IMT Institute for Advanced Studies, Lucca

Lucca, Italy

Wire delay effects reduction techniques and topology optimization in NUCA based CMP systems

PhD Program in Computer Science and Engineering

XXI Cycle

Francesco Panicucci

2009

The dissertation of Francesco Panicucci is approved.

Programme Coordinator: Prof. Ugo Montanari, Università di Pisa

Supervisor: Prof. Cosimo Antonio Prete, Università di Pisa

Tutor: Prof. Pierfrancesco Foglia, Università di Pisa

The dissertation of Francesco Panicucci has been reviewed by:

Prof. Stefanos Kaxiras, University of Patras, Greece

Prof. Ben Juurlink, Delft University of Technology, Netherlands

IMT Institute for Advanced Studies, Lucca

2009

Contents

List of Figures

List of Tables

Acknowledgments

The activities performed for this thesis work are supported by the HiPEAC and SARC projects.
HiPEAC (High Performance Embedded Architectures and Compilation) is an European Network of excellence (www.hipeac.net).
SARC (Scalable Architecture) is an European integrated project concerned with long term research in advanced computer architecture (www.sarc-ip.org).

Vita

March 12, 1980
Born, Cecina(LI), Italy

November 14, 2002
Bachelor of Science in Computer Science Engineering
Final marks: 101/110
Università di Pisa
Pisa, Italy

October 26, 2005
Master Degree in Computer Science Engineering
Final marks: 105/110
Università di Pisa
Pisa, Italy

Publications

1. P. Foglia, F. Panicucci, C. A. Prete, M. Solinas, An evaluation of behaviors of S-NUCA CMPs running scientific workload. In Proceedings of the 12th EUROMICRO Conference on Digital System Design (DSD09), Patras, Greece, 27-29 July 2009. to appear

2. P. Foglia, F. Panicucci, C. A. Prete, M. Solinas, Investigating Design Trade-Off in S-NUCA baseb CMP Systems. In Proceedings of the Workshop on UNIQUE CHIPS and SYSTEMS (UCAS-5), Boston, MA, April 26, 2009.

3. P. Foglia, G. Gabrielli, F. Panicucci, C. A. Prete, M. Solinas, Reducing Sensitivity to NoC Latency in NUCA Caches. 3rd Workshop on Interconnection Network Architectures: On-Chip, Multi-Chip (INA-OCMC'09), Paphos, Cyprus, January 25, 2009.

4. P. Foglia, F. Panicucci, C. A. Prete, M. Solinas, Investigating Design Trade-Off in CMP Systems. Proceedings of the Poster Session of the 4th International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES2008), L'Aquila, Italy, July 2008.

5. P. Foglia, F. Panicucci, C. A. Prete, M. Solinas, Facing the False Miss Problem in D-NUCA based CMP Systems. Proceedings of the Poster Session of the 4th International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES2008), L'Aquila, Italy, July 2008.

6. P. Foglia, F. Panicucci, C. A. Prete, M. Solinas, CMP L2 NUCA Cache Energy Consumption Model. Proceedings of the Poster Session of the 4th International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES2008), L'Aquila, Italy, July 2008.

7. P. Foglia, F. Panicucci, C. A. Prete, M. Solinas, CMP L2 NUCA Cache Power Consumption Reduction Technique. Proceedings of IEEE Symposium on Low Power and High-Speed Chips (COOLChips XI),Yokohama, Japan, pp. 163, April 16-18 2008.

8. P. Foglia, F. Panicucci, C. A. Prete, M. Solinas, Techniques for Reducing Power Consumption in CMP NUCA caches. Proceedings of the Poster Session of the 3nd International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES2007), L'Aquila, Italy, July 2007.

Abstract

One of the most important issues designing large last level cache in a CMP system is the increasing effect of wire delay problem which affects the banks access time and reduces the performances. Some CMP systems adopt a shared L2 cache to maximize cache capacity, instead other architectures use private L2 caches, replicating data to limit the delay from slow on-chip wires and minimize cache access time. Ideally, to improve performance for a wide variety of workloads, CMPs prefer both the capacity of a shared cache and the access latency of private caches. In this context, NUCA caches have been proved to be able to tolerate wire delay effects while maintaining a huge on-chip storage capacity. In this thesis we analyze the influence on system's behaviour of different coherence protocols (MESI and MOESI) and the effect of topology changes as design tradeoffs for S-NUCA based CMP system. Our results show that CMP topology has a great influence on performances, instead, in this scenario, the protocol has not. Then we propose and evaluate a novel block migration scheme to reduce access latency in a shared cache for D-NUCA based systems, in which are addressed two specific problems that can arise due to the presence of multiple traffic sources. Finally, we present a power consumption model we used to evaluate the energy behaviour of both static and dynamic NUCA systems. We observe the most important element of power consumption is always the static component, but the influence of the dynamic consumption is increasing.

Keywords: cache, NUCA, wire delay, latency, topology

Chapter 1

Introduction

Contents

1.1 Overview

Increasing performance of microprocessor systems is a major constrain in the design process. Improvements in semiconductor nanotechnology have continuously provided a crescent number of per-chip transistors [1]. In such a context, the efforts historically performed for improving performance focused on the increase of clock frequency and the amount of work performed at each clock cycle. With the increasing number of transistors available on a chip per process generation, multiprocessor systems have shifted from multi-chip systems to single-chip implementations. Specfically, chip multiprocessors (CMPs) containing 2-8 processors have recently became commercially available [37, 42, 51]. In order to improve CMP performance, these CMPs require high-bandwidth low-latency communication between processors and their associated instructions and

data. By quickly providing processors with instructions and data, on-chip caches can signficantly improve CMP performance. Small private high-level caches integrated closely with the processor cores provide each processor quick access to their most recently requested instructions and data. However, these finite-sized caches satisfy only a portion of requests, and many other requests must access larger lower-level caches. These large on-chip caches should both store a lot of data, thus minimizing off-chip miss latency's impact on performance, and quickly retrieve requested data to reduce global wire delay's effect on performance. Low-level cache management presents a key challenge, especially in the face of the conflicting requirements of reducing off-chip misses and managing slow global on-chip wires. Current CMP systems, such as the IBM Power 5 [51] and Sun Niagara [36], employ shared caches to maximize the on-chip cache capacity by storing only unique cache block copies. While shared caches usually minimize off-chip misses, they have high access latencies since many requests must cross global wires to reach distant cache banks. In contrast, private caches [37, 43] reduce average access latency by migrating and replicating blocks close to the requesting processor, but sacrifice effective on-chip capacity and incur more misses. Another important point related to cache management policies is the coherence strategy to be adopted in order to coordinate the many private caches distributed throughout the system as part of providing a consistent view of memory to the processors.

1.2 Wire delay problem and NUCA paradigm

Current trends in silicon fabrication technology cause a continuous transistor size decreasing. This provide two benefits: first, since transistors are smaller, more of them can be placed on a single die,

providing area for more complex micro architectures. Second, technology scaling reduces transistor gate length and hence transistor switching time. Thus if microprocessor cycle time are dominated by gate delay, greater quantities of faster transistors means the possibility of achieving higher clock rates contributing directly to higher performance. Furthermore, the availability of a such large number of transistors, will permit the integration on the same die of multiple elaboration cores together with large memory hierarchies. However, reducing the feature sizes, has caused on chip wires width and height to decrease, resulting in larger wires resistance due their smaller cross-sectional area. Unfortunately the wire capacitance has not decreased proportionally. As the signals propagation delay in a wire is proportional to the resistance*capacitance product, the result is that in modern and future chip there will be present slower wires that, in conjunction with the faster achievable clock rates, will limit the number of transistor reachable in a single cycle to be a small fraction of those present on a chip 1. As a consequence, the long wire delays will result in pipeline stalls to allow the signal propagations among the functional units and inside the units themselves. Also for the on chip memories the wire delay will have catastrophic effects. If we think to a large, monolithic memory, its access time will be dominated by wire delays. As a result the access time for such memories will be equal to the time needed to reach data that physically resides in the farthest part of the memory with respect to the requesting unit. The bulk of access time will involve routing to and from banks and not the bank accesses themselves. The effects of wire delay on achieving better performance for future cpu are quite clear: performance grow will not be reachable exclusively throughout the increase of the frequency. Thus, to obtain performance growth, there is the need of increasing the IPC thorough new architectural designs both for cpu than for memory architectures.

New trends for cpus cover both designs in which the large number of available transistors is used to implement homogeneous and heterogeneous multicore systems (in order to exploit parallels execution) than designs based on clustered architectures (in order to localize communications to a limited set of functional units thus reducing the effects of wires delay). The importance of memory hierarchies, and in particular of cache memories, for system overall performances is obvious. However, in the past and in the present, almost all research works have been focused on the improvement of the hit rate of the cache memory, while the proposals for designs which can mitigate the effect of wire delays are still rare and, at our knowledge, all related to NUCA architectures. The basic idea of Non Uniform Cache Architectures is to substitute a large and monolithic cache with an array of banks, each of which physically resides at a different distance from the controller. While the access latency of the monolithic cache is dominated by the slowest of its sub-banks, in a NUCA there is a different access latency for each bank so that the banks closest to the controllers can be accessed faster. By moving the most accessed data in the banks that are closer to the controller it is possible to obtain an overall latency time that is considerably lesser than the one offered by the monolithic cache.

1.3 Coherence protocols in CMP systems

When moving from single processor to multi processors systems, applications have to be parallelized in order to achieve performance gain. As the most part of common parallel applications use the Shared Memory paradigm, processes running on different processors share the same address space. As cache memories are needed for reducing the average latency of memory accesses, each processor has to store a copy of the most accessed memory blocks in its pri-

vate cache; when such blocks are stored in more than one cache, it is important to provide a consistent view of the shared memory. The coherence protocol is a central point of design for multicore systems, since it is responsible for guarantee correctness of memory accesses: in particular, it must assure that every access to a memory block receives the most up-to-date value of the referred location. As a consequence, the coherence protocol is also a performance-sensitive characteristic of multicore systems since it introduces an overhead on the overall communication, and this overhead directly impacts on the system behaviors: in fact, it describes how the communication between processors and shared memory has to be managed, but also how block transfer among processors, memory and caches is performed. Different types of coherence strategies have been proposed in literature. They can be divided in two different classes: write update and write invalidate. An update protocol, when a write operation on a block is performed, propagates it to all the other copies present in remote caches, while an invalidate protocol, in case of write, invalidates all the remote copies of the modified block. Examples of update protocols are Dragon [41], Firefly [54], RST [48] and PSCR [27]; examples of invalidate protocols are Berkeley [33], Synapse [24], MESI (Illinois) [47] and MOESI [53]. Cache coherence schemes are tightly coupled to the type of the interconnection infrastructure and its properties. Most of the commercially successful multiprocessors used buses to interconnect the uniprocessors and memory: a bus-based coherence protocol relays on the broadcast nature of the communication, so cache controllers have to snoop on the bus, and take the appropriate coherence action when needed. However, with the increasing numbers of cores, a bus suffers scalability limits, as it doesn't have the bandwidth to support a large number of processors. Prior solutions for more scalable multiprocessors implement packet-switched interconnects topologies in

classical Distributed Shared Memory systems [38, 39]: in such systems, the communication infrastructure has not an implicit broadcast communication paradigm. For this reason, it is necessary to adopt a directory-based scheme, in which nodes are able to access to a home node, typically called directory, that holds the information of which of the nodes has a copy of any memory block, together with some state information, and takes the appropriate coherence actions. In such scenario, the communication paradigm is based on message passing among the nodes in the system. As future CMP systems are expected to put hundreds of cores on a single chip, a bus-based solution would be undesirable, while more scalable interconnection infrastructures have to be adopted: for example, a Network-on-chip (NoC) [16, 15]. As a result, the coherence protocol in such large-scale CMP systems is a directory- based protocol, similar to those proposed in the past for classical DSM systems.

1.4 Thesis structure

This thesis is organized in seven chapters. The first is the introduction to our work and the the second chapter presents the related works. The third section shows our implementaion of MESI and MOESI protocol and in the fourth chapter we discuss the design tradeoff in S-NUCA based CMP systems. The subsequent section describes the the FMA protocol implemented in a D-NUCA system and the results we obtained exploring this architecture. At last, the seventh chapter discusses the conclusion and presents the future works.

Chapter 2

Related Works

Contents

In this section we present an overview of the state of the art about CMP systems, NUCA architecture and cache coherence.

2.1 CMP systems

2.1.1 Stanford Hydra CMP

Hydra [29] is a CMP architecture that has been designed at Stanford University,USA. This architecture is built using four MIPS-based cores as its individual processors. Each core has its own pair of primary instruction and data caches, while all processors share a single, large on-chip secondary cache. The processors support normal loads and stores plus the MIPS load locked (LL) and store conditional (SC) instructions for implementing synchronization primitives. Figure 2.1 shows the logical architecture of Hydra CMP. Connecting the processors and the secondary cache together are the read and write buses, along with a small number of address and control buses. In the chip implementation, almost all buses are virtual buses. While they logically act like buses, the physical wires are divided into multiple segment using repeaters and pipeline buffers, where necessary, to avoid slowing down the core clock frequencies. The read bus acts as a general-purpose system bus for moving data between the processors, secondary cache, and external interface to off-chip memory. It is wide enough to handle an entire cache line on clock cycle. This is an advantage possible with an on-chip bus that all but the most expensive multichip systems cannot match due to the large number of pins that would be required on all chip packages. The narrower write bus is devoted to writing all writes made by the four codes directly to the secondary cache. This allows the permanent machine state to be maintained in the secondary cache. The bus is pipelined to allow single-cycle occupancy by each write, preventing it from becoming a system bottleneck. The write bus also permits Hydra to use a simple, invalidation only coherence protocol to maintain coherent primary caches. Writes broadcast over the bus invalidate

copies of the same line in primary caches of the other processors. No data is ever permanently lost due to these invalidations because the permanent machine state is always maintained in the secondary cache. The write bus also enforces memory consistency in Hydra. Since all writes must pass over the bus to become visible to the other processors, the order in which they pass is globally acknowledged to be the order in which they update shared memory. It

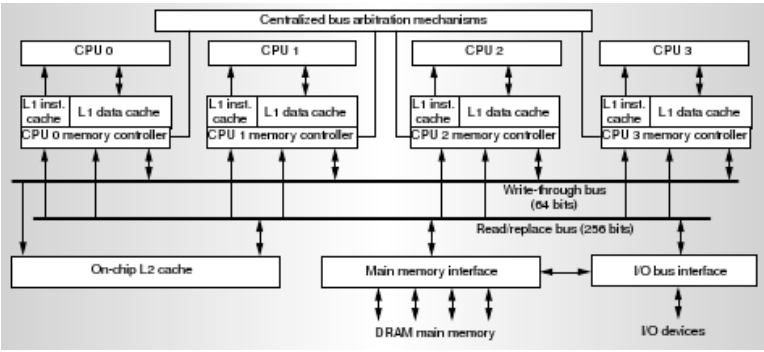

Figure 2.1: *An overview of the Hydra CMP*

is important to concern with minimizing two measurements of the design: the complexity of high-speed logic and the latency of interprocessor communication. Since decreasing one tends to increase the other, a CMP design must strive to find a reasonable balance. Any architecture that allows interprocessor communication between registers or the primary caches of different processors will add complex logic and long wires path that are critical to the cycle time of the individual processor cores; this complexity results in excellent interprocessor communication latency. Because it is now possible to integrate reasonable-size secondary caches on processor dies and

since these caches are typically not tightly connected to the core logic, it is possible to use that as the point of communication. In the Hydra architecture, this results in interprocessor communication latencies of 10 to 20 cycles, which are fast enough to minimize the performance impact from communication delays. After considering the bandwidth required by four single- issue MIPS processors sharing a secondary cache, a simple bus architecture would be sufficient to handle the bandwidth requirements for a four. This is acceptable for a four- to eight-processor Hydra implementation. However, designs with more cores or faster individual processors may need to use either more buses, crossbar interconnections, or a hierarchy of connections.

