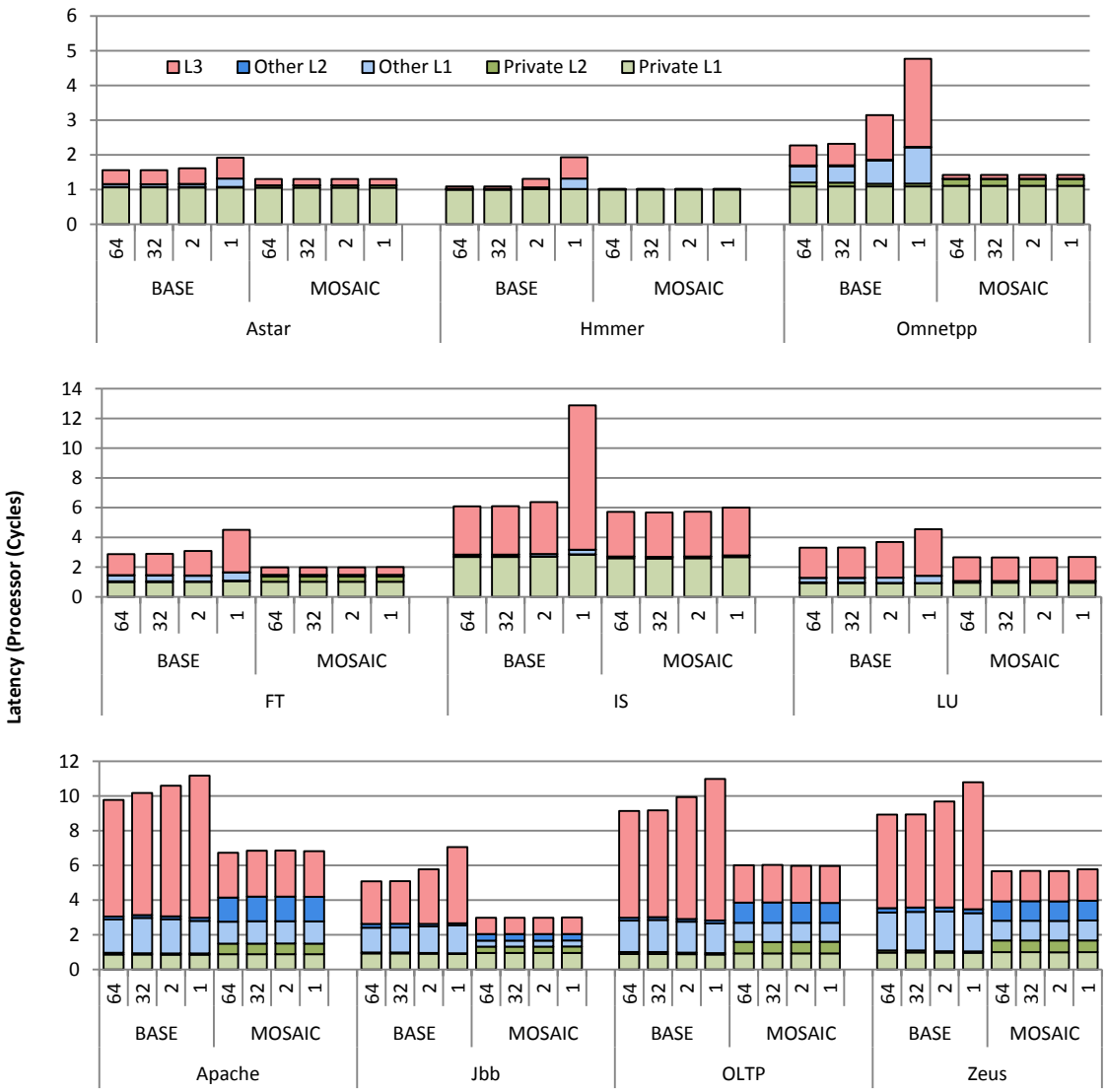

Figure 5-10. Average on-chip latency for a 16KB (2K entry) sparse directory when varying its associativity.

When this happens in the BASE directory, data allocated in the private levels that are used by the processors are invalidated, thus increasing the number of misses in the private levels noticeably, as can be seen in figure 5-9. On the contrary, MOSAIC replaces silently without invalidating, allowing those blocks to stay in the private levels until the processors replaces them.

To better understand how directory invalidations influence each protocol, figure 5-10 provides the average access time for on-chip hits. Again, the dissimilar behavior of the two protocols is notable. In some applications, MOSAIC shows half of the on-chip latency of BASE due to the extra misses in private caches in the latter. Those requests are mostly resolved by LLC with extra added latency, which explains its growing contribution when the directory-caused evictions in the private caches are more relevant. With MOSAIC, all the applications demonstrate a higher contribution of the private L2.

Moreover, for applications with a high sharing degree, the broadcast reconstruction message favors the forwarding between caches as the *Other* L1 and *Other* L2 contributions show, and so avoids an access to L3 as the conventional directory does. The steady miss latency values obtained demonstrate MOSAIC's stability even in the most extreme configurations, a direct-mapped directory with capacity to track just an eighth of the private cache blocks.

5.4.1.3 Sensitivity to Directory Size in a Realistic Private Cache configuration.

Up to now, we have been using limited private cache capacity and associativity. If we consider the configuration of commercial systems [4][130][131], L2 caches have between 1/8 and 1/4 of L3 capacity and both L1 and L2 have a larger associativity. Therefore, it is important to carry out a sensitivity analysis for the size of the directory with a realistic configuration for private caches. In this particular case, we try to mimic the L2 cache configuration in Intel's Nehalem [11] (4-way 32 KB of L1s and 8-way 256 KB of L2). We will keep the associativity fixed at 16-way (like in the data banks) and vary the capacity of the directory, from double [126] the full

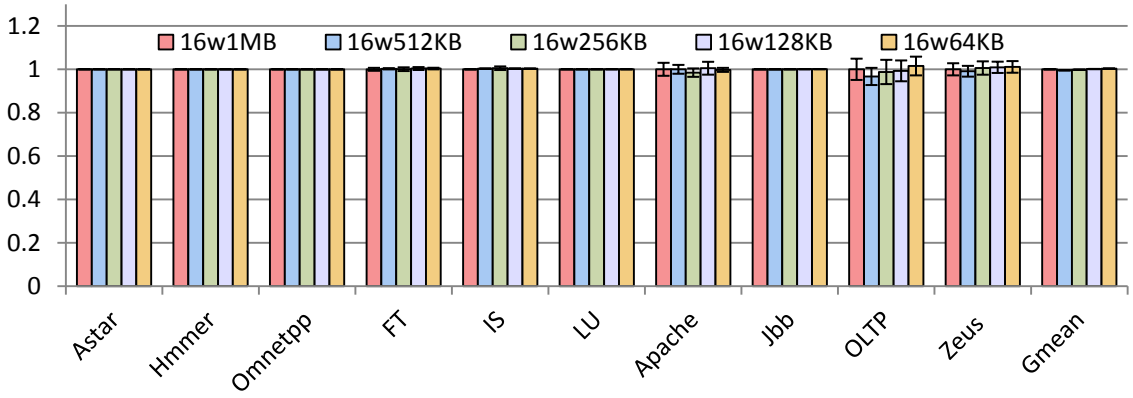

Figure 5-11. MOSAIC execution time normalized to Duplicate Tag Directory, for a Nehalem-like private caches configuration varying directory capacity.

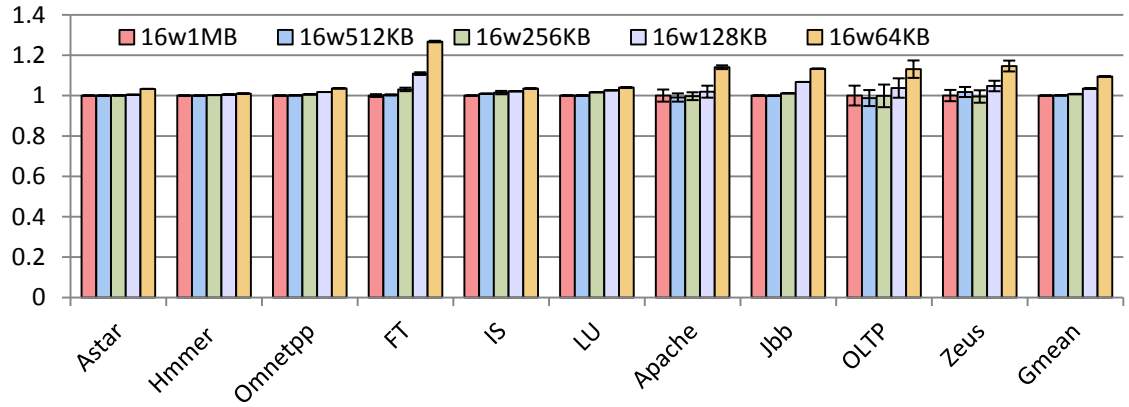

Figure 5-12. BASE execution time normalized to Duplicate Tag Directory, for a Nehalem-like private caches configuration varying directory capacity.

directory to one eighth of the full directory. For these private cache sizes, the duplicate-tag directory would need to track 5120 blocks for each core, needing 40960 entries in total. Considering the size of each entry to be 8 Bytes, it is necessary to have 320KB of space dedicated to the directory. To maintain the directory size as a power of 2, the total duplicate-tag size is established at 512 KB. Figure 5-11 show the average execution time for each application normalized to the double-sized directory (i.e. 1MB) where even with the smallest capacity, there is no performance impact. As can be seen in figure 5-12, when reproducing this same experiment for the BASE protocol, the performance impact is greater than 20% in some cases.

5.4.2 Cost Analysis: Bandwidth and Energy Overhead of MOSAIC

In light of the previous results, in contrast to a BASE protocol, MOSAIC's behavior is fairly independent of the directory configuration. Since the rationale of MOSAIC is to trade directory cost for on-chip bandwidth and additional snoops in private caches, we need to analyze the energy overheads. The first step in this analysis is to quantify how directory cost reduction influences the on-chip bandwidth consumption. If the network is using routers with support for handling multicast traffic [32], the real measure of bandwidth and energy consumption for the interconnection network is given by the average link utilization and not the end-point traffic consumption (see previous chapter). Figure 5-13 shows the average link utilization for the initial configuration (i.e. exclusive 32KB L1 and 64KB L2) when the capacity of the directory or its associativity is reduced. The values are normalized for a duplicate tag directory, i.e. capacity for 16K entries (128KB) and 64-way associative. The results show that on average and under the worst conditions (i.e. a 2-way associative directory, with an eighth of the capacity of the full directory) the traffic is just 5% higher than a duplicate tag directory.

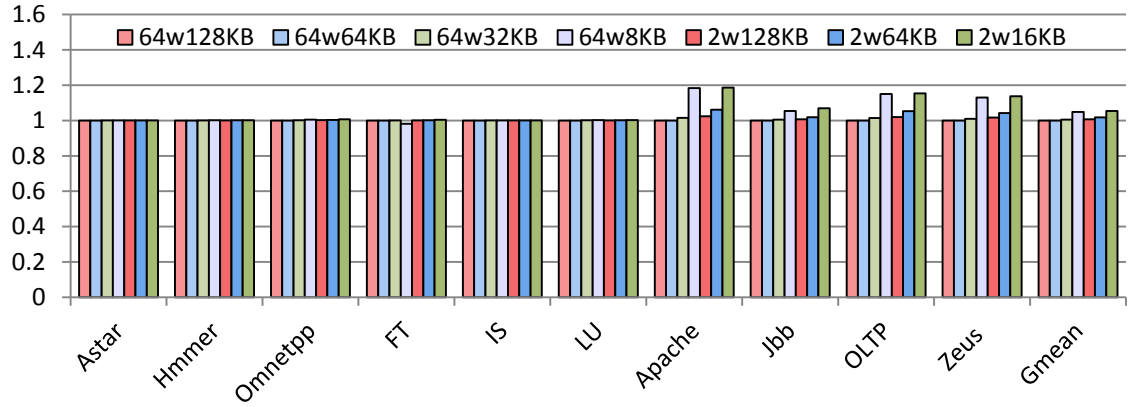

Figure 5-13. Average network link utilization of MOSAIC normalized to a duplicate tag directory, varying directory capacity and associativity.

Focusing our attention on each class of applications, multi-programmed workloads are completely insensitive to directory configuration. Since in these applications there is no information shared between the cores, this is the expected behavior. More noteworthy is the behavior of scientific applications, where there is a substantial amount of shared and highly contended data. In such cases, the directory replacement algorithm prevents the eviction of actively shared data and entries of private blocks are more prone to being replaced.

Consequently, traffic does not change. Server workloads seem to be the most sensitive, since in this case the amount of shared data is large, most of them being code. Therefore these blocks will be accessed in read-only mode and the directory will be less frequently accessed. As a consequence, the chances of evicting an actively shared entry are higher than in numerical applications and so too are the chances of requiring a multicast to reconstruct these entries. Nevertheless, even in the most adverse (and unpractical) directory configurations, this increment is less than 20%, which is substantially less than in broadcast coherence protocols [127][71][17].

The reason for this behavior is that multicast is only generated when, after a miss in the sparse directory, the data and tokens available in LLC are not enough to fully reconstruct the sharing information. If the block has all the tokens, it can be ensure that there are no copies in any private caches and consequently the multicast can be avoided. Since LLC can be very large, the most usual case will be this one and, therefore, multicast will be required only if the data is really shared. In contrast, if we compare the bandwidth consumption of MOSAIC and BASE protocols when the directory is simplified, the results are very different. As figure 5-14 indicates, the BASE protocol requires more on-chip bandwidth in most cases, especially when the directory is highly limited. In the most extreme case, i.e. a 16KB, 2-way associative directory, BASE requires up to 40% extra bandwidth consumption on average. The main reason for this is that MOSAIC has fewer misses in the private caches and directory evictions are silent. For instance, in SPEC applications all processors have mostly independent executions so the

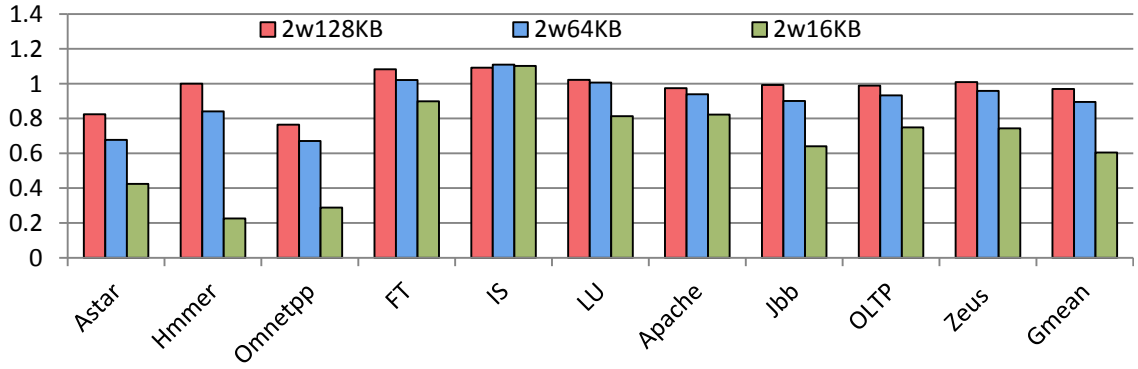

Figure 5-14. Average network link utilization of MOSAIC normalized to BASE directory.

conflicts that occur in the sparse directory with a conventional directory protocol induce a large number of invalidation messages to the private levels. These invalidation requests replace the data needed by the processors, which may still be useful. Subsequent misses will require extra communication with the directory. In contrast, MOSAIC leaves these data in the private levels avoiding extra misses in the sparse directory because it is private data and so it will not be requested again, by this means avoiding requests and data travelling back and forward through the network. When the difference in the number of misses between the two protocols is small and applications have a high sharing degree, broadcast messages of the reconstruction requests are more noticeable. With highly contended shared data, such as in numerical applications, the replacement algorithm of the directory inhibits evictions of actively used data and therefore the external invalidations in caches with BASE are fewer (at least with directory configurations that are not highly constrained). Under this configuration MOSAIC memory misses might increase the traffic due to the multicast traffic required to deal with them. Although this multicast traffic might be avoided using simple solutions such as [132], it seems irrelevant in most applications. The most relevant case is *IS*, which has a large MPKI. Even in these cases, the extra traffic is less than 10%. In server applications, shared blocks rarely change their state (from S) and they have the same probability to be evicted as private data blocks. Consequently, the number of invalidations of useful data in private caches is larger. The result is that the extra traffic required to deal with this situation is much greater than with MOSAIC.

The previous discussion partially addresses the potential added costs. To complete it, we need to look at the energy consumption, with emphasis on the cache hierarchy. Results of this analysis are shown for both protocols in figure 5-15 when using a 2-way associative sparse directory with three different sizes: 128KB, 64KB and 16KB. The results have been normalized to 128KB and a 2-way directory size of BASE protocol. The results are coherent with the traffic results: MOSAIC reacts in a more energy efficient way than the BASE protocol when the directory size is constrained. Therefore, we can conclude that the extra costs derived in the bandwidth-directory tradeoff overhead are favorable in our proposal.

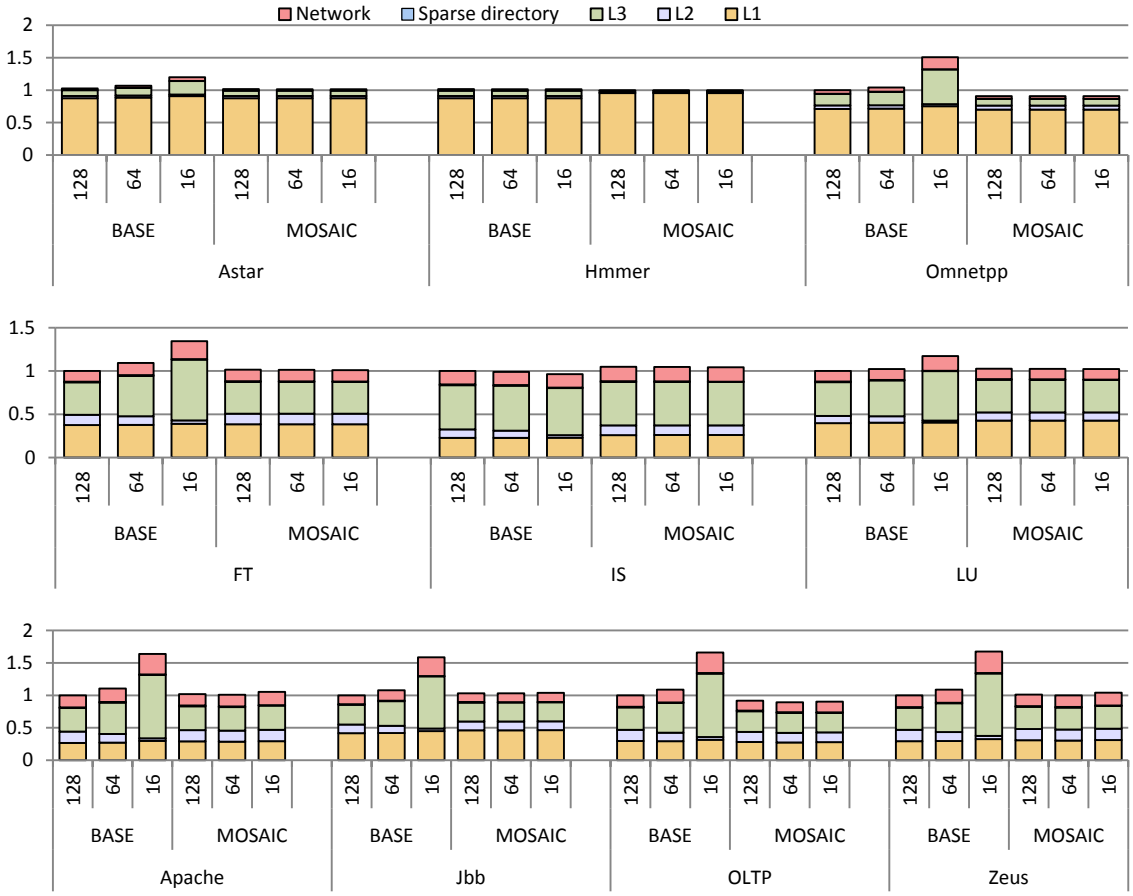

Figure 5-15. Normalized dynamic energy used by caches and network normalized to the directory-based coherence protocol with an aggregate 128KB sparse directory.