2.1.2 Piranha CMP

Piranha [9] is a research prototype developed at Compaq to explore chip multiprocessing architectures targeted at parallel commercial workloads. The centerpiece of the Piranha architecture is a highly-integrated processing node with eight simple Alpha processor cores, separate instruction and data caches for each core, a shared second-level cache, eight memory controllers, two coherence protocol engines, and a network router all on a single die. Multiple such processing nodes can be used to build a glueless multiprocessor in a modular and scalable fashion. In addition to exploring chip multiprocessing, the Piranha architecture presents some characteristics. First, the design of the shared second-level cache uses a sophisticated protocol that does not enforce inclusion in first level instruction and data cache in order to maximize the utilization of on-chip caches. Second, the cache coherence protocol among nodes incorporates a number of unique features that result in a fewer protocol messages and a lower protocol engine occupancies compared

to previous protocol design. Finally, Piranha has a unique I/O architecture, with an I/O node that is a full-fledged member of the interconnect and the global shared-memory coherence protocol. Figure 2.2 shows the block diagram of a single Piranha processing chip. Each Alpha CPU core (CPU) is directly connected to dedicated instruction (iL1) and data (dL1) modules. These first-level caches interface to other modules through the Intra-Chip Switch (ICS). On the other side of the ICS is a logically shared second level cache (L2) that is interleaved into eight separate modules, each with its own controller, on-chip tag, and data storage. Attached to each L2 module is a memory controller (MC) which directly interfaces to one bank of up to 32 direct Rambus DRAM chips. Also connected to the ICS are two protocol engines, the Home Engine (HE) and the Remote Engine (RE), which support shared memory across multiple Piranha chips. The interconnect subsystem that

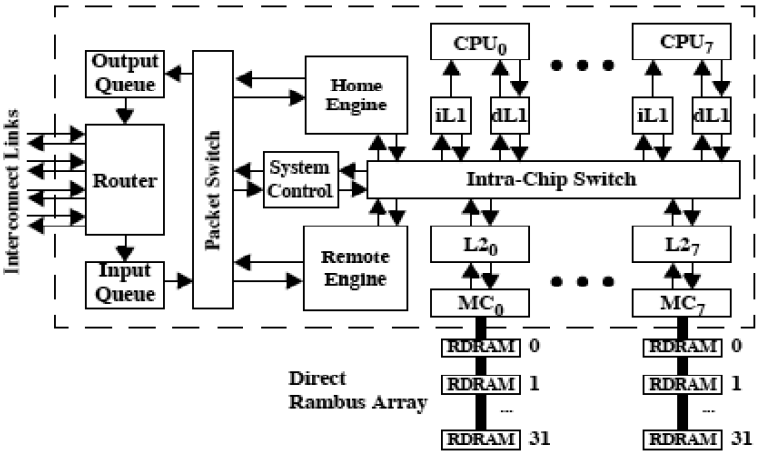

Figure 2.2: *Block diagram of a single-chip Piranha processing node*

links multiple Piranha chips consists of a Router (RT), an Input Queue (IQ), an Output Queue (OQ) and a Packet Switch (PS). The total interconnect bandwidth (in/out) for each Piranha processing chip is 32 GB/sec. Finally, the System Control (SC) modules takes care of miscellaneous maintenance-related functions, (e.g., system configuration, initialization, interrupt distribution, exception handling, performance monitoring). It should be noted that the various modules communicate exclusively through the connections shown in Figure 2.2, which also represent the actual signal connection. This modular approach leads to a strict hierarchical decomposition of the Piranha chip which allows for the development of each module in relative isolation along with well defined transactional interfaces and clock domains. While Piranha processing chip is a complete multiprocessor system on a chip, it does not have any I/O capability. The actual I/O is performed by the Piranha I/O chip which is relatively small in area compared to the processing chip. Each I/O chip is a stripped-down version of the Piranha processing chip with only one CPU and one L2/MC module. The router on the I/O chip

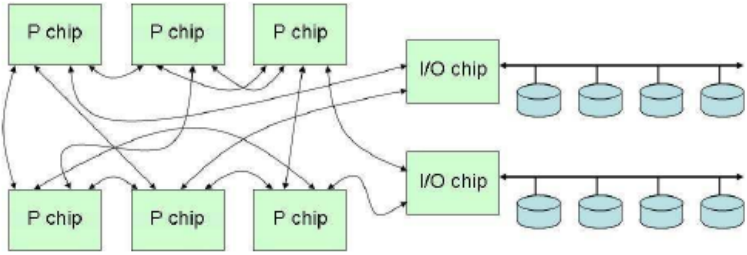

Figure 2.3: *Piranha system with six processing (8 CPUs each) and two I/O chip*

fully participates in the global cache coherence scheme. The presence of a processor core on the I/O chip provides several benefits: it enables optimization such as scheduling device drivers on this processor for lower latency access to I/O, or it can be used to virtualize the interface to various I/O devices (e.g., by having the Alpha core interpret accesses to virtual control registers). Figure 2.3 shows an example configuration of a Piranha system with both processing and I/O chips. The Piranha design allows for glueless scaling up to 1024 nodes, with an arbitrary ratio of I/O to processing nodes (which can be adjusted for a particular workload). Furthermore, the Piranha router supports arbitrary network topologies and allows for dynamic reconfigurability. One of the underlying design decisions in Piranha is to treat I/O in an uniform manner as a full-fledged member of the interconnect. In part, this decision is based on the observation that available inter-chip bandwidth is best invested in a single switching fabric that forms a global resource which can be dynamically utilized for both memory and I/O traffic.

2.1.3 Intel Core Duo

Intel Core Duo [28, 44] is based on Pentium M processor 755/745 core microarchitecture with few performance improvements at the level of each single core. The major performance boost is achieved from the integration of dual cores on the die (CMP architecture). As Figure 2.4 shows, Intel Core Duo technology is based on two enhanced Pentium M cores that were integrated and use a shared L2 cache. The way we integrated the dual core in the system had a major impact on our design and implementation process. In order to meet the performance and power targets we aimed to do the following:

- Keep the performance similar to or better than that of single

thread performance processors in the previous generation of the Pentium M family (that use the same-size L2 cache);

- Significantly improve the performance for multithreaded and multi-processes software environments;
- Keep the average power consumption of the dual core the same as previous generations of mobile processors (that use a single core);
- Ensure that this processor fits in all the different thermal envelopes the processor is targeted to.

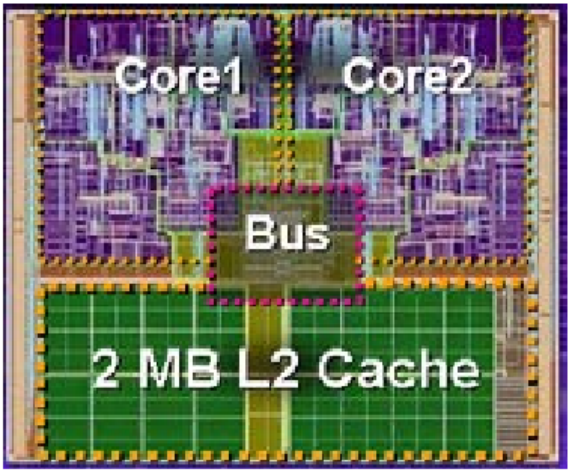

Figure 2.4: *Intel Core Duo processor floor plan*

CMP general structure. Intel Core Duo processor-based technology implements shared cache-based CMP microarchitecture in

order to maximize the performance of both ST and MT applications (assuming the same L2 cache size). The main characteristics of the Core Duo can be summarized as following:

- Each core is assumed to have an independent APIC unit to be presented to the OS as a separate logical processor;
- From an external point of view the system behaves like a Dual Processor (DP) system;
- From the software point of view, it is fully compatible with Intel Pentium 4 processors;
- Each core has an independent thermal control unit;
- The system combines per-core power state together with package-level power state.

The coherence protocol. From the external observer, the behavior of a CMP system should be looked at as the behavior of a dual package (DP) system. For that purpose, Intel Core Duo processor implements the same MESI protocol as in all other Pentium M processors [44]. In order to improve performance, the protocol is optimized for faster communication between the cores, particularly when the data exist in the L2 cache. A noticeable example of such a modification was done in order to allow the system to distinguish between a situation in which data are shared by the two CMP cores, but not with the rest of the world, and a situation in which the data are shared by one or more caches on the die as well as by an agent on the external bus (can be another processor). When a core issues an RFO, if the line is shared only by the other cache within the CMP die, we can resolve the RFO internally very fast, without going to the external bus at all. Only if the line is shared with another agent

on the external bus do we need to issue the RFO externally. For most Intel Core Duo systems, when only one package exists, this is a very important optimization. In the case of a multi-package system, the number of coherence messages over the external bus is smaller than in similar DP or MP systems, since much of the communication is being resolved internally. The number of required coherency messages is also much smaller than in the case of using a split cache which requires all the communication between the cores and split L2 caches to be done over the external bus.

2.2 NUCA cache architecture

This section describes the state of the art in Non-Uniform-Cache-Architectures (NUCA) cache designs. It is a recent design proposal to reduce the effects of the wire delays that will dominate the latencies of future large sized memory hierarchies. The first idea of NUCA is to replace a large and monolithic cache with an array of banks; each bank is physically placed at different distances from the cache controller. While the response latency of a monolithic cache is dominated by the slowest of its subbanks, a NUCA presents different latencies for different banks, so that banks closest to the controllers can be accessed with a smaller latency.

2.2.1 Single core NUCA architecture

The first NUCA architeture was proposed by Keckler, Burger and Kim [34] for a single core system. They based their considerations on the fact that huge, monolithic caches are strongly wire-delay dominated, due to the need of reaching the most far line of the cache at each access. By avoiding this need, they proposed various sub-banked organization for the cache, and the common charac-

teristic was that the access time changes with the distance from the cache controller. Figure 2.5 shows the proposed NUCA organizations, compared to classical monolithic solution, named UCA (Uniform Cache Access); the numbers over each cache or NUCA banks represent the access latency in terms of clock cycles, for the configuration and nanotechnology considered in the papers. The

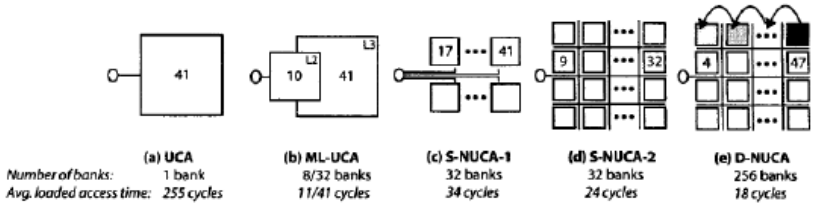

Figure 2.5: *The NUCA cache architectures proposed by Kecler, Burger and Kim compared to classical memory systems*

first and the second represent two classical caches at uniform access time; the second is multilevel. Instead, the third and the fourth scheme represent two different types of Static-NUCA (S-NUCA); in particular, in the configuration shown in figure 2.5c present an aggressive subbanked organization in which each bank uses a private, two-way, pipelined transmission channel and the mapping of data into banks is predetermined based on the memory address and the bank index (S-NUCA-1), while the configuration shown in figure 2.5d the private channels are substituted by a two-dimensional switched network allowing a consistent space saving and a further aggressive bank partitioning (S-NUCA-2). In the last scheme is shown the idea of Dynamic-NUCA (D-NUCA), in which the subbanked organization is similar to S-NUCA-2, and memory blocks are able to dynamically migrate toward the banks that exhibit lower la-

tencies with respect to the cache controller. D-NUCA exploits the

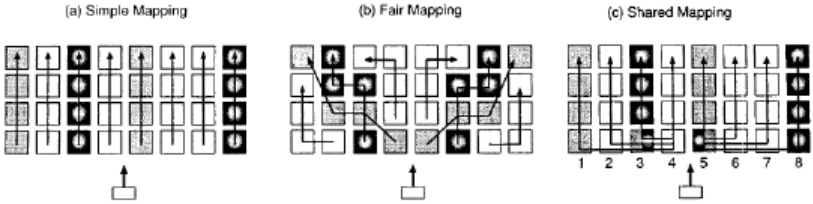

Figure 2.6: *The three mapping solutions proposed for the D-NUCA*

banks access latencies non uniformity by placing frequently accessed data in closer (and faster) banks and less important data in farther banks. Some solutions are identified and proposed for the three main topics in NUCA designs:

- Mapping: how maps data to the banks? In which bank a datum can reside?
- Search: how the possible locations for a datum are searches to find a line?
- Movement and replacement: how and when the data should migrate from a bank to another? Where a new datum should be placed?

For the mapping problem the proposal is the use of the multi-banked D-Nuca cache as a set-associative structure, in which each set is spread over multiple banks and each bank hold one way of the set. Three methods are proposed for the allocation between banks and sets: the simple mapping (Figure 2.6a), the fair mapping (Figure 2.6b) and the shared mapping (Figure 2.5c). In the simple mapping each column in the cache is a set while each row is a

way. A search is performed by first selecting the column and then searching among the banks of the chosen column for the datum. This solution is characterized by low architectural complexity but the access latencies to the various sets are not uniformly distributed. Because of the fact that the latencies are wire delay dominated, the banks belonging to the external sets will be always affected by higher latencies than those belonging to the central ones. In the fair mapping, this is solved at the cost of additional complexity: the banks are allocated to the various sets so that the average access times of each set are equalized. In the shared mapping the sets share the bank closest to the controller so that a fast bank-access is provided to all the sets. For the search problem two solutions are proposed: the incremental search and the multicast search. In the incremental search the banks of a set are searched in order starting from the closest one until the request line is found or a miss occurs in the farthest bank meaning a global cache miss. In the multicast search the request for a line is sent in parallel to all the banks of a set. This offer higher performances at the cost of increased energy consumption and networks contentions. To reduce the miss resolution time a partial tag comparison is also proposed for both the solutions: a smart search array is located in the cache controller and stores some bits of each tag. The array can be searched in parallel to the banks so that, if no matches occur, the miss processing is start early. In order to maximize the number of hits in the closest banks a generational promotion movement policy is proposed: when a hit occurs to a cache line it is swapped with the line in the bank that is next closest to the cache controller. Thus, the heavily used lines will migrate toward close and fast banks while the infrequently used lines will be demoted into farther, slower banks. The new line insertion policies evaluated are: tail insertion in which the new line is inserted in the farthest bank, head insertion in which the new line is inserted

in the closest bank and middle insertion in which the new line in inserted in the middle banks. The replacement policies evaluated are: zero-copy in which the victim line of the insertion is evicted from the cache and the one-copy in which the victim is moved to a lower-priority bank replacing a less important line farther from the controller.

2.2.2 NuRapid

Chisti, Powell and Vijaykumar in [13], describe some limitations of the NUCA cache architecture. The problems they describe are classified in four topics:

1. Tag Search. In NUCA caches, tags and data of a bank are always accessed in parallel; some times all the banks belonging to the same set are also accessed in parallel. Both those behaviours result in considerably high energy requirements that could be avoided throughout the sequential tag-data access used in many large L2 and L3 caches. Furthermore NUCA's tag array is distributed in throughout the cache along with the data array. As a consequence, searching for the matching block requires traversing a switched network which consumes energy and internal bandwidth.

2. Placement. NUCA (as in conventional caches) couples data placement with tag placement: the position in the tag array implies the position in the data array. This means that each set has a statically assigned group of banks in cache and that only a small number of ways (typically 1 or 2) of each set can be placed in the fastest banks. To mitigate this limitation, NUCA promotes the frequently accessed lines from slower to faster banks but these promotions are energy-hungry and also

consume internal bandwidth. Furthermore there are cases in which some sets are heavily accessed resulting in a big number of lines switching while at the same time some other sets are slightly used resulting in unused spaces in the fast banks that could be conveniently used to accommodate lines from the heavily used sets.

3. Data Array Layout. Usually the bits of am individual cache block are spread over many subarrays for area efficiency and error tolerance. To obtain the same latency for all the bits of a block, the NUCA design constrains them to be spread only over a few small subarrays compromising both the area efficiency than the error tolerance.

4. Bandwidth. NUCA uses an high bandwidth switched network to support parallel tag searches and line swaps; however these mechanism introduce an artificial bandwidth need while the real demand from the CPU is filtered by L1 caches and is usually low and does not justify the complexity of the switched network.

As a solution to the outlined problems, a "Non-Uniform access with Replacement And Placement usIng Distance associativity" cache architecture is proposed (briefly NuRAPID in the following). In NuRAPID the tags are decoupled from the data and they are contained in a centralized array which is located near the controller. The tag array is accessed before the data array (using a sequential tag-data access) and, upon an hit in the tag array, a pointer is used to identify which block contains the data. As shown in Figure 2.7, the tags array is accessed using conventional set associativity while "distance associativity" is introduced to manage the data array. The data array is divided in some "distance groups" (named d-group in

the figure), each characterized by a constant latency. In contrast to NUCA in which each block is statically assigned to a set, each d-group can accommodate lines coming from any set. This is obtained by the use of the tag-to-data pointer and by the use of large sized d-group (each group can be up to 2-Mbyte). It turns out that, if the running application require it, all the ways of a single set could be accommodated in the faster d-group. So, while the sequential tag-data access solves problem 1, the use of distance associativity allow to solve the placement related limitations. NuRAPID ben-

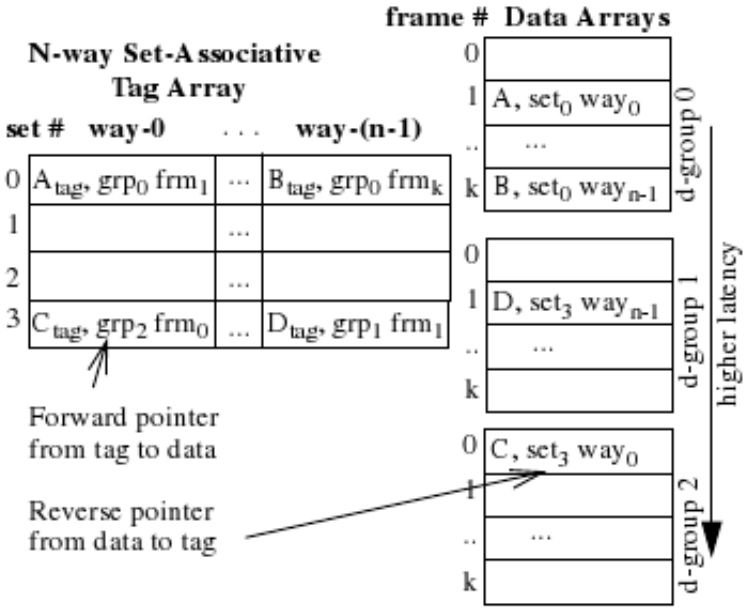

Figure 2.7: *The NuRAPID cache architecture*

efits from distance-associativity also for replacement and for pro-

motion/demotion policies. Upon a L2 cache miss, the new line is always inserted in fastest d-group. To achieve this, the victim tag is chosen in the array using an LRU policy, such tag will generically point to a data line, contained in the kth d-group, that will be liberated throughout write back to the main memory. After this a line is randomly demoted into the d-group k from the d-group k-1, freeing a slot in such bank. This is repeated for the subsequent d-groups until a slot is liberated in the d-group 0 to accommodate the new line taken from the memory. Two promotion policies are also proposed: in the next-fastest when a line in any d-group other than the fastest is accessed it is promoted to the next d-group, demoting, if needed, a randomly chosen line; in the fastest policy the line is promoted to the fastest d-group, gradually demoting a randomly chosen line a described for the data replacement. It is worth noting that in NuRAPID the lines to be evicted from the cache or demoted to slower d-groups must not necessarily belong to the same set of the newly inserted/promoted one. Obviously each demotion requires the updating of the pointer in the tag array; as the lines to be demoted are randomly chosen, a further back-pointer (from data array to tag array) is needed to identify the tag to be updated. NuRAPID architecture requirements of bandwidth are quite smaller than the ones of NUCA and, as a result, NuRAPID uses single port caches (where NUCA use multiported ones) and the architecture is not banked being sufficient the execution of a single operation at time. The use of few and large d-groups, as opposite to the big number of banks used in NUCA, allows the spreading of the bits belonging to the same data block widely over the cache area fixing the Data Array Layout problem.