Different sizes: 128KB, 64KB and 16KB (8, 4 and 1 KB per slice).

5.4.3 Scalability Analysis

To complete the cost analysis, MOSAIC's reaction in a CMP with 16 cores is studied. In this system configuration the number of LLC banks is doubled and a 6×6 mesh is used to connect them with private caches and four memory controllers. The remaining configuration parameters are maintained unchanged. To scale on-chip cache bandwidth, the number of banks and consequently the network has to be scaled up [111]. Comparing the results in figure 5-13 and figure 5-16, it can be seen that the differences are unnoticeable for most of the applications, even in extreme situations such as the one corresponding to a 2-way set associative directory with capacity to track an eighth of the private caches, which has only 7% more link utilization on average than a Duplicate Tag Directory. As with the 8-core CMP, the server applications, due to their high sharing degree of read-only data, are the most sensitive to directory structure. Even in these cases, with a quarter of the directory capacity, the average extra traffic is less than 10%.

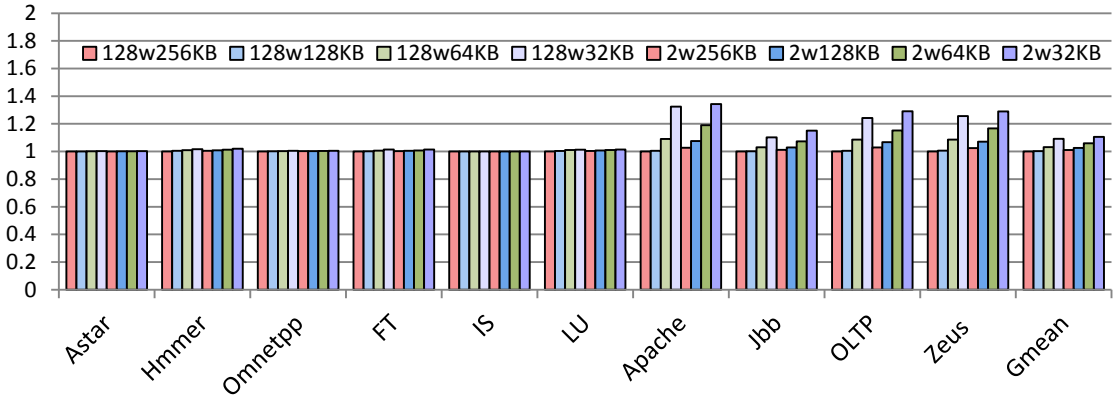

Figure 5-16. Average link utilization of MOSAIC normalized to a Duplicate Tag Directory (128-way associative, 256KB), varying directory capacity and associativity) in a 16-core CMP.

When checking the performance comparison between the two protocols, MOSAIC's stability is remarkable while BASE suffers up to 60% degradation of the execution time when varying the size and associativity of the sparse directory (figure 5-17). These results with 16 cores are even better than those obtained in the 8-core CMP configuration. The reason for this is that misses (due to directory invalidations) in private caches take longer to be resolved in LLC due to the larger size of the system.

Given the complexity of the evaluation environment and the architecture of the system evaluated, it is not possible for us to increase the number of cores simulated beyond this point. Nevertheless, comparing the evolution from 8 to 16-core CMP systems, we can infer that the progression with larger numbers of cores should be similar. Since extra traffic will be proportional to the number of cores, the bandwidth overhead compared with an unfeasible Duplicate Tag Directory in bigger CMPs or with more realistic private cache hierarchies will be

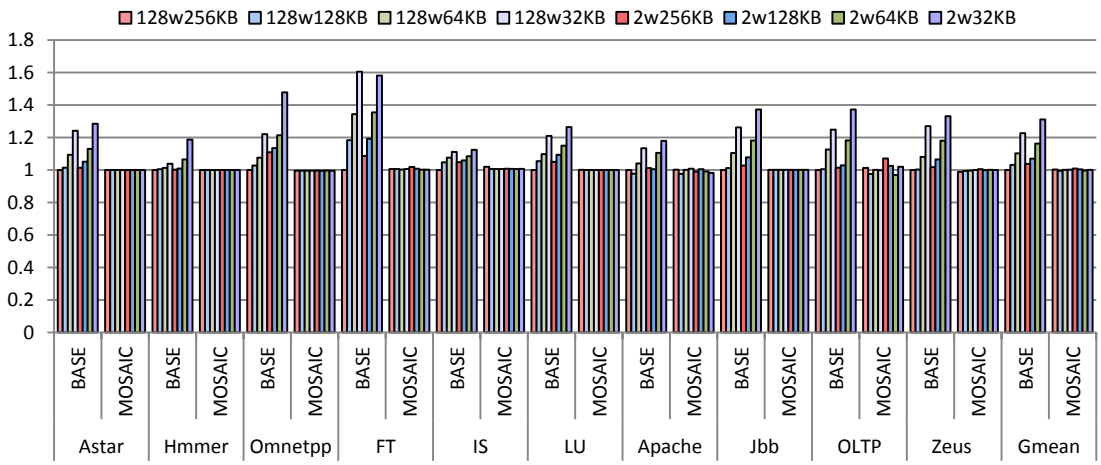

Figure 5-17. Execution time of BASE and MOSAIC normalized to a Duplicate Tag Directory (128-way associative, 256KB), varying directory capacity and associativity in a 16-core CMP.

similar. Finally, note that to prevent on-chip and off-chip bandwidth impact on performance when increasing the number of cores in the chip, on-chip interconnection network bandwidth has to be extended [133]. In our particular case the bisection bandwidth has increased 50%, (from 4 to 6 bidirectional links), which is substantially larger than MOSAIC's traffic overhead in the most unfavorable directory configurations. Consequently, it seems reasonable to assume that MOSAIC will scale up for much larger systems.

5.4.4 In-cache analysis

Although the previous results have been focused on a sparse directory configuration, MOSAIC has also been implemented in in-cache architecture. It was mentioned in previous chapters that this cache design has several advantages when compared to the sparse directory. As well as the simplicity of the coherence protocol increasing considerably, it avoids having to duplicate cache block tags in order to keep the sharing information in a standalone structure. However, when the aggregate private cache capacity grows, the in-cache design is seriously affected, because of the inclusiveness that has to be maintained to store all the private cache tags. Under these circumstances, a coherence protocol such as MOSAIC, where inclusiveness is not an essential characteristic to guarantee correct function, can make this type of design the best choice.

Therefore, when comparing in-cache MOSAIC to BASE when the relation between the total amount of private cache capacity and the LLC size is closer to 1, i.e. the same size in both of them, MOSAIC's advantage is greater. This is so because the majority of the entries in LLC are used to track blocks in the private caches and when there is a replacement in the LLC, MOSAIC does not invalidate any data block allocated in the private cache, while BASE does. Figure 5-18 shows the execution time of MOSAIC normalized to the BASE directory with two different ratio

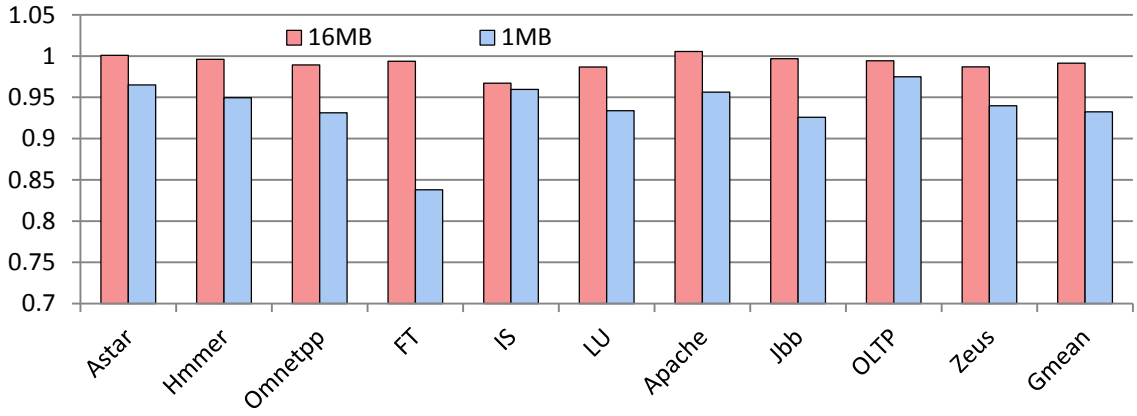

Figure 5-18. Execution time of MOSAIC normalized to BASE directory when using in-cache MOSAIC in a 8-core CMP and varying the LLC size.

sizes: 1/16 and 1. Considering the initial 8-core configuration (32KB of instruction and data L1 and 64 KB of L2), when the size of the LLC is set to 1MB all the entries in the LLC could be used to track all the data blocks of the private levels. However, this would mean that the LLC will not have any capacity that could be used as a victim cache. Therefore, when the difference in size of the aggregate capacity and the LLC size is very large, both protocols react the same way and there is no difference between their execution time. However, when this ratio is 1, MOSAIC obtains better performance results because it does not have to invalidate any private copy whenever it has to replace an entry in LLC, while the BASE directory does have to.