2.2.3 Triangular D-NUCA

An optimization to the original NUCA design is proposed in [18] based on the key observation that the hits in a NUCA are not uniformly distributed over the banks of the cache. As a consequence of the data migration mechanism the most frequently accessed data are near the controller. Starting from such distribution, a triangular shaped NUCA cache (briefly named TD-NUCA in the following) is proposed. TD-NUCA aims to reduce the cache area and consequently the cache energy consumption with low performances degradation. TD-NUCA are proposed in two different organization: increasing TD-NUCA in which the number of banks for each way increases when moving far from the controller and decreasing TD-NUCA in which the number of banks decreases when moving far from the controller. Figure 2.8 show the two different mapping

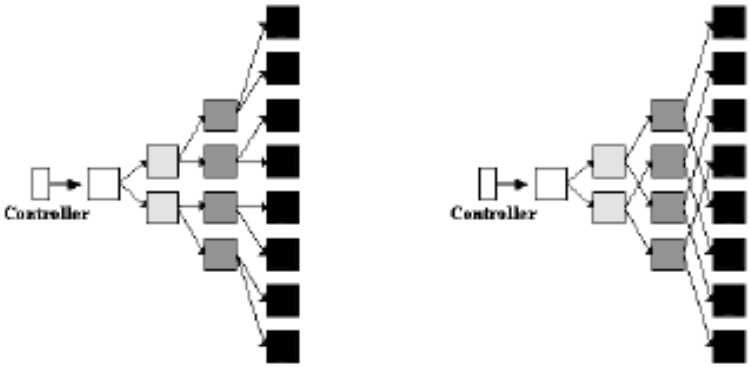

Figure 2.8: *The simple mapping (left) policy and the fair (right) mapping policy for an increasing TD-NUCA cache*

policies that are considered for the TD-NUCA; similarly to the orig-

inal D-NUCA the simple mapping is affected by latencies distribution unfairness among different sets while fair mapping correct this. The considered search policies are the incremental search in which each bank, starting from the controller, is sequentially accessed only when the previous one has reported a miss and the multicast search in which the cache request is propagated to all the banks.

2.2.4 Flexible Cache Sharing in CMP systems

A first CMP system adopting a NUCA cache as the shared last-level-cache was proposed in [32]. They proposed a CMP architecture in which 16 processors are placed on two opposite sides of a shared L2 NUCA cache. Such cache is organized as a 256 bank matrix, and a centralized directory is placed in he middle of the banks for managing the coherence of private L1 copies. Figure 2.9 shows the considered CMP architecture. This paper proposed an

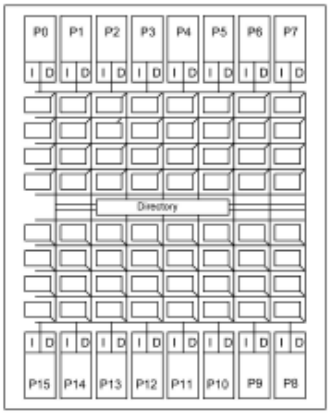

Figure 2.9: *The flexible CMP cache architecture*

evaluation of different sharing-degree, that make the shared cache being completely shared (sharing degree = 16), completely private of each CPU (sharing degree = 1), or partially shared. This aspect are not relevant for the scope of this dissertation. Differences between static and dynamic mapping and use of data migration are topics that have been previously discussed [34, 35]. However in this paper the attention goes to how CMPs pose new challenges to such topics. While in single core NUCA the migration moves data in a single direction (and in particular towards the core), in CMPs the migration can happen in multiple directions (as cores occupy different places in the chip) and this can cause conflicts with, as an extreme consequence, shared blocks ping-ponging between two processors; technically, this phenomenon is called conflict hit. Results highlight that there are some parallel applications that don't take advantage from the adoption of a dynamic migration mechanism, due to the conflict hit problem. A first attempt of facing such problem has been proposed in [6], in which a flag-based strategy is proposed for limiting the phenomenon. Particularly, a flag is used to mark just demoted blocks; blocks marked by this flag will not be promoted upon a hit, instead their flag will be reset, and the block will migrate upon the next hit. In this way, cache blocks are less prone to conflict hits.

2.2.5 NuRapid for CMP

In [14] the NuRAPID architecture is extended to the CMPs case (naming it CMP-NuRAPID). A trade-off between private and shared L2 caches is proposed with a private-tag and a shared data architecture (Figure 2.10). Each core (P0, P1, P2, and P3) has a private tag array while the data array is shared among all the cores throughout a crossbar or a network. Like private caches, tag arrays snoop

on a bus to maintain coherence and cores use it to access external memory. Like NuRAPID, CMP-NuRAPID uses sequential tag-data access, uses forward and reverse pointers, divides the data array into several distance groups (d-group) and employs distance associativity. In CMP-NuRAPID the distance of a d-group from each core is different, so each core has a different access latency for each d-group and, to exploit non-uniform access, each core will rank the d-groups in terms of preference to place its own lines of data. To boost the

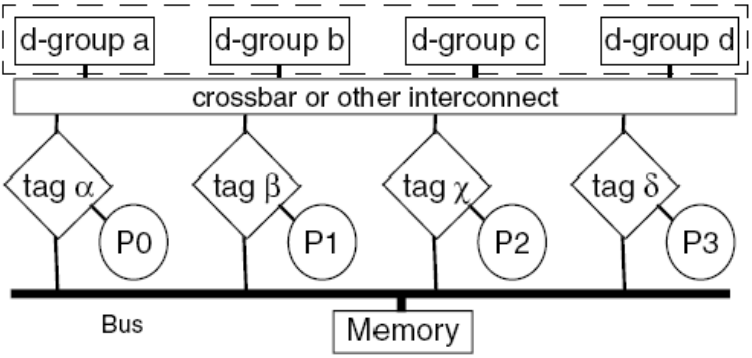

Figure 2.10: *The CMP NuRAPID architecture*

performance of CMP-NuRAPID three novel ideas are proposed:

- Controlled replication for read-only shared data
- In-situ communication for inter processor communications induced by read-write sharing
- Capacity stealing to optimize the use of cache space

Controlled replication uses private tag arrays and shared data array to achieve fast access to shared data by keeping separate copies

of the shared line close to each processor without wasting precious on-chip capacity with uncontrolled full replication. When a reader first misses on a line which is already present in the shared data array, the reader obtains the data from the already-existing on-chip copy: the reader makes a tag copy but not a data copy. Statistical measurements have shown that a cache line either is not reused or is reused two or more times. Therefore, on the second use, a data copy is made in the reader's closest d-group to avoid slow accesses for future reuses. The coherence among replicated tag is guaranteed by invalidation messages that are sent on the snoopy bus when a core decides to replace a data block which is shared and thus pointed by two or more tags. In-situ communication utilizes the hybrid structure of the cache to provide fast access to read-write shared data without incurring in coherence misses and in updates traffic overheads. For a read-write shared line only one copy is forced to be present in cache. The writer and the readers have their private tag copies which point to the same single data copy. As statistical measurements have show that each write is read more than once by each reader, the data copy is placed close to one of the readers. To support in situ communication, a coherence protocol has been developed starting from MESI with a "communication state" which substitute the shared state for those blocks containing read-write shared data. Capacity stealing uses the shared data array of the CMP-NuRAPID to guarantee L2 space to the cores proportionally to their capacity demands. Unlike a private cache in which bringing a new block to the cache means evicting another block even if space is left unused in an other core cache, in CMP-NuRAPID the cores with more capacity demand can denote their less-frequently-used data to unused frames in the d-groups closer to the cores with less capacity demands. Statistical measurements have shown that the capacity stealing is less important for multithreaded workload,

because core usually have uniform capacity demands, but it is especially beneficial for multi-programmed workloads which usually have non-uniform capacity demands. As for the other proposed designs, the placement, replacement and promotion policies are discussed. Placement and promotion policies evaluated are very similar to the one evaluated for NuRAPID: all the private lines are initially placed in the data d-group closest to the initiating core, on a hit the next-fastest or the fastest promotion policy is used; shared blocks don't move around in the cache to avoid updating to many reverse pointers. Data replacement choose the tag to be replaced applying LRU policy and starting from invalid tag and then going to private and shared. If the data line pointed from the evicted pointer is private then it is evicted from the cache and then some distance replacement could be needed to clear space in the closest d-group. If data line pointed from the evicted pointer is shared, then it is not evicted from the cache and it is left for the others sharers. One or more distance replacement are need to clear space for the new line.

2.2.6 The "Tetris" CMP architecture

Beckmann and Wood in [10] propose an 8 CPUs CMP system, in which the L2 cache is shared among all cores and organized with the NUCA paradigm. In this paper the authors introduce and evaluate some techniques to a NUCA-based CMP system, projected in a 45 nm technology:

- The use of hardware-directed stride-based prefetching (both L1 and L2 prefetching are evaluated), that utilize the prediction of repeating memory access patterns to tolerate cache miss latency;

- The migration of the frequently accessed blocks to cache banks

closer to the requesting processor, with the purpose of reducing global wire delay from L2 hit latency by moving frequently accessed cache blocks closer to the requesting processor;

- The use of on-chip transmission lines to provide fast access to all cache banks.

Figure 2.11 shows the baseline design is based on a 16MB L2 CACHE with the 8 cores, each with private L1 data and instruction caches, plugged to the four sides of the shared cache. Similarly to the original proposal, this CMP system statically partitions the address space across cache banks, which are connected via a 2D mesh interconnection networks. The 16 MB L2 storage array is partitioned into 256 banks. The width of links connecting switches, banks, and other entities is of 32 bytes. The block migration reduces global wire delay from L2 hit latency by moving frequently accessed cache blocks closer to the requesting processor. The design that uses the block migration is referred as CMP-DNUCA. CMP-DNUCA physically separates the cache banks into 16 different bankcluster, shown as the shaded "tetris" pieces in figure 2.11, furthermore CMP-DNUCA logically separates the L2 cache banks into 16 unique banksets. Each bankcluster contains one bank from every bankset. The bank cluster are grouped into three distinct region: the local region, the central region and the inter region. In figure 2.11 they are shown using different grey tones shading. The CMP-DNUCA implements a simple static allocation policy for the new line insertions based on the low-order bits of the cache tags to select a bank within the block's bankset. The migration policy of CMP-DNUCA moves blocks along a six bankcluster chain:

$$OtherLocal \rightarrow OtherInter \rightarrow OtherCentral \rightarrow MyCentral \rightarrow MyInter \rightarrow MyLocal$$

The search for a line is based on a two-phase multicast policy: in a first step the request is broadcasted to the appropriate banks

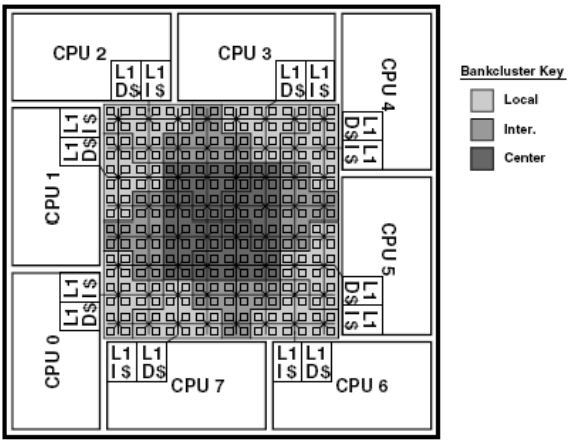

Figure 2.11: *The Tetris shaped NUCA-based CMP system*

within the six previously listed bankclusters then, if all of them report a miss, the request is broadcasted to the remaining 10 banks of the bankset. On-chip transmission line technology reduces L2 cache access latency be replacing slow conventional wires with ultra-fast transmission lines. The delay in conventional wires is dominated by a wire's resistance-capacitance product, or RC delay. The RC delay increases with improving technology as wires becomes thinner to match the smaller feature sizes below. Specifically, wire resistance increases due to the smaller cross-sectional area and sidewall capacitance increases due to the greater surface area exposed to adjacent wires. On the other hand, transmission lines attain significant performance benefit by increasing wire dimensions to the point where the inductance-capacitance product (LC delay) determines delay. While on-chip transmission lines achieve significant latency reduc-

tion, they sacrifice substantial bandwidth and require considerable manufacturing costs.

2.3 Coherence protocols

2.3.1 DASH multiprocessor

Directory Architecture for SHared memory (DASH) [38, 39] is a scalable shared- memory multiprocessor currently being developed at Stanford's Computer Systems Laboratory. A key feature of DASH is its distributed directory-based cache coherence protocol. Unlike traditional snoopy coherence protocols, the DASH protocol does not rely on broadcast; instead it uses point-to-point messages sent between the processors and memories to keep caches consistent. Furthermore, the DASH system does not contain any single serialization or control point.

The architecture. The architecture consists of powerful processing nodes, each with a portion of the shared-memory, connected to a scalable, high-bandwidth low- latency interconnection network. The physical memory in the machine is distributed among the nodes of the multiprocessor, with all memory accessible to each node. Each processing node, or cluster, consists of a small number of high-performance processors with their individual caches, a portion of the shared-memory, a common cache for pending remote accesses, and a directory controller - corresponding to its portion of the shared physical memory - interfacing the cluster to the network; the directory memory stores the identities of all remote nodes caching that block. A bus-based snoopy scheme is used to keep caches coherent within a cluster, while inter-node cache consistency is maintained using a distributed directory-based coherence protocol. The high-level organization of the protocol is shown in Figure 2.12.

The coherence protocol. The DASH coherence protocol is an invalidation-based ownership protocol. A memory block can be in one of three states as indicated by the associated directory entry: i) uncached-remote, that is not cached by any remote cluster; ii) shared-remote, that is cached in an unmodified state by one or more remote clusters; or iii) dirty-remote, that is cached in a modified state by a single remote cluster. The directory does not maintain information concerning whether the home cluster itself is caching a memory block because all transactions that change the state of a memory block are issued on the bus of the home cluster, and the snoopy bus protocol keeps the home cluster coherent. The protocol maintains the notion of an owning cluster for each memory block. The owning cluster is nominally the home cluster. However, in the case that a memory block is present in the dirty state in a remote cluster, that cluster is the owner. Only the owning cluster can complete a remote reference for a given block and update the directory state.