5.5 Future optimizations in MOSAIC

After the analysis done for both architectures when using MOSAIC and, having verified the good behavior of our proposal, we have noticed several additional features that can further improve the MOSAIC coherence protocol.

One of these optimizations has to do with the different treatment of private and shared data. We have observed that the protocol could obtain benefits from the fact that most of the data blocks for which the line is being reconstructed are private data, i.e. they are requested by one processor and they will not be shared with others later. For this reason, it is possible to modify the original protocol to avoid replacing an entry in the directory when the arriving request is for a private data block. As the directory is not able to know when a data block is private or not, it would be necessary to modify the directory controller to take decisions while receiving token location information from the potential sharers. Thus, when the directory receives a request, it could initiate the reconstruction process as before, but without replacing any existing line. If it receives any reply for that reconstruction process with information about not all the tokens, it can assume that the data block is present in at least one cache. This means that it already is or it will be a shared block and it will have to replace an entry to make space for the new reconstructed entry. On the other hand, if it only receives one reply from the requestor itself informing that it has all the tokens in an exclusive way, the directory will know that the requestor received the data from the LLC or from off-chip memory, becoming a private data block and making it unnecessary to replace any existing entry. The sparse design will directly benefit from this additional feature since it reduces the impact that the size of the directory might have on the system performance and all the private data blocks will not need to have a line in the directory. This will reduce the number of misses of those requests that need the coherence information in the directory. On the contrary, to obtain benefits in the in-cache design from this optimization, it would have to be complemented with some *cache bypassing* [97][96] to avoid replacing, in the LLC, private data blocks removed from the private caches that have

low probability of being used again. This will reduce the number of cache writes of private data blocks which are not used once the private caches evict them.

Another possible way to optimize MOSAIC is by adding a filter to reduce the on-chip traffic. Each of the entries in the sparse directory could include a bloom filter [86]. This filter could include information about which data blocks matching that entry are allocated inside the chip. Whenever a data block is present in the private caches, the filter is set and in this way, when a miss occurs in the directory and after checking that requested data is not present in the LLC, the controller could check in the filter whether it has to reconstruct the line by broadcasting the request, or just send it to off-chip memory. To unset the filter for a specific block, it is necessary to know when the data block is sent to off-chip memory. For this purpose, we can use token counting and only permit replacements of data blocks present in the LLC with all the tokens collected. Lastly, the off-chip traffic could also be reduced with an additional filter such as the one explained previously, but associated with the blocks which are certainly allocated in the private levels. This is necessary because the on-chip filter will have false positives, i.e. it says that a requested block is present in private caches, but it is not. This means that for these cases it will be necessary to broadcast to all the private caches and also to memory in case the filter is incorrect. In this way, the MOSAIC reconstruction process will be more accurate and the total amount of traffic could be reduced.

5.6 Conclusions

A new coherence protocol that addresses the challenges of complex multilevel cache hierarchies in future many-core systems has been implemented. In order to limit coherence protocol complexity, inclusiveness is required to track coherence information across levels in this type of systems, but this might introduce unsustainable costs for directory structures. Cost reduction decisions taken to reduce this complexity may introduce artificial inefficiencies in the on-chip cache hierarchy, especially when the number of cores and private cache size is large. The coherence protocol presented in this Chapter, denoted MOSAIC, introduces a new approach to tackle this problem. In energy terms, the protocol scales like a conventional directory coherence protocol, but relaxes the shared information inclusiveness. This allows the performance implications of directory size and associativity reduction to be overcome. MOSAIC demonstrates that inclusiveness is escapable and can be removed from a directory coherence protocol, while maintaining the complexity constrained. In fact, MOSAIC is even simpler than a conventional directory. The results of our evaluation show that the approach is quite insensitive, in terms of performance and energy expenditure, to the size and associativity of the directory.

Chapter 6. Conclusions and Future Work

In this chapter we will present the main conclusions of this thesis, the publications directly derived from it and those that it has spawned. To finish up, we will describe the main research lines for the near future.

6.1 Conclusions

The main conclusions of this work are obviously associated with the different aspects that have guided its development and the results obtained from the proposals made.

Coherence protocols. Complexity

Although throughout this whole document complexity has not been given the importance it really deserves, it is actually one of the main problems that coherence protocol design entails. From the beginning of the protocol development, through the difficult path of the verification process, up to the achievement of protocol correctness, the whole process is quite convoluted. As has been mentioned before, both the protocol implementations presented in this work were developed with the simulator GEMS, using its Specific Language for Cache Coherence protocols SLICC [19]. GEMS provided the possibility of performing the initial debugging tasks with the tools available in the simulator, i.e. Ruby tester with synthetic and random workloads. Thus, it is possible to obtain files with all the information about what is happening in the different specified components of the initial simulations (controllers, buffers, SLICC, caches, etc.) as well as the consecutive transitions that the coherence protocol controllers change to for each address. Fortunately for the designer, SLICC does not let you cheat when implementing the protocol, which reduces the possibilities of making mistakes while designing it and losing control of the whole system (although this is a double-edged sword since on some occasions this complicates the designer's work preventing the easy achievement of an approximate estimation for new ideas or fast prototyping).

In any case, the conclusion reached is that the process itself is highly time consuming and extremely difficult. For this reason, coherence protocols and solutions to coherence problems should be kept simple in order to avoid further complicating the whole development process. Many works and proposals, associated with coherence protocols and not, do not even mention this additional complexity and yet, many architectural decisions are made based on this complexity.

Protocol-network interaction

From the beginning of the work, it was clear that the relationship between the interconnection network and the coherence protocol is significant. It is well-known that the communication subsystem becomes critical when considering system performance. However, its relation to coherence protocol is less clear. For instance, the availability of hardware mechanisms to handle broadcast traffic extends the scalability of any broadcast-based coherence protocol.

Moreover, trying to optimize the interconnection network in an isolated way by applying an excessive restriction to the hardware resource assignment may degrade system performance. For this reason, it is advisable to consider the interconnection network and the memory hierarchy together when designing large-scale CMPs, while neglecting or omitting the analysis of either of them can lead to non proper solutions for the problem presented.

Trading bandwidth for latency

Latency, and not bandwidth, is the primary performance constraint in on-chip transactions. Even assuming tens of cores executing various threads each and with workloads with a high miss rate, it is possible to implement efficient interconnection networks with enough bandwidth. However, a large percentage of CMP performance is a consequence of the miss latency, for which the way communications are made plays an important role. For this reason, trading bandwidth for latency becomes profitable since critical data will reach the processor faster. This means that, as long as this trade can be made, it seems advisable to utilize any mechanism that takes advantage of all the bandwidth available in order to reduce the final miss latency. For example, using broadcast messages to favor cache-to-cache transfers, which have high impact on the full-system performance, is a way of benefiting from the bandwidth availability in a CMP, as was demonstrated with the LOCKE protocol.

Scalability

When the number of cores increases, the hardware required to handle coherence problems might become impracticable. Even with available bandwidth, broadcast-based mechanisms might flood the interconnection network with requests from all the cores and directory-based protocols will impose high costs to hold all the coherence information necessary to locate where the data is. These two problems justify the existing skepticism about the scalability of systems with shared memory and hardware coherence. However, in this thesis MOSAIC, a scalable strategy for a high number of cores, is proposed based on three observations. Firstly, it is possible to store precise information about where any data are located, but it is not essential to maintain this information for all the private blocks in the system. Secondly, it is possible to trade bandwidth for storage by using broadcasts to find the data needed when its information is not stored

anywhere. Thirdly, additional mechanisms are needed to limit the two restrictions that affect scalability: broadcasting and storing.

Simulation framework

Finally, an additional conclusion which is worth highlighting is the importance of the whole testing environment for the architectural proposals. Although this is affirmed by all researchers in Computer Architecture, in many cases the reality is quite different. Initially, a simplified analysis of any proposal can be made, but the amount and variety of parameters that affect the behavior of a CMP require its complete simulation, including the operating system, in order to reach a reasonable level of reliability in the results. In order to achieve this, it is essential to have extensive knowledge of the tools necessary to validate the proposals and a wide variety of workloads should be used in order to represent a large set of future applications for the designed systems.

6.2 Future work

The chip multiprocessor situation, especially the coherency field, has been thoroughly explored during this work, contributing some general interest innovations to the field. However, there is still a wide variety of research lines waiting to be explored and others that have been opened during and since the development of this work. In the previous chapter, a clear line was presented for improvement of the MOSAIC protocol, which takes advantage of the majority of private blocks in the application executions obtaining some encouraging results. Besides this optimization, there are other important ones to be developed which will bring significant benefits and improvements to CMPs. In particular there are three important paths to be followed immediately after the completion of this thesis: traffic filtering, hierarchical coherence protocols, and non-volatile memory. In any case, it must be taken into account that changes in this area are so fast that they might modify our research lines.

Traffic Filtering

Orienting the research to multi-CMP systems, it seems absolutely necessary to filter the traffic that goes both ways in and out of the chips. It is absolutely necessary to augment the directory with a filter to predict when a block is "within" the chip and when is not in order to reduce the on-chip and off-chip traffic. As was mentioned in the previous chapter, additional filter information for each entry in the directory could help to reduce the number of broadcasts and would add more accuracy to the directory information. Moreover, adding some filtering for the requests traveling off chip will make it possible to use a coherence protocol like MOSAIC in a multi-CMP system without having to send massive broadcasts whenever data is not found in the directory, while maintaining a hierarchical coherence.

Hierarchical Coherence

When considering architectures with hundreds or thousands of cores, it seems difficult to imagine a single logic substrate as a good way of handling the coherence protocol information. The most suitable way to organize it seems to be using a hierarchical organization that can take advantage of the applications' locality. If efficient strategies for request filtering are found, it seems natural to group processors in order to treat them logically as only one. Thus, a request is sent to the whole group when the protocol has the certainty that the group includes in its storage space a copy of the data block required. This might lead to the development of hierarchical coherence protocols where different protocols work together in order to obtain better performance. These protocols might have heterogeneous or homogeneous features. As a heterogeneous example, there could be small groups of processors whose internal coherence is maintained with a LOCKE protocol, while the whole set of groups could be managed by a MOSAIC protocol. Homogeneous is understood as for example, to have a multi-CMP where each chip has a "small" MOSAIC protocol inside, while a "larger" MOSAIC might control the coherence among all of them.

Non-volatile memory

One technological step that can significantly change the memory hierarchy as we consider it nowadays is the use of non-volatile memories. Memories with technologies such as STT-MRAM, CBRAM, etc. have a combination of characteristics that make them attractive for using them in substitution of the DRAM or even SRAM used for the LLC. Their non-volatile features, the large density integration and above all, the absence of leakage, could provide very significant energy reductions. Therefore, including them inside the chip and combining them with 3D stacking would relieve the bandwidth-wall noticeably.

However, in order to be able to start using these technologies, there are still some important issues that must be addressed. For instance, memory writes take too long and increase the request latency. Even more seriously, they have an endurance problem since the number of writing operations that can be supported before they stop working is much lower than CMOS-based memories. In all these aspects the coherence protocol can play an important role to overcome these limitations because, in the whole system, it has the most information about what is being written in the memory and when.

Appendix A. Simulation tools

Currently, simulation is one of the most important parts of any innovation in the computer architecture research field. Simulating machines with large levels of detail enable specific improvements to be achieved and failures to be found, without having to implement any prototype physically. Nevertheless, the acquisition of the necessary tools to accomplish these simulations, precise knowledge about them and their maintenance are very time-consuming tasks.

The two proposals presented in this thesis, as well as their counterparts, have been checked and proven by using a full-system simulator composed of different simulation tools from different developers. Each of them focuses on independent areas of the system, but together they make it possible to obtain accurate results about how the proposal would behave in a real and complete system. A sketch of the different modules forming the simulation environment is depicted in figure. a.

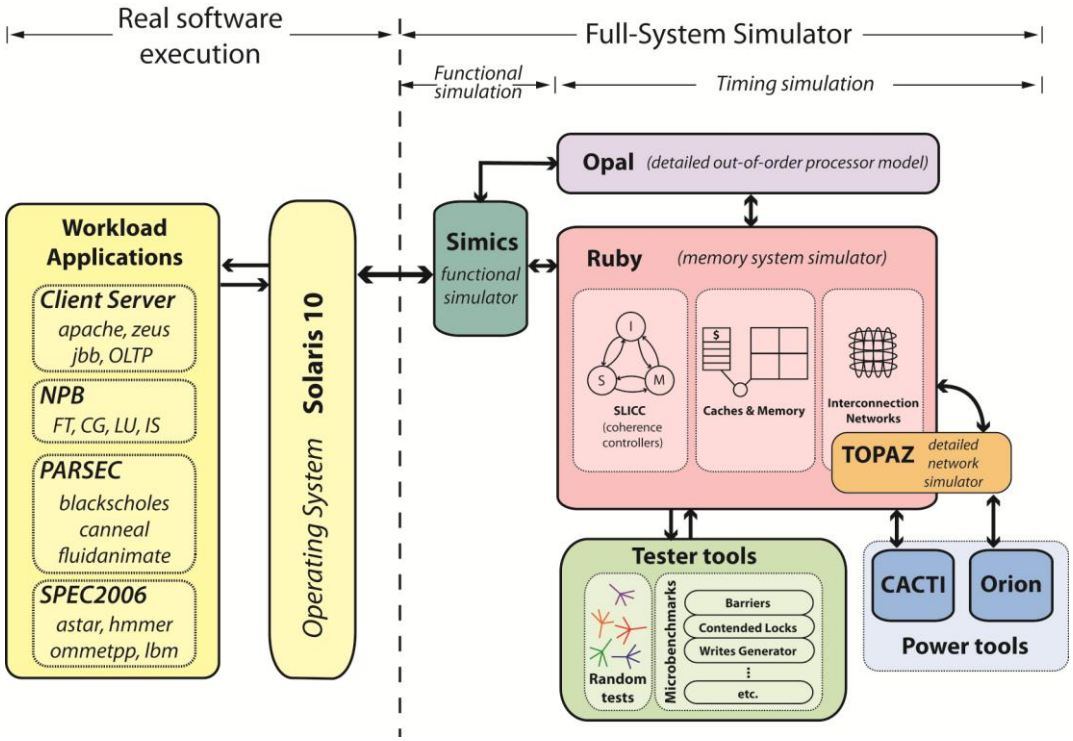

Figure. A. Complete simulation framework.

The main toolset employed for our evaluation is GEMS [19]. It is a modular simulation infrastructure that includes Ruby, Opal and additional testing tools. It decouples functionality and timing and it is able to perform full-system simulations, including the complete software stack. This tool relies on Simics [18], a functional simulator which provides enough fidelity to

boot an unmodified operating system, Solaris 10 in our case. On top of the operating system it is possible to execute realistic workloads from different benchmark suites. These benchmarks allow the simulation of diverse workload behaviors, each with specific characteristics (sharing degree, network demand, etc.) that might affect the CMP in very different ways.

Next, we will give a brief description of each of the simulators and tools of the whole framework, a brief summary of the workloads used with their main features and a short description of the workflow used.

A.1. Simics

Simics is a full system simulator [18] which enables the execution of unmodified operating systems that can to run realistic workloads, for a given hardware platform (Sunfire server in our case). Simics is in charge of maintaining the execution state and controlling the instructions that are executed at any moment. Its functional execution can be complemented with the timing details provided by the internal tools (MAI) or when used together with other time modeling tools such as GEMS [19].

A.2. GEMS

GEMS [19] (*General Execution-driven Multiprocessor Simulator*) is a simulation toolset used to evaluate multiprocessor hardware systems. Its structure is organized in different modules, it being possible to obtain different detail levels in each of the modules simulated. It has been developed by Wisconsin University under a GNU General Public License.

Basically, GEMS is the tool in charge of providing the timing of the application instructions that are executed in Simics. It can simulate the key elements that are part of the CMP architecture, allowing very accurate timing models to be defined. Its two main modules are Ruby and Opal.

A.2.1.Ruby

Ruby is in charge of modeling the system caches and memory and coherence controllers. This event-driven memory simulator combines C++ programmed objects, which simulate each of the hardware components in the memory hierarchy, with SLICC programmed components. SLICC (Specification Language for Implementing Cache Coherence) is a language included in GEMS, specifically created to specify new coherence protocols. Both coherence protocols LOCKE and MOSAIC, as well as all the other counterparts needed for comparison, have been implemented using this special-purpose language. It enables the consistent definition of the coherence

controllers' state machine, specifying the sets of states, events, actions and transitions (see chapter 2) needed to implement a coherence protocol.

Ruby Testing Tools

Ruby is connected to several simulation drivers which generate the requests that Ruby has to manage. One of these important modules (continuously employed in this work) is the r*andom tester* [134]. Executing pseudo-random memory accesses, it is used to stress the memory system and find an important number of errors appearing throughout the coherence protocol development. Another useful module is the *microbenchmarks* set which offers the possibility of analyzing the performance of any proposal for some specific conditions, i.e. barriers, contended blocks and other deterministic drivers.

A.2.2.Opal

The execution done by Simics is sequential which, especially nowadays, makes it indispensable to add a module that can provide the timing of a system with several cores each with advanced characteristics. In GEMS this task is run by Opal, which enables the simulation of a highly configurable out-of-order superscalar processor. As only the most frequently used part of the instruction set has been implemented, in this particular case SPARCv9 [135], each time Opal executes an instruction the processor status is compared with the one from Simics to ensure that it has been correctly done. Discrepancies among their values, which occur in less than 1% of the cases, are solved by choosing Simics' results [136].

A.3. TOPAZ

Ruby also includes a simple interconnection network simulator. As well as being simple, it models latency and bandwidth for the messages travelling between the memory hierarchy components. However, for detailed values when simulating contention, another simulator has to be used to add more accuracy. This simulator in our case is TOPAZ [20]. It is a general-purpose interconnection network simulator. It enables the detailed modeling of a wide variety of message routers, with different tradeoffs between accuracy, simulation speed and precision. The simulator includes several standard configurations, but it is possible to implement new components in the networks with specific routing and behavior in order to simulate new proposals (such as the *I-trees* included in LOCKE). It is object oriented and it is implemented in C++. TOPAZ can be used as a stand-alone tool with synthetic loads or with GEMS, substituting its interconnection network simulator and giving very detailed network results when needed.