In case of a read request coming from the processor, if the location is present in the processor's first-level cache, the cache simply supplies the data. If not present, then a cache fill operation must bring the required block into the first level cache. A fill operation first attempts to find the cache line in the processor's second-level cache, and if unsuccessful, the processor issues a read request on the bus. This read request either completes locally or is signaled to retry while the directory board interacts with the other clusters to retrieve the required cache line. The check for a local copy is initiated by the normal snooping when the read is issued on the bus. If the cache line is present in the shared state then the data is simply transferred over the bus to the requesting processor and no access to the remote home cluster is needed. If the cache line is held in a dirty state by a local processor, the directory controller takes the

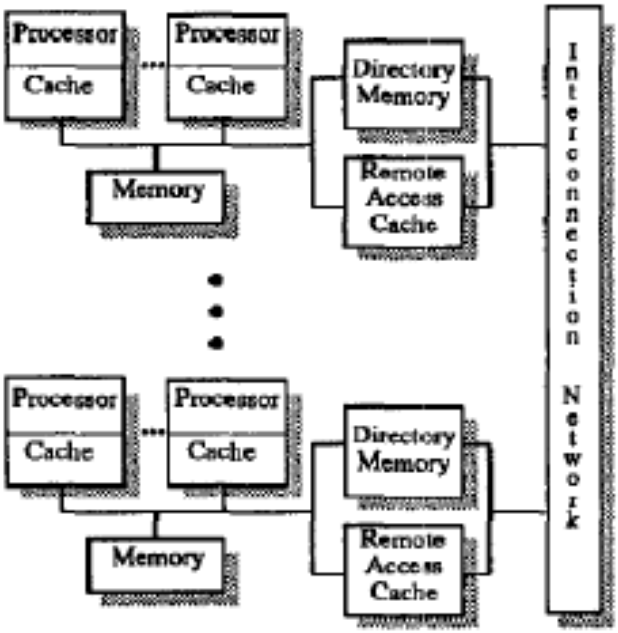

Figure 2.12: *DASH architecture*

ownership of that block. If a read request cannot be satisfied by the local cluster, the processor is forced to retry the bus operation, and a request message is sent to the home cluster; at the same time, an entry is allocated in the Remote Access Cache (RAC). When the read request reaches the home cluster, it is issued on that cluster's bus. This causes the directory to look up the status of that memory block. If the block is in an uncached remote or shared-remote state the directory controller sends the data over the reply network to the requesting cluster. It also records the fact that the requesting cluster now has a copy of the memory block. If the block is in the dirty-remote state, however, the read request is forwarded to the owning, dirty cluster. The owning cluster sends out two messages in response to the read. A message containing the data is sent directly to the requesting cluster, and a sharing writeback request is sent to the home cluster. The sharing writeback request writes the cache block back to memory and also updates the directory. In case of write operations initiated by a store from the processor, a read-exclusive request transaction begins. In case of miss in both first and second level cache, a read exclusive request is issued to the bus to acquire sole ownership of the line and retrieve the other words in the cache block. Once the request is issued on the bus, it checks other caches at the local cluster level. If one of those caches has that memory block in the dirty state (it is the owner), then that cache supplies the data and ownership and invalidates its own copy. If the memory block is not owned by the local cluster, a request for ownership is sent to the home cluster. As in the case of read requests, a RAC entry is allocated to receive the ownership and data. At the home cluster, the read-exclusive request is echoed on the bus. If the memory block is in an uncached-remote or shared-remote state the data and ownership are immediately sent back over the reply network. In addition, if the block is in the shared-

remote state, each cluster caching the block is sent an invalidation request. The requesting cluster receives the data as before, and is also informed of the number of invalidation acknowledge messages to expect. Remote clusters send invalidation acknowledge messages to the requesting cluster after completing their invalidation. Instead, if the directory indicates a dirty-remote state, then the request is forwarded to the owning cluster as in a read request. At the dirty cluster, the read-exclusive request is issued on the bus. This causes the owning processor to invalidate that block from its cache and to send a message to the requesting cluster granting ownership and supplying the data. In parallel, a request is sent to the home cluster to update ownership of the block. On receiving this message, the home sends an acknowledgment to the new owning cluster.

2.3.2 SGI Origin

The SGI Origin 2000 [39] is a cache-coherent non-uniform memory access (ccNUMA) multiprocessor designed and manufactured by Silicon Graphics, Inc. The Origin system was designed from the ground up as a multiprocessor capable of scaling to both small and large processor counts without any bandwidth, latency, or cost cliffs. The Origin system consists of up to 512 nodes interconnected by a scalable Craylink network. Each node consists of one or two processors, up to 4 GB of coherent memory, and a connection to a portion of the X I 0 10 subsystem. This systems employs distributed shared memory (DSM), with cache coherence maintained via a directory-based protocol.

The architecture. A block diagram of the SGI Origin architecture is shown in Figure 2.13. The basic building block of the Origin system is the dual-processor node. In addition to the processors, a node contains up to 4 GB of main memory and its correspond-

ing directory memory, and has a connection to a portion of the IO subsystem. The nodes can be connected together via any scalable interconnection network. The cache coherence protocol employed by the Origin system does not require in-order delivery of point-to-point messages to allow the maximum flexibility in implementing the interconnect network. The DSM architecture provides global addressability of all memory. While the two processors share the same bus connected to the Hub, they do not function as a snoopy cluster. Instead they operate as two separate processors multiplexed over the single physical bus.

The coherence protocol. Like the DASH [38] protocol, the Origin cache coherence protocol is non-blocking. Memory can satisfy any incoming request immediately; it never buffers requests while waiting for another message to arrive. The Origin protocol also employs the request forwarding of the DASH protocol for three party transactions. Request forwarding reduces the latency of requests which target a cache line that is owned by another processor. In order to prevent deadlock, two separate networks are provided for requests and replies.

2.3.3 Token Coherence

A technique proposed in 2003, token coherence, directly enforces the coherence invariant through a simple technique of counting and exchanging tokens. Token coherence [40] associates a fixed number of tokens with each block. In order to write a block, a processor must acquire all the tokens. To read a block, only a single token is needed. In this way, the coherence invariant is directly enforced by counting and exchanging tokens. Cache tags and messages encode the number of tokens using Log2N bits, where N is the fixed number of tokens for each block. Token coherence allows processors

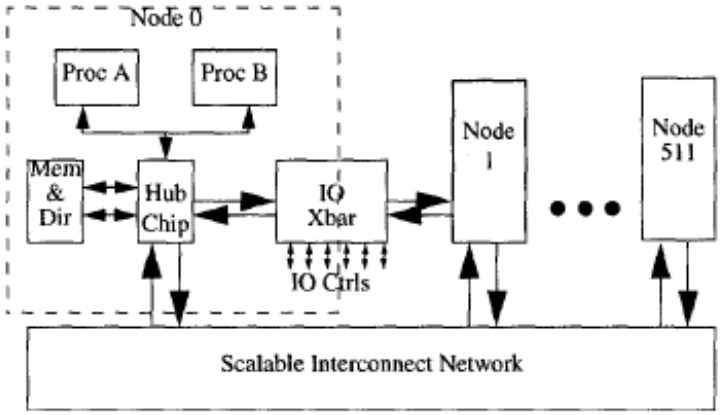

Figure 2.13: *SGI diagram*

to aggressively seek tokens without regard to order. A performance policy is used to acquire tokens in the common case. For example, a processor in a multiprocessor could predict which processor possesses the tokens and only send a message directly to it. However prediction can be incorrect and a processor's request may fail to acquire the needed tokens. Thus while a performance policy seeks to maximize performance, token coherence also provides a correctness substrate to ensure coherence and liveness. There are two parts to the correctness substrate: safety and liveness. Coherence safety ensures the coherence invariant at all times by counting tokens. Ensuring liveness means that a processor must eventually satisfy its coherence request. Since the requests used by the performance policy, transient requests, may fail, the correctness substrate provides a stronger type of request that always succeeds once invoked. These persistent requests, when invoked, ensure liveness by leaving

state at all processors so that in-flight tokens forward to the starving processor. Different mechanisms ensure that only one persistent request for a given block is active, and that starving processors eventually get to issue a persistent request. With a correctness substrate in place, a performance policy uses transient requests to locate tokens and data in the common case. The TokenB performance policy targets small-scale glueless multiprocessors. TokenB broadcasts a requestor's GETM and GETS message to every node in the system. Nodes respond to GETS and GETM requests with tokens and possibly data. An owner token designates which sharer should send data to the requestor. Since TokenB operates on an unordered interconnect and does not establish an ordering point, races may cause requests to fail. For example, P1 and P5 may both issue GETM requests for a cache line. Sharer P2 might respond to P1's request with a subset of tokens and sharer P6 might respond to P5's request with another subset of tokens. Since both requests require all tokens, both requests fail to acquire the needed permission. TokenB detects the possible failure of a request by using a timeout. After the timer expires, TokenB may issue a fixed number of retries before it activates a persistent request (to establish the order of racing requests). Replacements in token coherence are straightforward. The replacing processor simply sends a message with the tokens to the memory controller without additional control messages. Token counting ensures coherence safety regardless of requests that race with writeback messages. However, completely silent replacement of unmodified shared data is not possible and tokens must replace to memory. Token coherence enables a broadcast protocol on an unordered interconnect. Due to this aspect, it doesn't scale with the number of processors: when the correctness substrate starts to operate, it strongly relays on broadcast communication to all the processors nodes that can have a private copy of a given block.

When the number of possible sharers grows, then the number of broadcast messages that has to be sent grows with it. As future CMP are expected to have thousands of core per chip, broadcast communication represents a bottleneck from the scalability point of view.

Chapter 3

The coherence protocols implementation

Contents

3.1 MESI and MOESI features

This section introduces directory-based version of both MESI and MOESI. Such kind of protocols have had a renewed relevance in the context of CMP systems, but it is difficult to find a detailed description of their characteristic in recent CMP papers. In the considered version, both the protocols relay on a shared L2 cache: the processors of the CMP systems have private L1 caches, and have to access a shared L2 cache when a referred block misses in the private

cache; the on-chip cache hierarchy is supposed to be inclusive, and this property has to be enforced when a block is evicted from the L2 caches. The directory information is held in the shared cache: in this way, it is possible to avoid the need of holding a full directory of all memory blocks; instead, only for actually cached blocks the corresponding state and directory information is managed by the L2 directory. Of course, if the shared L2 cache is sub-banked (i.e., it is a NUCA cache), then the directory is held in the L2 bank each block is mapped to, according to any mapping policy. When a block is missed in the private L1 cache, an appropriate request message is built and sent to the shared L2 directory, that works in order to provide the L1 requestor with the most up-to-date copy of the block. An important characteristic of the directory is that it is non-blocking (with the exception of a new request received for a L2 block that is involved in a L2 replacement). A non-blocking directory is able to serve a subsequent request for a given block even if this is still undergoing on a previous transaction, without the need of stalling the request or nacking it [26]. Both the protocols relay on three virtual networks [16, 15, 52]: the first one (called vn0) is dedicated to requests that the L1s send to the interested L2 directory; the second one (called vn1) is used by the L2 directory to provide the requesting L1 with the needed block (L2-to-L1 transfer), but also by the L1 that has the unique copy of the block to send it to the requesting L1 (L1-to-L1 transfer); the last one (called vn2) is used by the L2 directory to forward the request received by an L1 requestor to the L1 cache that holds the unique copy of the block (L2-to-L1 request forwarding). The protocols were design without requiring total ordering of messages. In particular, vn0 e vn1 were developed without any ordering assumption, while vn2 only requires point-to-point ordering. The reason of such choice is that the performance of NUCA cache are strongly influenced by the per-

formance (and thus by the circuital complexity) of network switches [3, 4, 5, 17]. By utilizing wormhole flow control and static routing it is possible to design high-performance switches [15], particularly suited for NUCA caches.

3.2 MESI coherence protocol

The base version of our MESI protocol is similar to the one described in [26]. A block stored in L1 can be in one of the four states M (Modified: this is the unique copy among all the L1s, and the datum is dirty with respect to the L2 copy), E (Exclusive: this is the unique copy among all the L1s, and the datum is clean with respect to the L2 copy), S (Shared: this is one of the copies stored in the L1s, and the datum is clean with respect to the L2 copy) and I (Invalid: the block is not stored in the local L1). The L1 controller receives LOAD, IFETCH and STORE from the processor; in case of hit, the block is provided to the processor, and the corresponding coherence actions are taken; in case of miss, the corresponding request is build as a network message and is sent to the L2 directory through the vn0 (LOAD and IFETCH requests generate the same sequence of actions in any case, so from this moment on we consider only LOAD and STORE operations). When the L2 directory receives a request coming from any of the L1s, it can result in a hit or in a miss. In case of hit, the corresponding sequence of coherence actions is taken; in case of miss, a GET message is sent to the Memory Controller, a block is allocated to the datum, and the copy goes in a transient state [52] while waiting for the block; when the block is received from the off-chip memory, it is stored in the L2 cache, and a copy is sent to the L1 requestor. In the following are discussed the actions taken by the L1 controller when a LOAD or a STORE is received, assuming that a L1-to-L2 request always hits in the L2 bank.

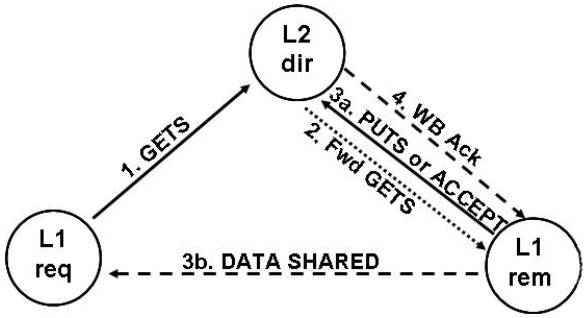

Figure 3.1: *Sequence of messages in case of Load Miss, when there is one remote copy. Contiguous lines represent request messages travelling on vn0; non-contiguous lines depict response messages on vn1; dotted lines represent messages travelling on vn2.*

3.2.1 Protocol actions

- *Load hit.* The L1 controller simply provides the processor with the referred datum, and no coherence action is taken.

- *Load miss.* The block has to be fetched from the L2 cache: a GETS (GET SHARED) request message is sent to the L2 home directory on the vn0, a block in the L1 is allocated to the datum, and the copy goes in a transient state while waiting for the block. When the L2 directory receives the GETS, if the copy is already shared by other L1s, the requestor is added to the sharers list, and the block is provided, marked as shared, directly by the L2 cache; if the block is present in exactly one L1, the L2 directory assumes the copy might be dirty, and the request is forwarded to the remote L1 on vn2, then the L2 copy goes in a transient state while waiting for a response. When the remote L1 receives the forwarded GETS,

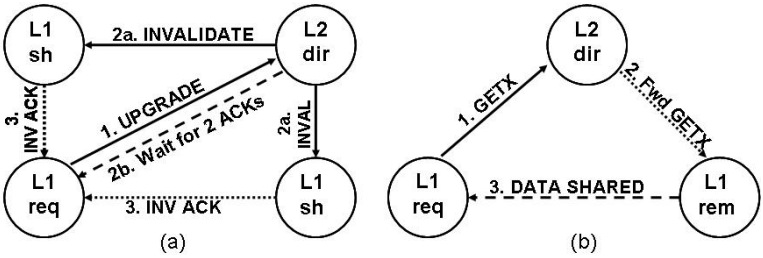

Figure 3.2: *Sequence of message in case of Store Hit (a) when the block is shared by two remote L1s, and Store Miss (b) when there is one remote copy*

provides the L1 requestor with the block, then issues toward the L2 directory -on vn0- a PUTS message (PUT SHARED: it carries the latest version of the block to be sent to the bank -the L2 copy has to be updated) if the local copy was in M, or an ACCEPT message (a control message that notifies that the L2 copy is still valid) if the local copy was in E; once the L2 directory receives the response from the remote L1, updates directory information, a WriteBack Acknowledgment is sent to the remote L1, and the block is marked as Shared. Figure 3.1 shows this sequence of actions. Of course, if the block is not present in any of the L1s, the L2 directory directly sends an exclusive copy of the block to the L1 requestor. When the block is received by the original requestor on vn1, if it is marked as shared the copy goes in the S state, otherwise it goes in E.

- *Store hit.* If the block is in M, the L1 controller provides the processor with the datum, and the transaction terminates.

If the block is in E, the L1 controller provides the processor with the datum, the state of the copy changes to M and the transaction terminates. If the copy is in S, the L1 controller sends a message to the L2 directory, on vn0, in order to notify it that the local copy is going to be modified, and the other shares have to be invalidated, then the copy goes in a transient state waiting for the response from the L2 directory. When the L2 directory receives the message from the L1, sends an Invalidation message to all the sharers (except the current requestor) on vn2, then clears all the sharers in the block's directory, and sends on vn1 to the current requestor a message containing the number of Invalidation Ack to be waited for. When a remote L1 receives a Invalidation for an S block, sends on vn1 an Invalidation Ack to the L1 requestor, then the copy is invalidated. Once the L1 requestor has received all the Invalidation Acks, the controller provides the processor with the requested block, the block is modified, then the state changes to M and the transaction terminates. Figure 3.2a shows this situation.

- *Store miss.* A GETX (GET EXCLUSIVE) message is sent to the L2 directory on vn0, a cache block is allocated to the datum and the copy goes in a transient state, waiting for the response. When the L2 cache receives the GETX, if there are two or more L1 sharers for that block, the L2 directory sends the Invalidation messages to all the sharers on vn2, then sends the block, togetherr with the number of Invalidation Acks to be received, to the current L1 requestor, on vn1; from this moment on, everything works as in the case of Store Hit of a block in the S state. If there are no sharers for that block, the L2 directory simply stores the ID of the L1 requestor in the

block's directory information and sends on vn1 the datum to the L1 requestor. If there is just one copy stored in one L1, the L2 assumes it is potentially dirty, and forwards the request on vn2 to the L1 that holds the unique copy, then updates the block's directory information clearing the old L1 owner and setting the new owner to the current requestor. When the remote L1 receives the GETX in forwards, sends the block to the L1 requestor on vn1, then invalidates its copy. At the end, the L1 requestor receives the block, then the controller provides the processor with the datum, the state of the copy is set to M and the transaction terminates. Figure 3.2b depicts this sequence of actions.

- *Replacement from L1 cache.* In case of conflict, the L1 controller chooses a block to be evicted from the cache, adopting a pseudo-LRU replacement policy. If the block is in the S state, the copy is simply invalidated, without notifying the L2 directory that the copy is no longer locally cached (as a consequence, the L1 has to reply to invalidations received for blocks that are no longer cached). If the block is either in the M or in the E state, the L1 Controller sends a PUTX (PUT EXCLUSIVE, in case of M copy: this message contains the last version of the block to be stored in the L2 cache) or an EJECT (in case of E copy: this is a very small control message that simply notifies the L2 directory that the block has been evicted by the L1, but the old value is still valid). When the L2 cache receives one of those messages, updates the directoryinformation by removing the L1 sender, updates the block value in case of PUTX, then issues a WriteBack Acknowledgment to the L1 sender; once this receives the acknowledgment, invalidates the copy.

- *Replacement from L2 cache.* As the cache hierarchy is supposed to be inclusive, when a block has been selected for eviction and is going to be replaced, the L2 directory must invalidate all the private L1 copies of that block, if any. If the copy was present in L2 but not in any L1 cache, then it can be directly evicted (it is invalidated if it was clean with respect to the main memory copy, otherwise a copy is sent to the memory). If the copy was present in exactly one L1 cache, an invalidation message is sent to the current owner, that will respond with either the copy (in case it was locally modified) or a simple acknowledgment to the directory. If the copy was shared by more L1 caches, the directory assumes its copy is up-to-date, so invalidates the L1 copies, waits for all the acknowledgments, then evicts its copy.

3.3 MOESI coherence protocol

The MOESI coherence protocol adopts the same four states M, E, S, and I that characterize MESI, with the same semantic meaning; the difference is that MOESI adds the state O for the L1 copies (Owned: the copy is shared, but the value of the block is dirty with respect to the copy stored in the L2 cache). The L1 that holds its copy in the O state is called the owner of the block, while all the other sharers have their copies stored in the classical S state, and are not aware that the value of the block is dirty with respect to the copy of the L2 cache. For this reason the owner has to maintain the information of dirty copy, and update the L2 value of that block in case of L1 Replacement. Also this MOESI coherence protocol is designed with the L2 directory that is a non-blocking directory. In the following are presented only the differences with MESI, referring to the Owner state.

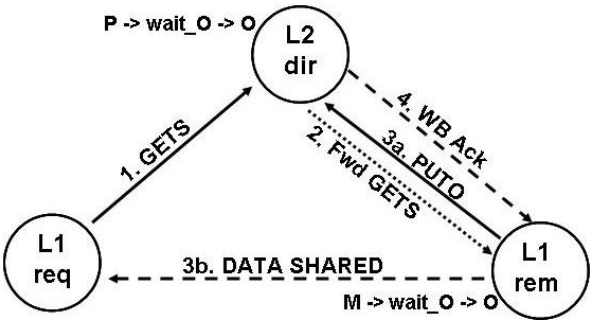

Figure 3.3: *Sequence of messages in case of Load Miss, when the block is modified in one remote L1. The remote copy is not invalidated; instead, when the WriteBack Ack is received by the remote L1, it is marked ad Owned*

3.3.1 Protocol actions

- *Load hit.* The L1 controller simply provides the processor with the referred datum, and no other coherence action is taken.

- *Load miss.* The GETS message is sent to the L2 cache on vn0. When the L2 directory receives the request, if the copy is private of a remote L1 cache, the L2 cache assumes it is potentially dirty and forwards the GETS to the remote L1 through the vn2, then goes in a transient state while waiting for a response. When the remote L1 receives the forwarded GETS, if the block is dirty (i.e. in the M state) then a PUTO (PUT OWNER: this control message notifies the L2 directory that the block is dirty and is going to be owned) is sent to the L2 cache, otherwise if the block is clean (i.e. in the E state) then an ACCEPT message is sent to the L2 directory

in order to notify it that the copy was not dirty, and the copy has to be considered as Shared and not Owned; in both cases, the remote L1 sends a copy of the block to the current requestor on vn1, and the L2 directory responds to the remote L1 owner with a WriteBack Acknowledgment, then updates the directory information by storing that the block is either owned, in case of PUTO, by the remote L1 or shared, in case of ACCEPT. Once the L1 requestor receives the block, the controller provides the processor with the referred datum, then the copy is stored in the S state. Figure 3.3 illustrates this sequence of actions. If the copy was already Owned, when the L2 directory receives the GETS request, simply adds the L1 requestor to the sharers list and forwards the request to the owner, that will provide the L1 requestor with the last version of the block.

- *Store hit.* When a store hit occurs for an O copy, the sequence of steps is the same as in the case of a store hit for an S copy in MESI.

- *Store miss.* The GETX message is sent to the L2 directory through the vn0. If the block is tagged as Owned by a remote L1, the GETX is forwarded through the vn1 to the current owner (together with the number of Invalidation Acknowledgment to be waited by the L1 requestor) and an Invalidation is sent to the other sharers in the list, then the sharers list is empty and the L1 requestor is set as the new owner of the block. When the current owner receives the forwarded GETX, sends the block to the L1 requestor together with the number of Invalidation Acknowledgment that it has to wait, then the local copy is invalidated. Once the L1 requestor has received the block and all the Invalidation Acknowledgment,

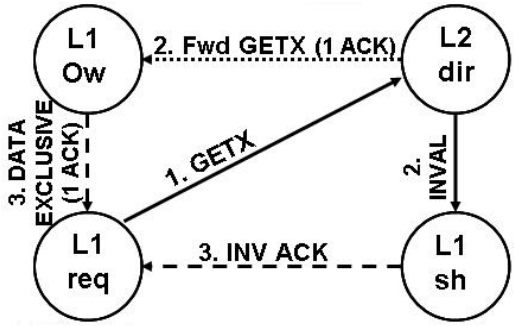

Figure 3.4: *Sequence of messages in case of Store Miss when the copy is Owned by a remote L1*

the cache controller provides the processor with the referred datum, then the block is modified and stored in the local cache in the M state. Figure 3.4 shows this case.

- *Replacement from L1 cache.* When the L1 controller wants to replace a copy in O, sends a PUTX message to the L2 directory. Once this message has been received, the L2 cache updates the directory information by clearing the current owner, then stores the new value of the block in its cache line, and sends a WriteBack Acknowledgment to the old owner (from this moment on, the block is supposed to be Shared as in the case of MESI). When the owner receives the WriteBack Acknowledgment, invalidates its local copy.

- *Replacement from L2 cache.* When the L2 controller wants to replace an Owned copy, knows that copy is potentially dirty in the L1 that holds the ownership, and that all the other copies are coherent with that value. So, the invalidation message

is sent to all the sharers and to the owner. The sharers will respond with a simple acknowledgment, while the owner sends either the modified copy of another acknowledgment. Once the L2 cache has collected all the response, the block is evicted (invalidated if the memory copy is up-to-date, updated if the memory copy was stall).

3.4 Non-blocking directory

A directory coherence protocol may relay on some mechanism, such as the adoption of NACK/retry messages or requests buffering, in order to prevent race and deadlock conditions that could affect the system correctness. For example, if a subsequent request is received by the directory when it is in some "busy" state, such request might be NACKed or buffered while waiting for the previous transaction to complete; the use of NAKs is the case of the SGI Origin coherence strategy [39]. Such kind of behavior is known as blocking directory. A non-blocking scheme adopts a directory node that is always able to serve an incoming request, even if for the requested memory block it is still undergoing on a previous transaction. Typically, such schemes are able to immediately update the directory state of a memory block when a message arrives at the home node, without the need of waiting any response; for those cases that do need a response to the directory (i.e. passing through one or more transient states can't be avoided), the home node has to be able to satisfy subsequent requests even if that response message has not been received yet. The directory versions of the MESI and MOESI protocols considered in this dissertation adopt a non-blocking scheme for the L2 home directory, except in the case of a new request received when it is engaged in a L2 replacement. In fact, when a block is going to be evicted from the L2 cache, all the L1 copies have to be inval-

idated; for this reason, if another L1 requestor accesses that block, the request can't be served until the replacement terminates (and the conflicting block can be loaded). During this period, the new request is not popped by the incoming queue (on the vn0); in order to prevent subsequent requests for different blocks to be stalled, the L2 controller reads from the incoming queue the messages that arrives after the blocked request, and serves them as usually. The adopted non-blocking strategy strongly relays on the ordering property of vn2: such virtual network is used by the L2 cache to forward the request toward an owner, but also to send Writeback acknowledgments and Invalidation messages. By ensuring that the point-to-point ordering property for such classes of messages is guaranteed, the L2 can assume that directory information stored in the L2 TAG field is always up-to-date and consistent with the final status of all the L1 nodes.

3.5 Main differences

From a design point of view, MESI has four base L1 states that can be represented with 2 bits. Instead, MOESI introduces an extra state, and thus L1 base states need an extra bit to be represented. However, a key feature of MOESI is that it privileges L1-to-L1 block transfers, while MESI presents a higher number of L2-to- L1 block transfers. In fact, in case of MESI the L2 directory forwards a new request to the "owner "only when there is just one copy in a remote L1 cache. Instead, MOESI has a wider concept of ownership with respect to MESI, because an L1 copy can be tagged as Owned, meaning the block is shared but the value of the L2 copy is to be updated; when the L2 directory receives a new request for an Owned copy, the request must be forwarded to the owner, because only the owner has the latest value of the block. MESI doesn't

have this feature, because when a GETS is forwarded on vn2 to the L1 remote cache that holds the unique L1 copy, if the block was previously modified (e.g., due to a previous Store Hit that caused a transition from E to M in the local L1 copy) the L2 copy is updated with a PUTS message. In this dissertation, this different behaviors have been highlighted as they are expected to be responsible for differences in performance and total amount of traffic in future CMP systems. Choosing between MESI and MOESI may be not so easy as one might expect, as the relative position of L1 requestor, L2 directory and the L1 owner influences the average L1 miss latency in different ways, depending on how long the miss transaction takes to be completed. In a NUCA environment, such difference is not obvious, as the access time to the shared L2 cache is not uniform, thus each L2 access exhibit a different latency. Moreover, if the request has to be forwarded to a remote L1-owner, the distance between the L2 bank and L1 owner, together with the distance between the L1 owner and L1 requestor, make the L1 miss latency more application dependent.

Chapter 4

Design tradeoff in S-NUCA CMP systems

Contents

This chapter presents our analysis of performances variations changing the topology and adopting different coherence protocols in a CMP system with large L2 shared NUCA cache as we proposed in [20]

4.1 Introduction

In the past, Distributed Shared Memory (DSM) systems with coherent caches were proposed as an high-scalable architectural solution, as they were characterized by powerful processing nodes, each with a portion of the shared memory, connected through a scalable interconnection network [38, 39]. In order to maintain high level of scalability with respect to the number of cores, the coherence

protocol usually adopted in such system was a directory coherence protocol, where directory information was held at each node. Directory coherence protocols rely on message exchange between the nodes that need a copy of a given cache block, and the home node (i.e. the node in the system that has to manage directory information for the block). With the increasing number of transistors available on-chip due to technology scaling [1], multiprocessor systems have shifted from multi-chip systems to single-chip systems (Chip Multiprocessors, CMP) [45, 30], in which two or more processors exist on the same die. Each processor of a CMP system has its own private caches, and the last level cache (LLC) can be either private [37, 42] or shared among all cores [51, 36, 10, 43]; hybrid designs have been also proposed [14, 12, 56]. CMPs are characterized by low communication latencies with respect to classical many-core and DSM systems, as the signal propagation delay in on-chip lines is lower than in off-chip wires [31]. However, as clock frequencies increase as well as the delay in communication lines, signals need more clock cycles to be propagated on the chip, thus resulting in higher wire delay, and this delay significantly affects performance [30, 39]. In order to face the wire delay problem, Non-Uniform Cache Access (NUCA) architecture [34, 32, 10] has been proposed: a NUCA is a bank-partitioned cache in which the banks are connected by means of a communication infrastructure (typically, a Network-on-chip, NoC [16, 15]), and it is characterized by a non-uniform access time. NUCAs have been proved to be effective in hiding the effects of wire delay. When adopted in CMP systems, a NUCA typically represents the LLC shared among all the cores [10, 32], and all the private, lower cache levels have to be kept coherent by means of a coherence protocol; the cores in the system are able to communicate both among themselves and with NUCA banks. As NoCs are characterized by a message-passing communi-

cation paradigm, the communication among all kind of nodes in the system (i.e. shared cache banks and processor with private caches) is based on the exchange of many types of messages. In this context, the coherence protocol is implemented as a directory-based protocol, similar to those designed for DSM systems, in order to meet the same high degree of scalability. By exploiting the fact that the LLC is shared among all cores, our proposal is to adopt a non-blocking directory [26], that is distributed in NUCA banks: NUCA banks can be adopted as home nodes for cache blocks, and the directory information is stored in the TAG field of each block present in the NUCA. Previous works proposed various CMP architectures based on NUCA cache, each adopting as the base coherence protocol either MESI [14, 32] or MOESI [10]. However, to the best of our knowledge, none of them motivated the choice of neither the coherence protocol nor the system topology; instead, we believe that the behavior of a NUCA-based CMP is heavily influenced by both these aspects.

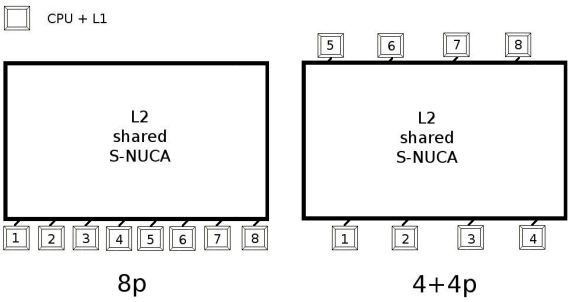

Figure 4.1: *The two considered S-NUCA CMP topologies*

4.2 Methodology

We considered two different configurations of a Shared L2 S-NUCA based CMP system with 8 processors, shown in Figure 4.1. We refer to each configuration as 8p (a) and 4+4p (b). We performed full-system simulation using Simics [50]. We simulated an 8-cpu UltraSparc II CMP system, each cpu using in-order issue, running at 5 GHz. We used GEMS [25] in order to simulate the cache hierarchy and coherence protocols: private L1s have 64 KB of storage capacity, 2 ways set associate Instructions and Data caches (32 KB each), while the shared S-NUCA L2 cache is composed by 256 banks (each of 64 KB, 4 ways set associative), for a total storage capacity of 16 MB; we assumed Simple Mapping, with the low-order bits of index determining the bank [34]. We assumed 2 GB of main memory with a 300-cycle latency. Cache latencies to access TAG and TAG+Data have been obtained by CACTI 5.1 [11] for the specified nanotechnology (65 nm). The NoC is organized as a partial 2D mesh network, with 256 wormhole [16, 15] switches (one for each NUCA bank); NoC link latency has been calculated using the Berkeley Predictive Model.

Table 4.1 summarizes the configuration parameters for the considered CMP. Our simulated system runs the Sun Solaris 10 operating system. We run applications from the SPLASH-2 [55] benchmark suite, compiled with the gcc provided with the Sun Studio 10 suite. Our simulations run until run completion, with a warm-up phase of 50 Million instructions.

4.3 Topology issue

Figure 4.2 shows four different cases in which two of them (a and b) represent the behavior of MESI, and in the others (c and d) is

Number of CPUs	8
CPU type	UltraSparcII
Clock Frequency	5 GHz (16 FO4 @ 65 nm)
L2NUCA Cache	16 MB, 256 x 64KB, 16 ways s.a.
L1 cache	Private 32 Kbytes I + 32 Kbytes D, 2 way s.a., 3 cycles to TAG, 5 cycles to TAG+Data
L2 cache	16 Mbytes, 256 banks (64 Kbyte banks, 4 way s.a., 4 cycles to TAG, 6 cycles to TAG+Data
NoC configuration	Partial 2D Mesh Network; NoC switch latency: 1 cycle; NoC link latency: 1 cycle
Main Memory	2 GByte, 300 cycles latency

Table 4.1: *S-NUCA simulation parameters*

depicted the behavior of MOESI. If the L2 home is placed close to the L1 requestor (a and c), then MESI should perform better than MOESI, because the data packets have to travel along a shortest path, thus resulting in lower latency and bandwidth occupancy; on the other hand, if the L2 home is far from the L1 requestor (b and d), than MOESI should outperform MESI, when the L1 requestor and owner are close (the big data packet has to traverse a shortest path) (d); otherwise (L1 owner and requestor are not close) the behavior of MOESI should be similar to the situation reported in Figure 4.2c. Of course, it is important to consider how much such L1-to-L1 transf-ers impact on performance. In particular, whether they represent a significant part of the total block transfers toward the L1 caches (i.e. how many L1-to-L1 transfers satisfy the total

L1-to-L2 requests), in the considered class of applications.

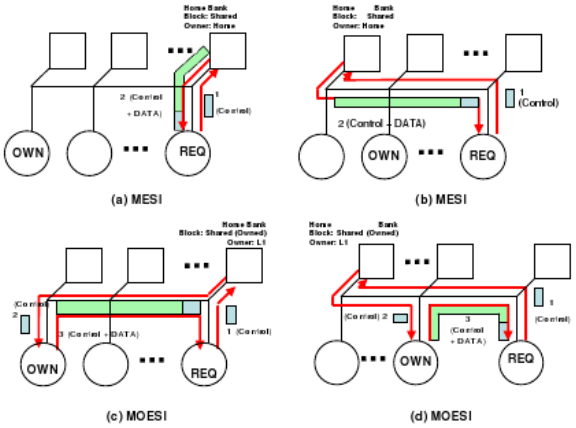

Figure 4.2: *Different topologies may take advantage from either MESI or MOESI*

How the topologies previously introduced can contribute in investigating such tradeoff? Figure 4.3 4 shows the 8p and the 4+4p configurations, for an S-NUCA cache composed by 16x16 banks, in which two L1 caches (green and yellow) issue a request message. The request issued by the green cpu (light green line) reaches the home bank (red), then it is forwarded (light green line) to an L1 owner (light blue L1), that will provide the requestor with the block (blue line). The request issued by the yellow cpu (pink line) reaches the home bank (blue) that directly provides the requestor with the block (orange line). As the application is the same, when moving from 8p to 4+4p topology, the mapping between blocks and S-NUCA banks doesn't change.