A.4. Power tools: CACTI and Orion

One of the most essential aspects of any new architecture proposal is its energy consumption. For this reason, tools that enable these values to be obtained are indispensable in any architecture simulation framework. The one used in this work includes CACTI [137] and Orion [138] for this purpose.

CACTI is the tool used for modeling the dynamic power, access time, area and leakage power of caches and other memories. The versions used in this thesis are 5.0 and 6.5 for LOCKE and MOSAIC respectively.

Orion 2.0 augments the TOPAZ simulator in order to estimate the network energy consumption of the new components. This tool is a suite of dynamic and leakage power models developed for various architectural components of on-chip networks, enabling rapid power-performance tradeoffs at the architectural level.

A.5. Workloads

Table. A shows a list of the applications used to analyze the performance of both coherence protocol proposals provided in this thesis. Both types, multiprogrammed and multi-threaded applications are considered, all running on top of the Solaris 10 OS.

The server benchmarks correspond to the whole Wisconsin Commercial Workload suite [129]. The numerical applications correspond to the NAS Parallel Benchmark suite (OpenMP implementation version 3.2) [139]. Three benchmarks of the PARSEC suite were chosen [140]. The remaining type corresponds to multi-programmed workloads using part of the SPEC CPU2006 suite [141] running in rate mode (where one core is reserved to run OS services).

All benchmarks are fast-forwarded to the point of interest during which page tables, TLBs, predictors and caches are warmed up. In iteration-based applications, such as NPB, a warm checkpoint is taken in the middle of the execution, with a reduced number of iteration runs. Transactional workloads are warmed up by running hundreds of thousands of transactions. Moreover, each application is simulated multiple times with random perturbations in memory access time in order to reach 95% of confidence intervals.

The chosen workloads have been selected trying to cover diverse use scenarios, varying the sharing degree (from none in SPEC applications to a large amount in Server Workloads) and sharing contention (from none in SPEC to a large amount in scientific applications). Among the NAS applications, we chose the ones with the highest sharing contention. From the SPEC suite, we chose applications with a variable range in working set size.

Table. A. Multithreaded and multiprogrammed workloads.

Multithreaded	Wisconsin Commercial Workload	Apache	Apache web server, Spec Web like, 25000 transactions
		Zeus	SpecWeb like, 25000 transactions
		Jbb	SpecJBB 70000 Transactions
		OLTP	IBM DB2 DBMS, TPC-C like, 10000 transactions
	NAS Parallel Benchmarks	FT	Fast Fourier Transform – class W
		CG	Class A
		LU	LU Diagonalization – class A
		IS	Integer Sort – class A
	PARSEC	blackscholes	Native
		canneal	Native
		fluidanimate	Native
Multiprogrammed	SPEC 2006	astar	Native [7 threads in 8 proc; 15 threads in 16 proc]
		hmmer	Native [7 threads in 8 proc; 15 threads in 16 proc]
		ommetpp	Native [7 threads in 8 proc; 15 threads in 16 proc]
		lbm	Native [7 threads in 8 proc; 15 threads in 16 proc]

A.6. Short description of the Workflow

As a summary of how all the modules of the simulation framework introduced in this appendix are used, a brief description of all the steps that are usually taken to develop any new coherence protocol proposal (from the moment it seems viable) were:

a) Definition of the state transition table for each coherence controller of the new protocol (private levels, LLC and main memory).

b) Implementation with SLICC of all the state machines necessary to accomplish all the specifications defined in the transition tables.

c) Customize the cache values.

   - Ruby for cache sizes, associativity and the rest of the architecture parameters.
   - Opal for the core characteristics.
   - TOPAZ or Ruby for the network parameters, depending on the expected detail.
   - CACTI for access cycles, network parameters with Ruby or TOPAZ

d) Use of the testing tools (random tester and microbenchmarks) for the initial verification process. This step is the one that takes the largest amount of effort, because it includes long debugging processes to find the exact situation that causes all the errors appearing in the protocol. In most situations, their consequent protocol fixes will mean new coherence situations which will require starting the whole debugging process again from the beginning.

e) Selection of the workload sets and their execution parameters.

f) Fixing new anomalies appearing due to corner cases that the execution of real applications exposes.

g) Analysis of the proposal performance (execution time, latencies, traffic, energy, etc.).

Finally, the whole simulation process has a high complexity and the total number of parameters that have to be considered is large. Moreover, the global simulation times are quite long, as is each iteration of the design process, and they take even longer as the development advances (days, or weeks in some extreme cases, to obtain reliable values). However, the high reliability of the results obtained with such a complex framework makes it worth the enormous effort that is required to use it correctly.

Bibliography

[1] G. E. Moore, “Cramming more components onto integrated circuits,” *Electronics*, vol. 38, no. 8, 1965.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th Ed. Morgan Kaufmann Publishers, 2007.

[3] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, “Design of Ion-Implanted MOSFET’s with Very Small Physical Dimensions,” *IEEE J. Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[4] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd, “Power7: IBM’s Next-Generation Server Processor,” *IEEE Micro*, vol. 30, no. 2, pp. 7–15, Mar. 2010.

[5] M. Butler, L. Barnes, D. Das Sarma, and B. Gelinas, “Bulldozer: An Approach to Multithreaded Compute Performance,” *IEEE Comput. Soc.*, vol. 31, no. 2, pp. 6–15, 2011.

[6] T. Jain and T. Agrawal, “The Haswell Microarchitecture - 4th Generation Processor,” *Int. J. Comput. Sci. Inf. Technol.*, vol. 4, no. 3, pp. 477–480, 2013.

[7] J. Feehrer, S. Jairath, P. Loewenstein, R. Sivaramakrishnan, D. Smentek, S. Turullols, and A. Vahidsafa, “The Oracle Sparc T5 16-Core Processor Scales to Eight Sockets,” *IEEE Comput. Soc.*, vol. 33, no. 2, pp. 48–57, 2013.

[8] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, “Scaling the bandwidth wall: challenges in and avenues for CMP scaling,” in *36th International Symposium on Computer Architecture (ISCA)*, 2009, vol. 37, no. 3, pp. 371–382.

[9] ITRS, “Roadmap 2012,” 2012. [Online]. Available: http://www.itrs.net/links/2012itrs/home2012.htm.

[10] P. Prieto, V. Puente, and J. A. Gregorio, “Multilevel Cache Modeling for Chip-Multiprocessor Systems,” *IEEE Comput. Archit. Lett.*, vol. 10, no. 2, pp. 49–52, Feb. 2011.

[11] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, “Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System,” in *18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009, pp. 261–270.

[12] F. Busaba, M. A. Blake, B. Curran, M. Fee, C. Jacobi, P.-K. Mak, B. R. Prasky, and C. R. Walters, “IBM zEnterprise 196 microprocessor and cache subsystem,” *IBM J. Res. Dev.*, vol. 56, no. 1, pp. 1:1–1:12, Jan. 2012.

[13] A. W. Topol, D. C. La Tulipe, L. Shi, D. J. Frank, K. Bernstein, S. E. Steen, A. Kumar, G. U. Singco, a. M. Young, K. W. Guarini, and M. Ieong, “Three-dimensional integrated circuits,” *IBM J. Res. Dev.*, vol. 50, no. 4, pp. 491–506, Jul. 2006.

[14] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, N. P. Carter, and C.-T. Chou, “DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism,” in *20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011, pp. 155–166.

[15] J. Howard, S. Dighe, S. R. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. K. De, and R. Van Der Wijngaart, “A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and

DVFS for Performance and Power Scaling," *IEEE J. Solid-State Circuits*, vol. 46, no. 1, pp. 173–183, Jan. 2011.

[16] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Commun. ACM*, vol. 55, no. 7, p. 78, Jul. 2012.

[17] M. M. K. Martin, M. D. D. Hill, and D. A. Wood, "Token Coherence: Decoupling Performance and Correctness," in *30th International Symposium on Computer Architecture (ISCA)*, 2003, pp. 182–193.

[18] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics : A Full System Simulation Platform," *Computer (Long. Beach. Calif).*, vol. 35, no. 2, pp. 50–58, 2002.

[19] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *ACM SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, 2005.

[20] P. Abad, P. Prieto, L. G. Menezo, A. Colaso, V. Puente, and J.-Á. Gregorio, "TOPAZ: An Open-Source Interconnection Network Simulator for Chip Multiprocessors and Supercomputers," in *6th International Symposium on Networks-on-Chip*, 2012, pp. 99–106.

[21] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2011.

[22] D. J. Sorin, M. Plakal, A. E. Condon, M. D. Hill, M. M. K. Martin, and D. a. Wood, "Specifying and verifying a broadcast and a multicast snooping cache coherence protocol," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 6, pp. 556–578, Jun. 2002.

[23] J. Archibald and J. L. Baer, "Cache coherence protocols: evaluation using a multiprocessor simulation model," *ACM Trans. Comput. Syst.*, vol. 4, no. 4, pp. 273–298, Sep. 1986.

[24] D. Culler and J. P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*. Ed. Morgan Kaufmann, 1998.

[25] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," in *17th International Symposium on Computer Architecture (ISCA)*, 1990, pp. 148–159.

[26] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," in *24th International Symposium on Computer Architecture (ISCA)*, 1997, pp. 241–251.

[27] A. Gupta, W. Weber, and T. Mowry, "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," in *International Conference on Parallel Processing (ICPP)*, 1990, pp. 312–321.

[28] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: a 32-way multithreaded SPARC processor," *IEEE Micro*, pp. 21–29, 2005.

[29] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor," *IEEE Micro*, vol. 30, no. 2, pp. 16–29, 2010.

[30] J. Duato, "A theory of deadlock-free adaptive multicast routing in wormhole networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, no. 9, pp. 976–987, 1995.

[31] S. S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb, “The Alpha 21364 Network Architecture,” in *Hot Interconnects*, 2001, pp. 113–117.

[32] N. E. Jerger, L.-S. Peh, and M. H. Lipasti, “Virtual Circuit Tree Multicasting: A case for on-chip hardware multicast support,” in *35th International Symposium on Computer Architecture (ISCA)*, 2008, pp. 229–240.

[33] T. Krishna, L. Peh, B. M. Beckmann, and S. K. Reinhardt, “Towards the Ideal On-chip Fabric for 1-to-Many and Many-to-1 Communication Categories and Subject Descriptors,” in *44th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, vol. 2, pp. 71–82.

[34] J. R. Goodman, “Using Cache Memory to Reduce Processor-Memory Traffic,” in *10th International Symposium on Computer Architecture (ISCA)*, 1983, pp. 124–131.

[35] Intel Corporation, “Intel Multibus ® Specification. Order Number: 9800683-04,” 1982.

[36] S. J. Frank, “Tightly Coupled Multiprocessor System Speeds Memory-Access Times,” *Electronics*, vol. 1, 1984.

[37] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon, “Implementing a Cache Consistency Protocol,” in *12th International Symposium on Computer Architecture (ISCA)*, 1985, pp. 276–283.

[38] M. S. Papamarcos and J. H. Patel, “A low-overhead coherence solution for multiprocessors with private cache memories,” in *11th International Symposium on Computer Architecture (ISCA)*, 1984, pp. 348–354.

[39] C. P. Thacker, L. C. Stewart, and E. H. J. Satterthwaite, “Firefly : A Multiprocessor Workstation,” *IEEE Trans. Comput.*, vol. 37, no. 8, pp. 909–920, 1988.

[40] E. M. McCreight, “The Dragon Computer System,” *Springer US*, vol. 96, no. Microarchitecture of VLSI Computers, pp. 83–101, 1985.

[41] C. K. Tang, “Cache system design in the tightly coupled multiprocessor system,” in *AFIPS’76 National Computer Conference*, 1976, pp. 749–754.

[42] L. M. Censier and P. Feautrier, “A New Solution to Coherence Problems in Multicache Systems,” *IEEE Trans. Comput.*, vol. C–27, no. 12, pp. 1112–1118, Dec. 1978.

[43] J. Archibald, J. Baer, and S. Wa, “An Economical Solution to the Cache Coherence Problem,” in *11th International Symposium on Computer Architecture (ISCA)*, 1984, pp. 355–362.

[44] C. Lin and L. Snyder, “A Comparison of Programming Models for Shared Memory Multiprocessors,” in *International Conference on Parallel Processing (ICPP)*, 1990, pp. 163–170.

[45] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood, “Cooperative shared memory: software and hardware for scalable multiprocessors,” *ACM Trans. Comput. Syst.*, vol. 11, no. 4, pp. 300–318, Nov. 1993.

[46] M. Galles and E. Williams, “Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor,” in *27th Hawaii International Conference on System Sciences*, 1994, pp. 134–143.

[47] D. H. Lawrie, “Access and Alignment of Data in an Array Processor,” *IEEE Trans. Comput.*, vol. c–24, no. 12, pp. 1145–1155, 1975.

[48] N-Cube Company, “NCUBE-2 Processor Manual,” 1990.

[49] A. Charlesworth and S. Tarfire, “Extending the SMP Envelope,” *IEEE Micro*, vol. 18, no. 1, pp. 39–49, 1998.

[50] D. M. Tullsen, S. J. Eggers, and H. M. Levy, “Simultaneous Multithreading: Maximizing On-Chip Parallelism,” in *22nd International Symposium on Computer Architecture (ISCA)*, 1995, pp. 392–403.

[51] L. Hammond, B. A. Nayfeh, and K. Olukotun, “A Single-Chip Multiprocessor,” *Computer (Long. Beach. Calif).*, vol. 30, no. 9, pp. 79–85, 1997.

[52] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, “Piranha: a scalable architecture based on single-chip multiprocessing,” *27th International Symposium on Computer Architure (ISCA)*, 2000, pp. 282–293.

[53] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy, “IBM POWER4 System Microarchitecture,” *IBM J. Res. Dev.*, vol. 46, no. 1, 2002.

[54] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, “The AMD Opteron processor for multiprocessor servers,” *IEEE Micro*, vol. 23, no. 2, pp. 66–76, 2003.

[55] S. Gochman, M. Avi, A. Naveh, and E. Rotem, “Introduction to Intel Core Duo Processor Architecture,” *Intel Technology Journal*, vol. 10, no. 02, pp. 89–98, 2006.

[56] A. Mendelson, J. Mandelblat, S. Gochman, A. Shemer, R. Chabukswar, E. Niemeyer, and A. Kumar, “CMP Implementation in Systems Based on the Intel Core Duo Processor,” *Intel Technology Journal*, vol. 10, no. 02, 2006.

[57] J.-L. Baer, *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*. Cambridge University Press, 2010.

[58] M. D. Hill and M. R. Marty, “Amdahl’s Law in the Multicore Era,” *Computer (Long. Beach. Calif).*, vol. 41, no. 7, pp. 33–38, Jul. 2008.

[59] J. H. Kelm, D. R. Johnson, S. S. Lumetta, and S. J. Patel, “Cohesion: an Adaptive Hybrid Memory Model for Accelerators,” *IEEE Comput. Soc.*, pp. 42–55, 2011.

[60] A. Ros and S. Kaxiras, “Complexity-Effective Multicore Coherence Categories and Subject Descriptors,” in *21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 241–251.

[61] H. E. Mizrahi, J. Baer, E. D. Lazowska, and J. Zahorjan, “Introducing memory into the switch elements of multiprocessor interconnection networks,” *16th Int. Symp. Comput. Archit.*, vol. 17, no. 3, 1989.

[62] N. Eisley, L.S. Peh, and L. Shang, “In-Network Cache Coherence,” in *39th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006, vol. 5, no. 1, pp. 321–332.

[63] S. Kaxiras, J. R. Goodman, and W. D. St, “The GLOW Cache Coherence Protocol Extensions for Widely Shared Data,” 10th *International Conference Supercomputing*, 1996, pp. 35-43.

[64] N. D. Enright Jerger, L.S. Peh, and M. H. Lipasti, “Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence,” *41st IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2008, pp. 35–46.

[65] J. F. Cantin, M. H. Lipasti, and J. E. Smith, “Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking,” in *32nd International Symposium on Computer Architecture (ISCA)*, 2005, pp. 246–257.

[66] A. Moshovos, “RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence,” in *32nd International Symposium on Computer Architecture (ISCA)*, 2005, pp. 234–245.

[67] N. Agarwal, L.-S. Peh, and N. K. Jha, “In-Network Coherence Filtering : Snoopy Coherence without Broadcasts,” in *42nd IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 232–243.

[68] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon, “Amoeba-Cache: Adaptive Blocks for Eliminating Waste in the Memory Hierarchy,” in *45th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012, pp. 376–388.

[69] C. Dubnicki and T. J. LeBlanc, “Adjustable Block Size Coherent Caches,” in *19th International Symposium on Computer Architecture (ISCA)*, 1992, pp. 170–180.

[70] J. Zebchuk, E. Safi, and A. Moshovos, “A Framework for Coarse-Grain Optimizations in the On-Chip Memory Hierarchy,” *40th IEEE/ACM Int. Symp. Microarchitecture*, pp. 314–327, 2007.

[71] A. Raghavan, C. Blundell, and M. M. K. Martin, “Token tenure: PATCHing token counting using directory-based cache coherence,” in *41st IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2008, pp. 47–58.

[72] M. R. Marty and M. Hill, “Coherence Ordering for Ring-based Chip Multiprocessors,” in *39th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006, pp. 309–320.

[73] K. Strauss, X. Shen, and J. Torrellas, “Flexible Snooping: Adaptive Forwarding and Filtering of Snoops in Embedded-Ring Multiprocessors,” in *33rd International Symposium on Computer Architecture (ISCA)*, 2006, pp. 327–338.

[74] K. Strauss, X. Shen, and J. Torrellas, “Uncorq: Unconstrained Snoop Request Delivery in Embedded-Ring Multiprocessors,” in *40th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007, pp. 327–342.

[75] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, “Introduction to the Cell multiprocessor,” *IBM J. Res. Dev.*, vol. 49, no. 4, pp. 589–604, Jul. 2005.

[76] D. B. Gustavson, “The Scalable Coherent Interface and related standards projects,” *IEEE Micro*, vol. 12, no. 1, pp. 10–22, 1992.

[77] R. Singhal, “Inside Intel ® Next Generation Nehalem Microarchitecture,” in *HOT Chips 20*, 2008.

[78] S. Damaraju, G. Varghese, S. Jahagirdar, and T. Khondker, “A 22nm IA Multi-CPU and GPU System-on-Chip,” in *IEEE International Solid-State Circuits Conference (ISSCC)*, 2012, vol. 44, no. 4, pp. 56–57.

[79] D. Sanchez and C. Kozyrakis, “The ZCache: Decoupling Ways and Associativity,” in *43rd IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010, pp. 187–198.

[80] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, “Cuckoo directory: A scalable directory for many-core systems,” in *17th International Symposium on High Performance Computer Architecture (HPCA)*, 2011, pp. 169–180.

[81] A. Seznec, “A case for two-way skewed-associative caches,” in *20th International Symposium on Computer Architecture (ISCA)*, 1993, pp. 169–178.

[82] P. Abad, V. Puente, and J.-A. Gregorio, “MRR: Enabling fully adaptive multicast routing for CMP interconnection networks,” in *15th International Symposium on High Performance Computer Architecture (HPCA)*, 2009, pp. 355–366.