If the CPUs involved in the considered transactions are moved,

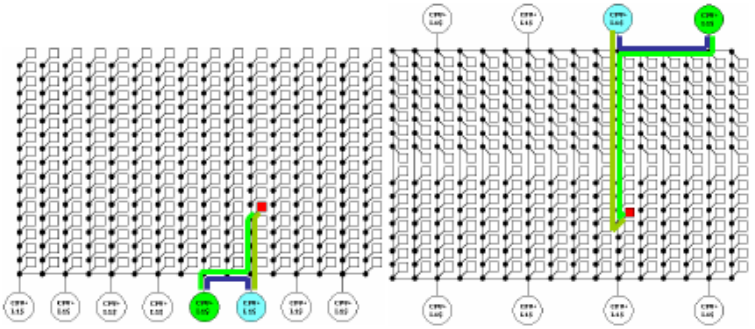

Figure 4.3: *The same application in two different configurations*

the network distance that each message has to traverse increases, and consequently the response latency is augmented. In particular, in the considered two hops transaction, with the 8p topology the data packet needs just one hop to reach the L1 requestor. When the requestor is moved to the other side of the cache, the data packet has to traverse 29 hops. Similar considerations can be done for the considered three hops transaction. This aspect affects not just the L2 response time, but also the NoC bandwidth utilization, and consequently the dynamic power consumption. In fact, if we assume a cache block of 64 bytes and a control message of 8 bytes, then a data packet is composed by 72 bytes. If such a large data packet has to traverse a lot of NoC links, then the bandwidth utilization increases as well as the response time.

4.4 Results

We simulated the execution of different benchmarks from the SPLASH-2 suite, running on the two different topologies (8p, 4+4p) we introduced before. We chose the Cycles-per-Instruction (CPI) as the reference performance indicator. Figure 4.4 shows the Normalized

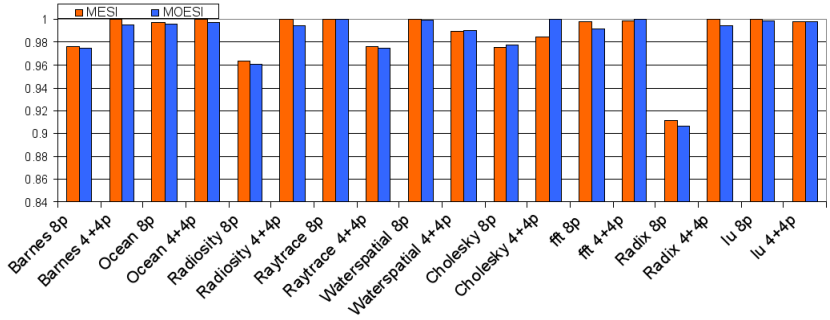

Figure 4.4: *Normalized CPI. The CPI is normalized with respect to the maximum CPI value for each benchmark*

CPI for the considered benchmarks. We notice that there is not a significant performance variation when the topology is fixed and the coherence protocols varies between MESI and MOESI (less than 1% in all the considered cases, except for Cholesky in the 4+4p configuration). To explain the little performance impact of the choice of the coherence protocol, we should consider that the main characteristic of MOESI is the wider concept of ownership it introduces with respect to MESI, leading to an increase of L1-to-L1 block transfer. Such block transfer may be faster or slower than L2-to-L1 transfer, depending on i) the bank access time, ii) network distance between L1 requestor and L2 directory, and iii) network distance between

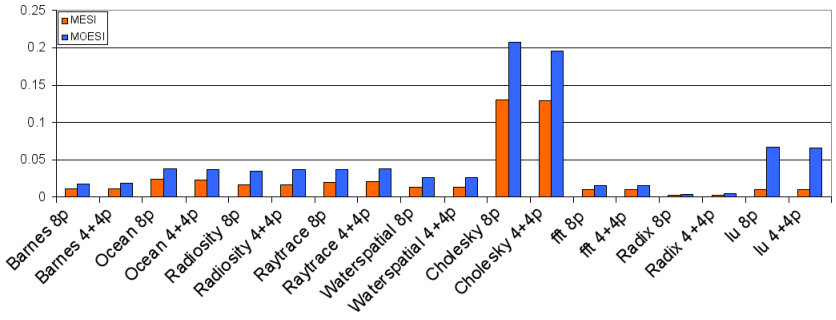

Figure 4.5: *(# L1-to-L1 transfers)/(# L1-to-L2 requests) Ratio*

L1 requestor and L1 Owner (the network distance is given by the number and length of links and number of switches that must be traversed in a L2-to-L1 or L1-to-L1 message transfer). Figure 4.5 shows the percentage of L1-to-L2 requests that are satisfied by L1-to-L1 transfers for the considered benchmarks. As the figure depicts, the number of L1-to-L1 block transfers increases for each topology when moving from MESI to MOESI, but the percentage of such three-hop transitions over all block transfers is, in the worst case, less than 6% (except for Cholesky). As a consequence, for the considered applications and protocol implementations, the CPI (Figure 4.4) and the miss latency (Figure 4.6) are not strongly influenced by the coherence protocol. The Cholesky exception in the 4+4p configuration is a consequence of the higher number of L1-to-L1 transfer (more than 14% for MESI, about 20% for MOESI). Figure 4.6 shows the average L1 miss latency. Such latency, when moving from 8p to 4+4p, can increase, decrease or stay constant depending on the running application. This is mainly due to the variation of the L2-to-L1 contribution to the L1 miss latency. The dependency may be explained

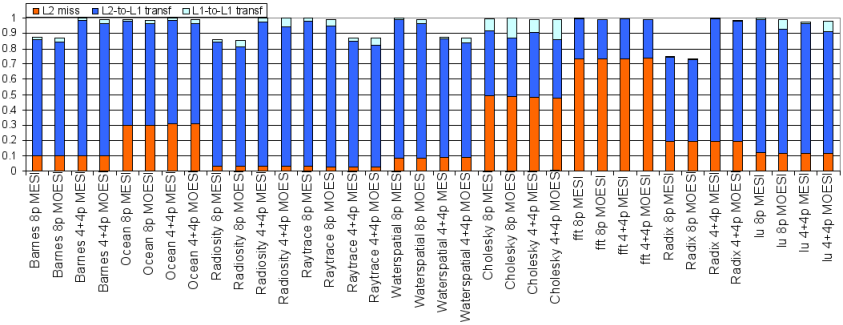

Figure 4.6: *Breackdown of Average L1 miss latency (Normalized)*

by considering how data are mapped in the NUCA cache, and how such data are accessed. In fact, in NUCA caches, the cache access time depends on the physical position of data (i.e., of the bank the data is mapped to) with respect to the CPUs. In particular, banks that are closer to CPUs exhibit lower access time as a consequence of the reduced number of switches and number and length of links to be traversed. In the considered topologies, the CPUs position varies with respect to NUCA banks, so the CPUs see a different access time. The variation shown in Figure 4.6, together with the fact that neither the L1 miss rate (Figure 4.7) nor the number and type of messages issued in the NoC (Figure 4.8) change with the topology, indicate that the access pattern to NUCA banks is not uniform. This has a direct impact on performance difference. In order to verify this aspect, we calculated the baricentre of the access frequency to each bank of the NUCA cache. We consider the NUCA cache as an ideal plane in which column indexes represent the abscissas while row indexes represent the ordinates. As the considered S-NUCA cache is organi-zed as a matrix of 16x16 banks, we

numbered the rows and the columns from 1 to 16. We define the NUCA's Baricentre as:

$$\mathrm{B} = \left[\mathrm{X} = \frac{\sum_{i=1}^{N} \left(i * \sum_{j=1}^{N} A_{i,j} \right)}{\sum_{i=1}^{N} \left(\sum_{j=1}^{N} A_{i,j} \right)}, \mathrm{Y} = \frac{\sum_{j=1}^{N} \left(j * \sum_{i=1}^{N} A_{i,j} \right)}{\sum_{j=1}^{N} \left(\sum_{i=1}^{N} A_{i,j} \right)} \right]$$

where Ai,j is the number of accesses to the bank of row i and column j. In the ideal case (all the banks present exactly the same access number) the Baricentre of the NUCA is (8.5;8.5). Figure 4.9 shows the baricentres of the NUCA of all the considered configurations. According to the Figure, we individuate three classes of applications, having the baricentre i) very close to the ideal case (e.g., ocean and lu), ii) in the lowest part of the S-NUCA (e.g., radix and barnes), or iii) in the highest part of the shared cache (e.g., raytrace and waterspatial). We observe three different behaviors: the "ocean class" of the applications don't present a significant performance variation when moving from 8p to 4+4p; cholesky presents a little performance degradation also for 4+4p, even if its baricentre is very close to the ideal case: this phenomenon is due to the great impact of L1-to-L1 transfers, that have to travel along longest paths. The "radix class" has a performance degradation when moving to 4+4p, as the most part of the accesses are in the bottom of the shared NUCA, so moving half (or more) of the cpus to distant sides of the NUCA leads to an increase of the NUCA's response time. In particular, radix is strongly unbalanced as its baricentre is near the bottom of the cache, leading to a performance degradation in the 4+4p configuration of about 10%. Finally, the "raytrace class" performs better with the 4+4p topology, as the most part of the accesses are in the top of the shared NUCA.

This feature is also confirmed by analyzing the CPI of Fig-

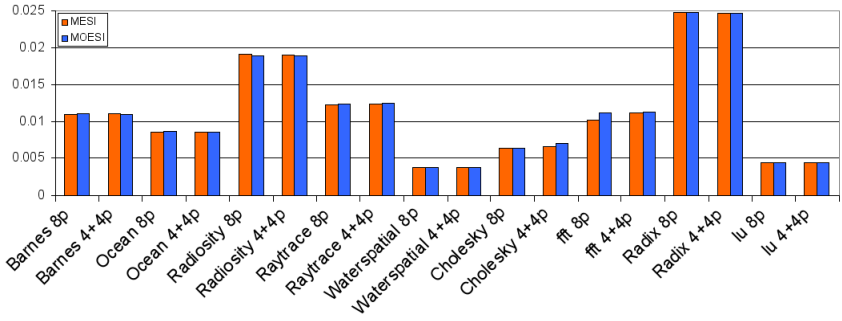

Figure 4.7: *L1 (I$+D$) miss rate (user+kernel)*

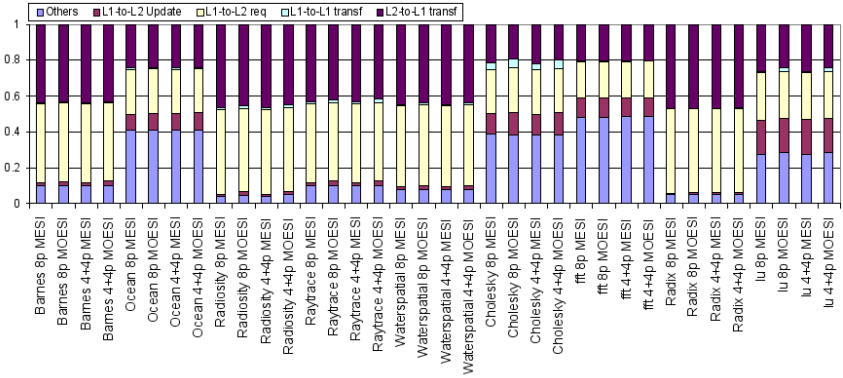

Figure 4.8: *Impact of different classes of messages on total NoC traffic*

ure 4.10 and the L1 miss latency of Figure 4.11, that compare, in the case of 8p configuration, architectures based on direct and inverse mapping policy. The 8p with inverted mapping has the CPUs connected to the opposite cache side with respect to the direct mapping. As the access pattern to L2 NUCA banks doesn't change, the baricentres in the two cases are the same, so moving all the CPUs to the other side of the chip leads to a different number of hops to reach the directory bank. As a result, we observe a performance degradation for applications of the "radix class", as a consequence of an increase in the average path length needed to reach the directory bank. For the "ocean class" of applications, there is not a significant performance variation, as a consequence of the central position of the baricentres. At the end, the "raytrace class" improve performance with the inverse mapping configuration.

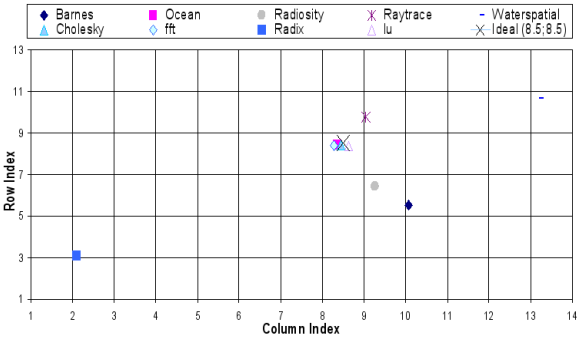

Figure 4.9: *Coordinates of accesses baricentres for the considered SPLASH-2 applications, in a 16x16 S-NUCA cache*

Another consequence of the imbalance of accesses is the variation of the bandwidth utilization in the NoC. Figure 4.12 shows how

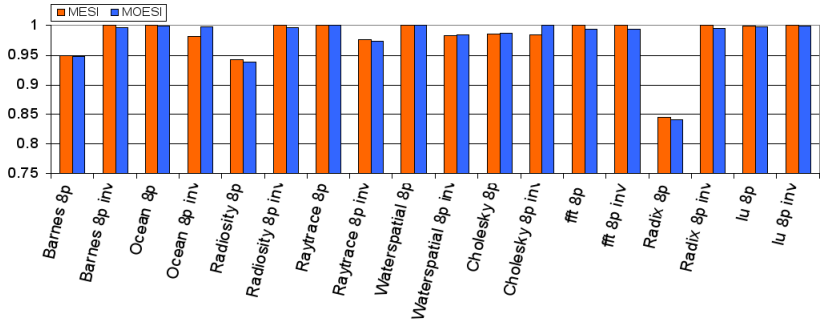

Figure 4.10: *Normalized CPI for the 8p configuration, direct vs inverse mapping*

each class of messages impacts on the bandwidth utilization. For the applications we considered, L2-to-L1 block transfers represent the higher component of total bandwidth utilization. In our implementation, control messages (L1-to-L2 Req and Others in the Figure) are composed by 8 bytes, while data messages are composed by 72 bytes (8 of control and the 64 bytes of the carried block): when the L2 directly provides the L1 requestor with the block, the 72 bytes of the message have to travel along the whole path toward the destination. As the number of L2-to-L1 transfers dominates the number of total block transfers, the relative component of the percentage of NoC bandwidth utilization is also dominant. When moving from 8p to 4+4p, the components of the bandwidth utilization varies depending on the position of the baricentre of each application: in fact, for those applications whose accesses are imbalanced toward the bottom (top) of the cache, when moving the CPUs to different sides of the chip, the higher (lower) number of hops to be passed to reach the directory bank and the L1 owner leads the NoC us-

age to increase (decrease). On the other hand, if the baricentre is very close to the ideal case, bandwidth utilization doesn't change. In conclusion, our evaluation shows that, while the choice of the coherence protocol doesn't have a significant impact on the whole system performance, the chip topology does have, in terms of both CPI and NoC bandwidth occupancy.

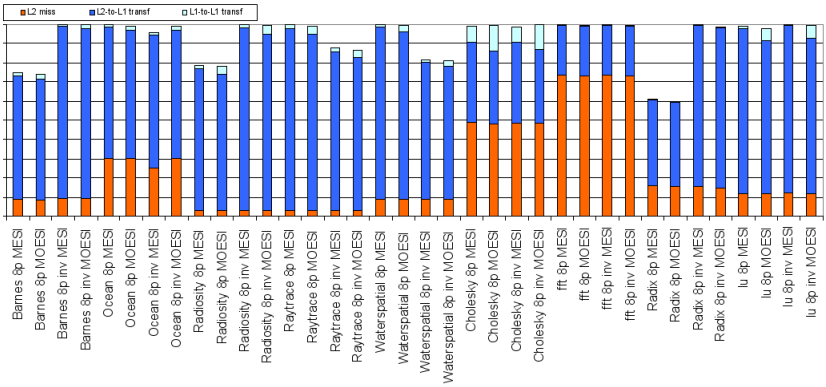

Figure 4.11: *Breakdown of Average L1 Miss Latency (Normalized) for the 8p configuration, direct vs inverse mapping*

For the considered benchmarks and designed coherence protocols, the 8p configuration presents the best performance with respect to 4+4p, for those application whose accesses are unbalanced in the bottom part of the S-NUCA. The 8p configuration performs worst then the 4+4p for applications that have the baricentre in the top part of the S-NUCA, or there is not a significant difference if the baricentre is close to the ideal case. We show that while choosing between MESI and MOESI is not a strict design issue, the CMP topology, and in particular the relative position of each CPU with

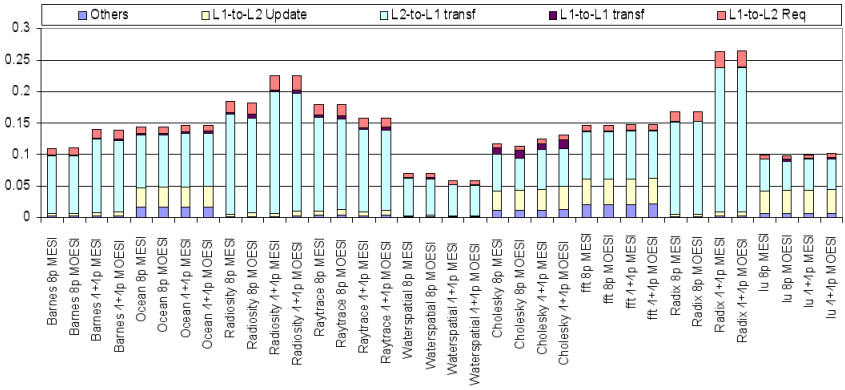

Figure 4.12: *Impact of different classes of messages on total NoC Bandwidth Link Utilization (%)*

respect to both the L2 directory bank and the other CPUs, is a key feature that designers have to take into account when designing an S-NUCA CMP. We observed the same behaviors for the total bandwidth utilization. This is also a central design point, as the dynamic component of the overall energy consumption is directly connected to NoC traffic, and thus to bandwidth utilization. Another important feature to be considered is the mapping policy: performance in static mapping case can be improved by placing most frequently accessed blocks in L2 banks closer to the CPUs. This can be achieved if the memory layout is properly managed by the compiler.

Chapter 5

CMP D-NUCA migration mechanism

Contents

This chapter presents our implementation of migration mechanism in D-NUCA architecture based on a MESI coherence protocol and the solutions we adopted to resolve the "multiple miss" and the "false miss" problems which are connected to this scenario.

5.1 Introduction

When designing a block migration protocol for NUCA caches, many race conditions have to be solved in order to guarantee correctness and prevent deadlock. While the most part of such race conditions can be easily managed with simple additional message exchange, both the false miss and the multiple miss require deep protocol modifications, and relay on strong network assumptions.

5.1.1 The false miss problem

A Multiple Miss occurs when two o more processors simultaneously send a request for the same block, and this block is not in cache; this generates multiple L2 misses and multiple requests to the main memory for the same line. Without managing properly this off-chip accesses, the off-chip memory could send the same line to different L2 banks of the same bankset. In a D-NUCA a physical address can be mapped in any bank of the bankset; consider a protocol in which:

- each L2 bank sends MISS to the L1 requestor if it doesn't have a valid copy of the data;
- when the L1 detects an L2 miss, it sends a request to the off-chip memory;
- the L2 bank located farther to the L1 requestor is the L2 entry point for the data;

If two processors located at the opposite sides of the DNUCA send simultaneously a request for the same data, both the requests will result in a miss, and the resulting off-chip accesses lead to have multiple copies of the same data. Figure 5.1 shows this particular race condition.

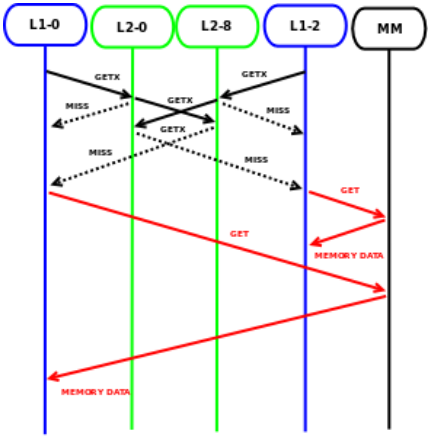

Figure 5.1: *The Multiple Miss problem*

5.1.2 The multiple miss problem

The false miss problem was first presented by Beckmann and Wood [49]. As a consequence of the migration mechanism, there could a be time interval in which none of the banks of the bankset is able to provide the requestor with the referred block, thus resulting on a L2 miss in spite of the actual on-chip presence of the block. To better understand this phenomenon, let's consider the sequence diagram shown in Figure 5.2.

In the reported scenario, the request sent by L1-0 generates a migration of the data from L2-8 to L2-4; a subsequent request sent from L1-2 during the migration generates miss in all the banks because the data is in cache but it's moving from a bank to another, and neither L2-8 nor L2-4 is able to satisfy the request (L2-8 doesn't have the copy of the block, L2-4 hasn't received yet the migrating block).

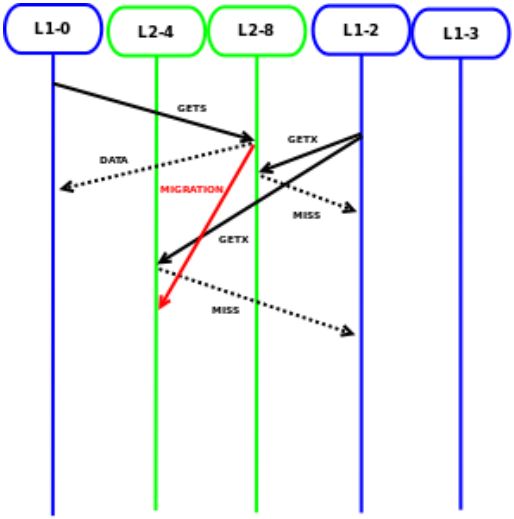

Figure 5.2: *The False Miss problem*

5.2 The Collector solution for multiple miss

The Collector is the bank of the bankset which manages all the off-chip accesses for a given physical address and constitutes the entry point for that address: for each address, only one bank in the bankset can act as a Collector, so that all the off-chip accesses pass necessarily through this particular bank; in this way we avoid the scenario described in Figure 5.1, in which the main memory receives two different requests for the same data and that data is sent to two different banks.

5.2.1 Basic assumptions

Collector's working bases on following hypothesis:

- only the Collector can send off-chip requests;
- the Collector is the block's entry point in L2 cache;
- when a processor sends a requests, it receives as many responses as many banks constitute a bankset; if we have N banks in a bankset, the responses can be:
  - N-1 MISS messages and a data message (Hit), or
  - N MISS messages (miss): the processor has to send an unicast request to the Collector;
- In case of Hit in a bank that's not a Collector for that address, the bank has to send an HIT message to the Collector, including the L1 requestor and the request type.

5.2.2 Operations

The collector has to keep trace of all the L1 requests broadcasted through the bankset, decide if the requests are solved with an L2 hit or with an L2 miss and, if an L2 miss occurs, send only one off-chip request even if multiple L1 requests are received: after the first one, all other L1 request don't have to be solved with a L2 miss while the collector is waiting for the data from the main memory.

L1 requests management. Let's consider a 4x4 DNUCA cache; if the data is not in L2 cache, an off-chip request has to be sent. Let's consider the protocol shown in Figure 5.3. L2-8 is the Collector within the bankset; when it receives a GET, it starts acting as a Collector, goint to a particular state in which it waits for one of the following messages:

- a second request of the same type coming from the same L1 requestor; it means that every bank of the bankset sent a MISS

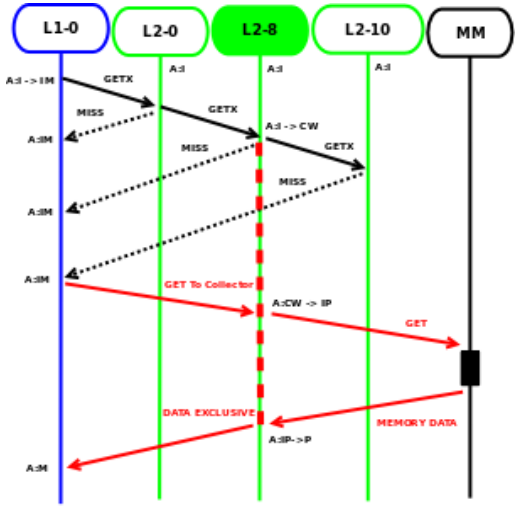

Figure 5.3: *Managing off-chip accesses due to an L2 miss through the Collector*

message to the L1 and an off-chip request has to be issued (see Figure 5.3);

- an HIT message coming from the bank that has a valid copy of the data: there's no need to issue an off-chip request so the Collector bank stops acting as a Collector and backs to an idle state (see Figure 5.4).

To let this protocol working, the L1 has to count how many MISS messages have been received, and, if the data message is received before all miss messages, the L1 can't replace the block until all the miss messages are received correctly. Due to the NoC topology and to routing policy, the Collector could receive the HIT message before the related GET message; however, as the Hit messages includes

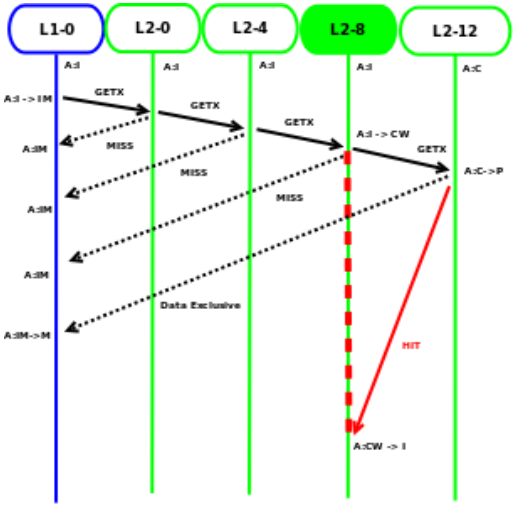

Figure 5.4: *The collector mechanism in case of Hit*

both the L1 requestor and the request type, the Collector can wait for the correct GET message: in fact, the MSHR mechanism avoid the L1 to issue multiple requests for the same block, so if the HIT is received before the GET, the Collector just waits for the request messages that is recognized thanks to the information carried by the HIT message.

Multiple requests management. Let's consider more than one processor requesting simultaneously the same data; as previously discussed, without a Collector, each request would result in an L2 MISS and issue an off-chip access. With the Collector, the only request which results in an off-chip request is the first request received by the Collector, while the subsequent messages will be held in the Collector's MSHR, and won't result in an off-chip miss.

Figure 5.5 shows the sequence of actions in case of multiple requests for the same memory block. As shown in figure 5.5, both L1-0 and L1-1 issue a request message for the same memory block. The request sent by the L1-0 is received for first and let the L2-8 bank start acting as a Collector: it waits for either the second GET or the HIT message.

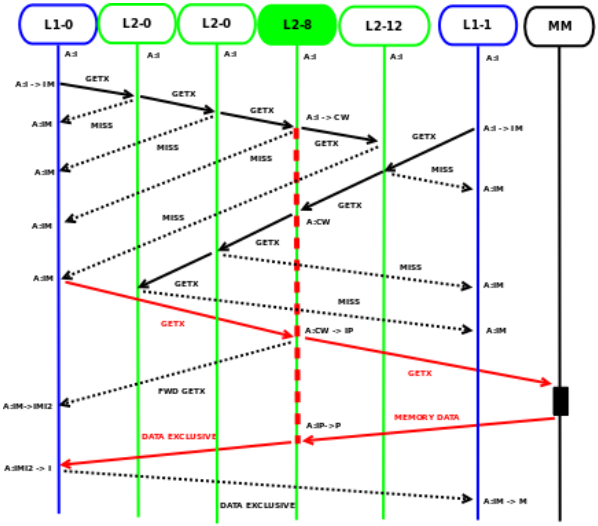

Figure 5.5: *Multiple request in case of an actual L2 miss*

As this is the case of actual L2 miss, the Collector will receive the second GET from the L1-0 (once the latter has collected all the MISS messages coming from the banks of the bankset). When the request sent by L1-1 is received, its' buffered in an internal queue; when the second L1-0 GET message is received the L2 miss is recognized, so the Collector sends an off-chip request and starts managing the buffered requests. In case of hit, if the GET request

coming from L1-1 is received by the Collector before any HIT message, the Collector has to wait both the HIT messages issued by the L2 bank that holds the block. Figure 5.6 shows this scenario; in order to simplify, we ignored the fact that both the hits in the L2-12 would result in a migration of the referred block.

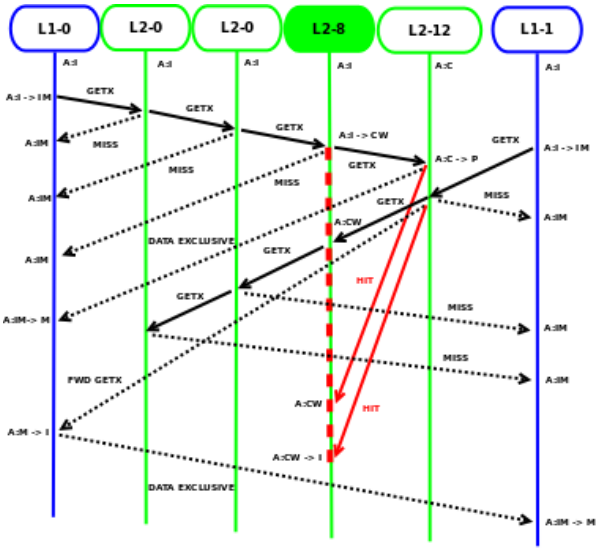

Figure 5.6: *Multiple requests and L2 HIT*

The second request (coming from L1-1) is buffered and not served until the second GET message (coming from L1-0) belonging to the first request arrives; when the HIT message is received, the L1-1's request is popped by the internal buffer and served: the corresponding MISS message is sent to L1-1, then the Collector waits for the second HIT message coming from the L2 bank that hold the block.

5.3 The FMA protocol to avoid the false miss

The FMA protocol (False Miss Avoidance) is a block migration protocol based on Migration Hiding: during migration the DNUCA keeps managing the requests and the migration process is transparent to the L1 caches. A preliminary version of the FMA protocol is proposed in [19].

5.3.1 Basic assumption

The FMA protocol is based on the following hypotesis:

- the communication is point-to-point ordered: if a node of the NoC sends one ore more messages to another node, the destination node receives that messages in the same order; this property is due to the deterministic routing policy of the considered NoC switches;

- each processor can send only one request at a time for a given block: if an L2 bank receives two requests for the same block from the same processors, this are related to the same transition (this is a consequence of the MSHR mechanism in the L1 caches);

- a block can migrate through adjacent banks of the same bankset;

- routing process in each switch of the NoC have to be performed for all the destination or for anyone: if the message in a switch has to be forwarded in more than one outport, but can't be forwarded through one or more of the outports, the routing process is delayed for all the outports.

5.3.2 Operations

A block can migrate from a bank to the next bank of the bankset towards the requesting processor. A migration occurs when there is hit in the L2 bank. The destination bank can reject a migration request or migration can be disabled in some cases. The migration of a block could cause the demotion of another block if all the ways in the destination L2 are already allocated to a cache line: one of the lines is chosen to be demoted towards the bank which started the migration process. In the banks at the edge of the DNUCA (i.e. banks in the first bank line with respect to the requesting processor), an Hit doesn't start a migration of the block.

Migration process. Let's consider the simple case in which the migration is accepted by the destination bank and there's no need to demote a block. This case is illustrate in Figure 5.7.

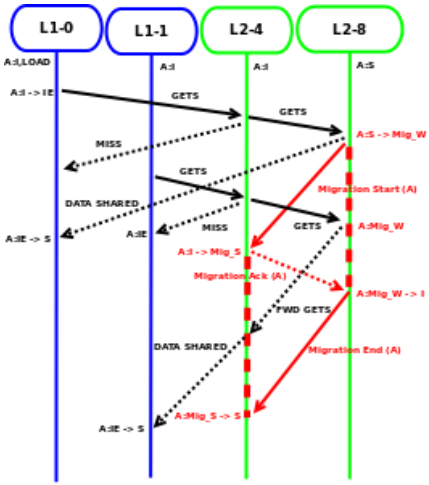

Figure 5.7: *Migration without demotion*

The request sent by L1-0 starts the migration process from L2-8 to L2-4. L2-8 send the requested data to L1-0 and stats an handshake with L2-4, that's nearer to the requesting processor. Basing on the FMA protocol, the L2 bank that stats the migration process behaves as follows:

- sends a migration request message (MIGRATION_START) to destination L2, including the data block and all directory information;
- waits for a MIGRATION_ACK message from destination L2: during this wait, all requests coming from other processors are forwarded to destination L2; this requests are served by the destination bank.
- Forwarding process ends as the MIGRATION_ACK message is received: subsequent requests are not forwarded (they will result in a MISS) and a MIGRATION_END message is sent to the destination bank to complete the handshake

When the MIGRATION_START message is received, the destination L2 bank acts as follows:

- allocates a line for the data block and serves all requests it receive, checking for duplicate requests;
- It waits for the MIGRATION_END message

Note that the forwarding process of all incoming requests (from the L2 that began the migration transaction) is effective in avoiding the false miss problem: in fact a false miss can occur when the referred block is migration and none of the banks is able to satisfy a new request; if the sending L2 waits for an acknowledgment before

deallocating the cache line, any new request is not ignored, but forwarded to the destination L2 bank. Once the acknowledgment is received, the sender L2 assumes that the destination bank is able to serve subsequent requests, so the line can be deallocated.

Managing duplicated requests. As a consequence of the forwarding process of the FMA protocol, it is possible that the destination L2 bank receives both the original request and the corresponding copy forwarded by the other L2 bank. This scenario is shown in Figure 5.8.

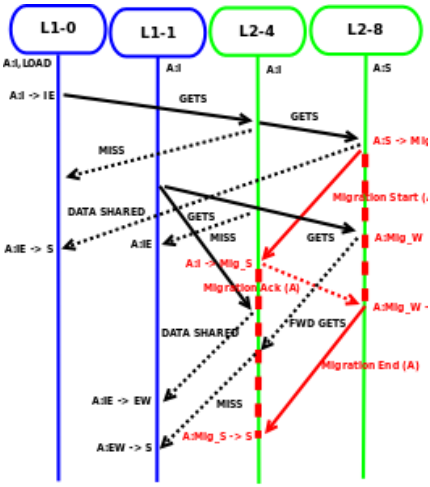

Figure 5.8: *Migration with duplicates management*

As we can see, L1-1 request reaches L2-4 when the migration process is in progress; L2-4 serves the request sending the data block to the L1, but after that it receives the same message forwarded by L2-8; the FMA protocol detects this duplicate requests in order to avoid a request to be served more than one time, keep-

ing trace of already served requests: if a GET is received after a MIGRATION_START, the following FWD_GETS will surely be a duplicate basing on hypothesis 2 of the FMA protocol. Note that only forwarded requests can be a duplicate: the miss message related to the L2 that started the migration process is sent by the destination L2 as the duplicate request is received, so the L1 can't send other requests until it receives all misses, included that one. For this reasons, new requests can't be a duplicate and they have to be served.