[83] P. Abad, “Design of Novel Router Architectures Reconciling Simplicity and Performance for Chip Multiprocessor Interconnection Networks,” PhD Thesis - University of Cantabria (Spain), 2010.

[84] H. Zhao, A. Shriraman, and S. Dwarkadas, “SPACE : Sharing Pattern-based Directory Coherence for Multicore Scalability,” in *19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010, pp. 135–146.

[85] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, “A tagless coherence directory,” in *42nd IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, p. 423-434.

[86] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[87] H. Zhao, A. Shriraman, S. Dwarkadas, and V. Srinivasan, “SPATL: Honey, I Shrunk the Coherence Directory,” in *20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011, pp. 33–44.

[88] M. Alisafaee, “Spatiotemporal Coherence Tracking,” in *45th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012, pp. 341–350.

[89] S.L. Guo, H.X. Wang, Y.B. Xue, C.M. Li, and D.S. Wang, “Hierarchical Cache Directory for CMP,” *J. Comput. Sci. Technol.*, vol. 25, no. 2, pp. 246–256, Mar. 2010.

[90] D. Sanchez and C. Kozyrakis, “SCD: A scalable coherence directory with flexible sharer set encoding,” in *18th International Symposium on High Performance Computer Architecture (HPCA)*, 2012, pp. 1–12.

[91] J. Sim, J. Lee, M. K. Qureshi, and H. Kim, “FLEXclusion: Balancing Cache Capacity and On-chip Bandwidth via Flexible Exclusion,” in *39th International Symposium on Computer Architecture (ISCA)*, 2012, no. June, pp. 321–332.

[92] P. J. Drongowski, “Basic Performance Measurements for AMD Athlon™ 64, AMD Opteron™ and AMD Phenom™ Processors,” *AMD withepaper*, vol. 25, pp. 1–26, 2008.

[93] VIA, “VIA C7 Processors.” [Online]. Available: http://www.via.com.tw/en/products/processors/c7/.

[94] A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely Jr., and J. Emer, “Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies,” in *43rd IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010, pp. 151–162.

[95] M. Chaudhuri, J. Gaur, N. Bashyam, S. Subramoney, and J. Nuzman, “Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches,” in *21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 293–304.

[96] J. Gaur, M. Chaudhuri, and S. Subramoney, “Bypass and insertion algorithms for exclusive last-level caches,” in *38th International Symposium on Computer Architecture (ISCA)*, 2011, pp. 81–92.

[97] C. Chi-Hung and H. Dietz, “Improving Cache Performance by Selective Cache Bypass,” in *22nd Annual Hawaii International Conference on System Sciences*, 1989, pp. 277–285.

[98] S. Gupta, H. Gao, and H. Zhou, “Adaptive Cache Bypassing for Inclusive Last Level Caches,” in *27th International Symposium on Parallel and Distributed Processing (IPDPS)*, 2013, pp. 1243–1253.

[99] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary, “JETTY : Filtering Snoops for Reduced Energy Consumption in SMP Servers,” in *7th International Symposium on High-Performance Computer Architecture (HPCA)*, 2001, pp. 85–96.

[100] P. Lotfi-Kamran, M. Ferdman, D. Crisan, and B. Falsafi, “TurboTag: lookup filtering to reduce coherence directory power,” in *16th International Symposium on Low Power Electronics and Design (ISLPED)*, 2010.

[101] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato, “Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks,” in *38th International Symposium on Computer Architecture (ISCA)*, 2011, pp. 93–104.

[102] G. Kurian, J. E. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. C. Kimerling, and A. Agarwal, “ATAC : A 1000-Core Cache-Coherent Processor with On-Chip Optical Network Categories and Subject Descriptors,” in *19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010, pp. 477–488.

[103] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *38th International Symposium on Computer Architecture (ISCA)*, 2011, p. 365.

[104] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, “An Analysis of Linux Scalability to Many Cores,” in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI10)*, 2010.

[105] A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, “The Multikernel : A New OS Architecture for Scalable Multicore Systems,” in *22nd Symposium on Operating Systems Principles*, 2009, pp. 29–43.

[106] J. H. Kelm, M. R. Johnson, S. S. Lumetta, and S. J. Patel, “WayPoint : Scaling Coherence to 1000-core Architectures,” in *19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010, pp. 99–110.

[107] A. Driskill-Smith and Y. Huai, “STTRAM--A new spin on universal memory,” 2008. [Online]. Available: http://www.future-fab.com/documents.asp?d_ID=4400.

[108] M. Kund, G. Beitel, C. U. Pinnow, T. Rohr, J. Schumann, R. Symanczyk, and G. Muller, “Conductive bridging RAM (CBRAM): An emerging non-volatile memory technology scalable to sub 20nm,” *IEEE International IEDM Technical Digest*, pp. 754–757, 2005.

[109] M. Zhang, A. R. Lebeck, and D. J. Sorin, “Fractal Coherence: Scalably Verifiable Cache Coherence,” in *43rd IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010, pp. 471–482.

[110] IBM, “IBM Power8 Processor Detailed. Hot Chips Interconnect,” 2013. [Online]. Available: http://www.hotchips.org/archives/hc25.

[111] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, “A NUCA substrate for flexible CMP cache sharing,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 8, pp. 1028–1040, 2005.

[112] K. Lee, S. Lee, and H. Yoo, “Low-power network-on-chip for high-performance SoC design,” in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2006, vol. 14, no. 2, pp. 148–160.

[113] N. Agarwal, L. Peh, and N. K. Jha, “In-Network Snoop Ordering (INSO): Snoopy Coherence on Unordered Interconnects,” in *15th International Symposium on High Performance Computer Architecture (HPCA)*, 2009, pp. 67–78.

[114] P. Abad, V. Puente, L. G. Menezo, and J. A. Gregorio, “Adaptive-Tree Multicast : Efficient Multidestination Support for CMP Communication Substrate,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 11, pp. 2010–2023, 2012.

[115] M. M. K. Martin, M. D. Hill, and D. A. Wood, “Token Coherence: a new framework for shared-memory multiprocessors,” *IEEE Micro*, vol. 23, no. 6, pp. 108–116, 2003.

[116] M. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, “Accelerating critical section execution with asymmetric multi-core architectures,” in *14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009, pp. 253–264.

[117] L. G. Menezo, “Locke Protocol Specification,” 2011. [Online]. Available: http://www.atc.unican.es/galerna/locke.

[118] C. Park, R. Badeau, L. Biro, J. Chang, T. Singh, J. Vash, B. Wang, and T. Wang, “A 1.2 TB/s on-chip ring interconnect for 45nm 8-core enterprise Xeon® processor,” in *IEEE International Solid-State Circuits Conference (ISSCC)*, 2010, pp. 180–181.

[119] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. K. Martin, and D. A. Wood, “Improving Multiple-CMP Systems Using Token Coherence,” in *11th International Symposium on High-Performance Computer Architecture (HPCA)*, 2005, pp. 328–339.

[120] Y. H. Song and T. M. Pinkston, “Efficient handling of message-dependent deadlock,” in *15th International Parallel and Distributed Processing Symposium (IPDPS)*, 2001, vol. 00, no. C.

[121] V. Zyuban and P. Kogge, “Optimization of high-performance superscalar architectures for energy efficiency,” in *International Symposium on Low Power Electronics and Design*, 2000, pp. 84–89.

[122] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, “Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor,” *IEEE Micro*, vol. 30, no. 2, pp. 16–29, 2010.

[123] L. G. Menezo, V. Puente, and J. A. Gregorio, “The Case for a Scalable Coherence Protocol for Complex On-Chip Cache Hierarchies in Many-Core Systems,” in *22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013, pp. 279–288.

[124] S. Przybylski, M. Horowitz, and J. Hennessy, “Characteristics Of Performance-Optimal Multi-level Cache Hierarchies,” in *16th International Symposium on Computer Architecture (ISCA)*, 1989, pp. 114 – 121.

[125] L. G. Menezo, “Mosaic Protocol Specification,” 2013. [Online]. Available: http://www.atc.unican.es/galerna/mosaic.

[126] A. Gupta, W. Weber, and T. Mowry, “Reducing memory and traffic requirements for scalable directory-based cache coherence schemes,” *Springer US*, pp. 167–192, 1992.

[127] L. G. Menezo, V. Puente, P. Abad, and J. A. Gregorio, “Improving coherence protocol reactiveness by trading bandwidth for latency,” in *9th ACM International Conference on Computing Frontiers (CF’12)*, 2012, pp. 143–152.

[128] D. Sanchez and C. Kozyrakis, “SCD: A scalable coherence directory with flexible sharer set encoding,” in *18th International Symposium on High Performance Computer Architecture (HPCA)*, 2012, pp. 1–12.

[129] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. D. Hill, D. A. Wood, and D. J. Sorin, “Simulating a $2M Commercial Server on a $2K PC,” *Computer (Long. Beach. Calif).*, vol. 36, no. 2, pp. 50–57, Feb. 2003.

[130] N. Kurd, J. Douglas, P. Mosalikanti, and R. Kumar, “Next generation Intel® micro-architecture (Nehalem) clocking architecture,” in *IEEE Symposium on VLSI Circuits*, 2008, pp. 62–63.

[131] M. Butler, “AMD ‘Bulldozer’ Core - a new approach to multithreaded compute,” in *HOT Chips 22*, 2010.

[132] G. H. Loh and M. D. Hill, “Efficiently enabling conventional block sizes for very large die-stacked DRAM caches,” in *44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, p. 454.

[133] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, “Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0,” in *40th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007, pp. 3–14.

List of figures

List of tables