Demotion mechanism. If all ways of the set related to migrating block are already allocated, one of the conflicting lines is chosen though an LRU policy to be demoted from the bank destination of the migrating block to the one which sent the MIGRATION_START message. Figure 5.9 shows the demotion mechanism.

The migration of the A block from L2-15 to L2-11 determines the demotion of the B block from L2-11 to l2-15; the demotion process starts only when migration process is complete. As we can see, as the MIGRATION_START (A) message is received, a line is allocated to A in the destination L2 by deallocating the B block. B goes to a transient state in which all requests will be normally served because it's allocated in the MSHR. As the MIGRATION_END message is receive, the migration of the A block is complete and the demotion of B can start. Note that in the L2 that started the migration process the line that was allocated to A has to be allocated to B; for this reason, when the MIGRATION_ACK is received the line is not deallocated: it goes in a particular state in which keeps the location busy until the DEMOTION_START message for the block B is received. For this reason, the MIGRATION_ACK message has to specify that the migration of A is accepted but it causes the demotion of B: as a consequence, both the banks are aware of the fact

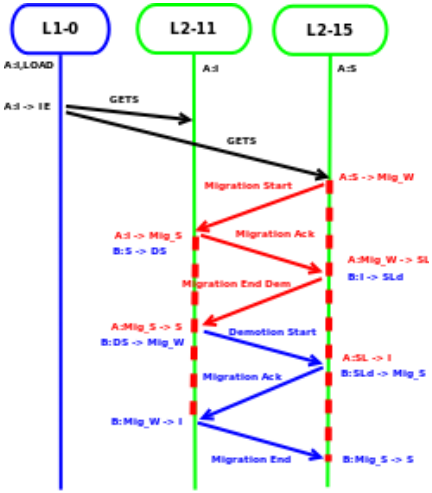

Figure 5.9: *Promotion and Demotion*

that the line B will be sent to L2-15 when the MIGRATION_END message reaches L2-11.

5.4 Results

For the D-NUCA evaluation, we considered an 8 CPUs CMP system, in two different topologies: 8p and 4+4p, similar to those showed in figure 4.1 for S-NUCA case. The shared cache is a NUCA composed by 64 banks, each of 256 KB and 4 way set-associative, for a total storage capacity of 16 MB. Table 5.1 summarizes the simulation parameters of our simulations. As running benchmarks, we considered some applications from the SPLASH-2 suite.

Dynamic block migration strives to reduce the NUCA cache hit

Number of CPUs	8
CPU type	UltraSparcII
Clock Frequency	5 GHz (16 FO4 @ 65 nm)
L2NUCA Cache	16 MB, 256 x 64KB, 16 ways s.a.
L1 cache	Private 32 Kbytes I + 32 Kbytes D, 2 way s.a., 3 cycles to TAG, 5 cycles to TAG+Data
L2 cache	16 Mbytes, 256 banks (64 Kbyte banks, 4 way s.a., 4 cycles to TAG, 6 cycles to TAG+Data
NoC configuration	Partial 2D Mesh Network; NoC switch latency: 1 cycle; NoC link latency: 1 cycle
Main Memory	2 GByte, 300 cycles latency

Table 5.1: *D-NUCA simulation parameters*

latency by moving frequently-accessed blocks. Figure 5.10 shows a comparison of the hit distribution of the considered SPLASH-2 applications; in particular, the hit distribution for D- NUCA 8p and 4+4p, and S-NUCA are reported. There is no difference in the hit distribution between S-NUCA 8p and 4+4p, due to the static mapping policy.

The reported hit distributions show that the migration mechanism succeeds in bring the most frequently accessed block in the banks characterized by the lowest latencies with respect to the referring processor. As a result, in the 8p configuration we can see that the most part of the 2 hits occur in the first row of banks (except for Ocean that uses both the first and the second line). In the 4+4p configuration, instead, the blocks migrate to the two sides

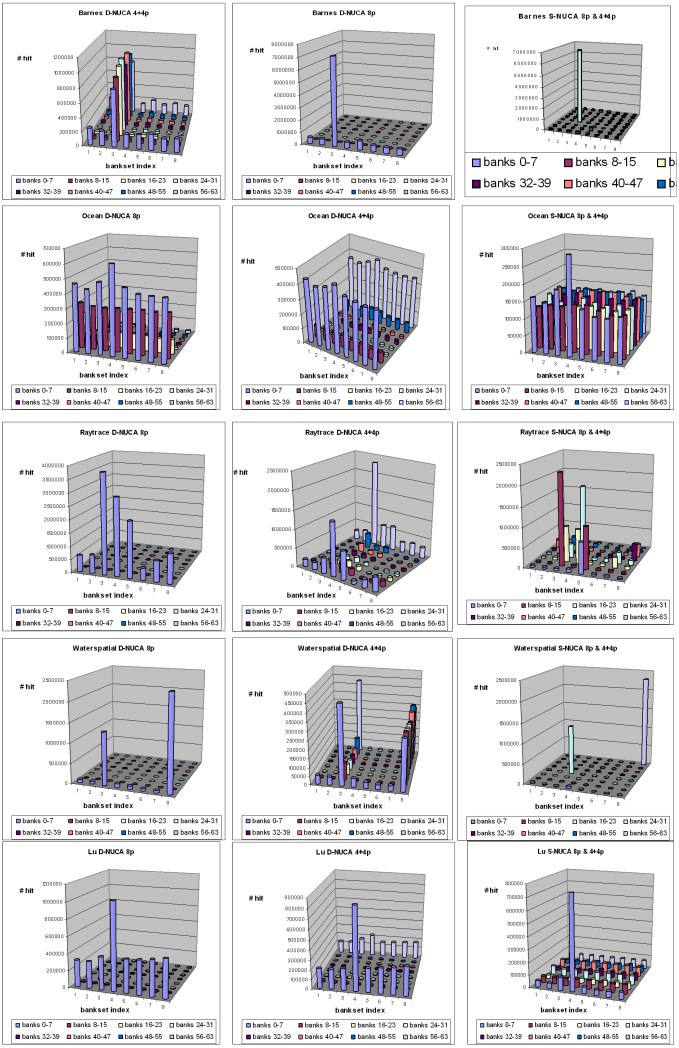

Figure 5.10: *Hit distribution for D-NUCA 8p, D-NUCA 4+4p and S-NUCA*

of the shared D-NUCA, as the processors are attached to two different sides; Barnes and Waterspatial can't succeed in bring such blocks near the referring CPUs as the shared blocks are accessed by threads running in processors placed at different sides, thus resulting in the conflict-hit phenomenon [6]. Again, Barnes and Waterspatial concentrate the most part of the hits in one bankset: this is a consequence of the mapping policy of the applications' data to the bankset (the same phenomenon can be observed for both the application in the S-NUCA distribution, in which the hits mostly occur in one bank Barnes or two banks Waterspatial). If we compare the D-NUCA and S-NUCA hit distribution, we can see that the distributed accesses to L2 banks in S-NUCA are avoided and concentrated in low-latency ways in the D- NUCA.

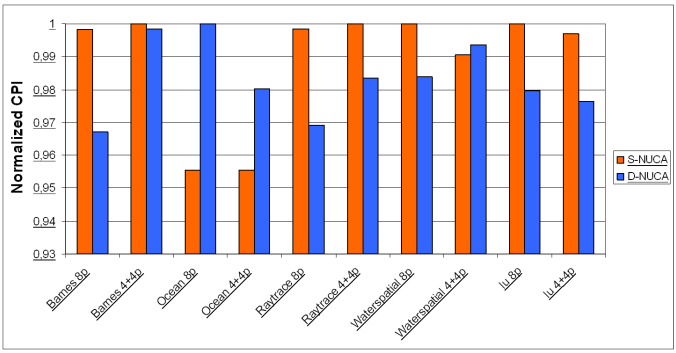

Figure 5.11: *Normalized CPI: S-NUCA vs D-NUCA, 8p vs 4+4p*

In order to evaluate the effectiveness of the adopted mechanisms with respect to a S-NUCA configuration, we considered the CPI for each configuration and considered application; the CPI is shown in figure 5.11. As Figure 5.11 demonstrates, for all the considered ap-

plications, excepts for Ocean, the migration mechanism introduces a little performance improvement with respect to the S-NUCA. However, such improvement is, in the best case, of about 4.5% with respect to the corresponding S-NUCA configuration. However, if we consider the L1 miss latency shown in Figure 5.12, in case of L2-to- L2 block transfer (i.e., this is the case of L1 miss requests that hit in the L2 cache, and the L2 cache directly provides the L1 requestor with the block, without the need of forward it to an L1 owner) we can observe a significant improvement in all the considered cases. Figure 5.12 demonstrates that the migration mechanism is effective in reducing the L2 hit time: for example, Barnes in the 8p configuration save about 30% of the time, on average, with respect to the corresponding S-NUCA system, and Waterspatial about 35% in the same case. However, both Barnes and Waterspatial present a very little improvement in L2 hit latency when the configuration is 4+4p, due to the conflict-hit phenomenon we discussed above.

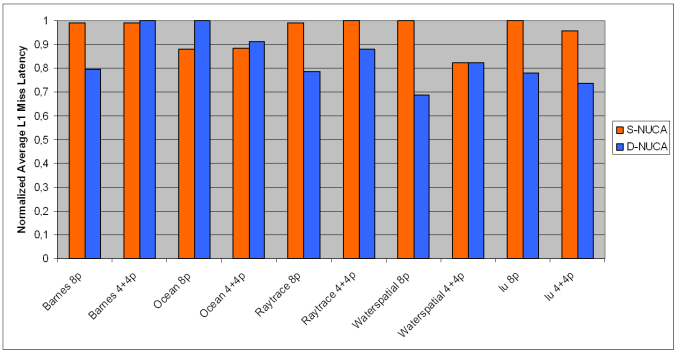

Figure 5.12: *Normalized L1 miss latency, in case of L2 hit with L2-to-L1 transfer*

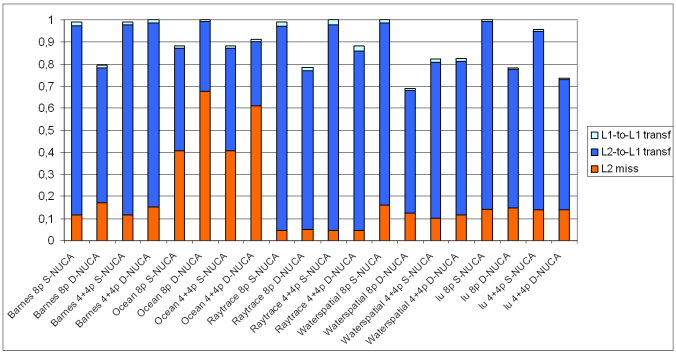

Figure 5.13: *Breakdown of Average L1 miss latency (Normalized)*

To better evaluate D-NUCA behaviors, we considered the total average L1 miss latency shown in figure 5.13. We observe an increase in the L2 miss component of the L1 miss latency; in particular, for Ocean the advantage of the L2-to-L1 latency reduction in the D-NUCA is invalidated by the higher L2 miss component. In order to understand this aspect, we show the L2 miss rate in figure 5.14.

Ocean doubles the L2 miss rate in the D-NUCA with respect to the S-NUCA, moving from about 1% of the S-NUCA to the 2% of the D-NUCA. This could be explained with an increase of conflicts that occur in the case of D-NUCA, as a consequence of the reduced number of banks that act as entry point (the Collector). In fact, in the case of S-NUCA each of the 64 banks acts as an entry point, and can replace its blocks when needed; in the case of D-NUCA, we just have one Collector for each of the 8 banksets, that have to manage the same amount of data, thus resulting in a higher number of conflict misses. Such effect in Ocean is due to the access pattern to the shared blocks: each thread works on a separated

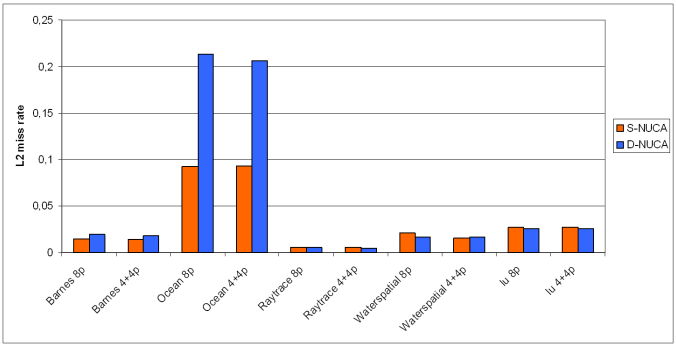

Figure 5.14: *L2 miss rate*

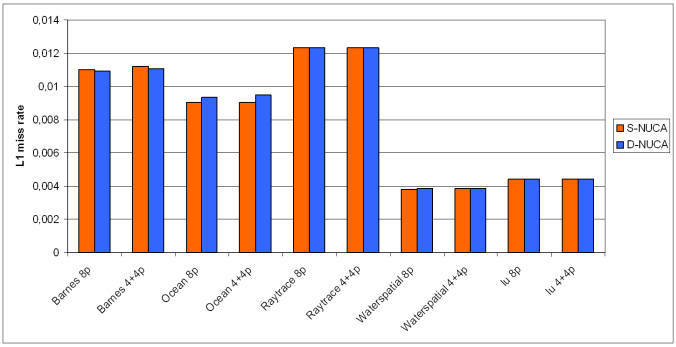

Figure 5.15: *L1 miss rate*

portion of the total shared space, thus making its blocks to compete with the others for the storage capacity in the Collectors. This problem has to be still faced, and will be the subject of my future research efforts. Due to the very low L1 miss rate (see figure 5.15), the high advantage of L1 miss latency reduction of D-NUCA with respect to the S-NUCA (about 15% on average, more than 30% in the best case) has not a great impact on performance (about 4.5% of performance improvement in the best case). Future works will also focus on a deeper evaluation of such aspect, in order to let the latency reduction benefits to have a greater impact on the overall performance.

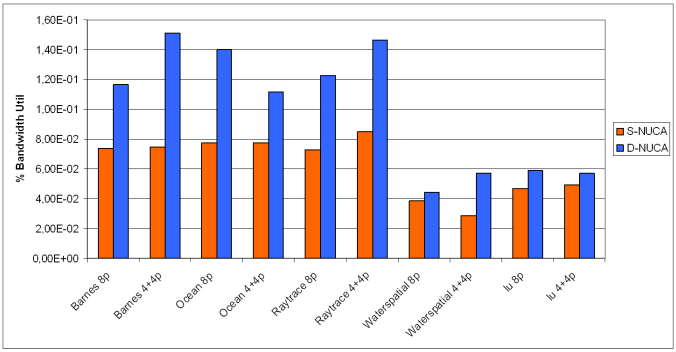

Figure 5.16: *Total NoC Link Bandwidth Utilization*

Another aspect that is interesting to evaluate is the NoC bandwidth utilization, that is shown in Figure 5.16. As one might expect, the percentage of the total bandwidth demand of D-NUCA is much higher than S-NUCA; in some cases, the NoC bandwidth occupancy doubles in D-NUCA with respect to S-NUCA (Ocean and Waterspatial). This is a consequence of the higher number of messages issued

in the NoC as a consequence of the migration mechanism, and due to the broadcast search policy adopted in our D-NUCA scheme. In Barnes, the D-NUCA in the 8p configuration is more bandwidth demanding than in the 4+4p configuration: in fact, as we can see in the accesses' distribution shown in Figure 5.10, in the 4+4p configuration Barnes doesn't succeed in bringing the most frequently accessed block near the CPUs, as a consequence of the access pattern to shared blocks; Raytrace and Waterspatial behave similarly to Barnes. Instead, Ocean reduces the D-NUCA bandwidth demand in the 4+4p case as the most accessed blocks have successfully migrated to low-latency NUCA banks. The bandwidth utilization is directly connected to dynamic energy consumption of the system. D-NUCA caches are expected to be more power consuming than S- NUCA. But the accesses' distribution suggests that some power-saving techniques can be adopted in CMP D-NUCA as in the case of uniprocessor system [23, 7, 21, 22].

Chapter 6

Power Consumption Model

Contents

6.1 Description

We defined a power consumption model to analyse the energy behaviour of cache memory systems, then we performed an analysis about the energy consumption in L2 NUCA cache using it. This

model can be applied to serveral memory architectures and it is cutomized on the different coherence protocols. We considered both dynamic and static NUCA architecture and we adapted the model to MESI and MOESI because the data search algorithm changes depending on the system we use. The study concerns static and dynamic energy and it has been realized through the combination of several simulation tools. We considered the static energy has two components: switch leakage and cache banks leakage. The dynamic energy instead concerns also the wires and the off-chip access consumption. Therefore we included switches, caches, wires and off-chip access energy in the evaluation of dynamic consumption. The scenario within we moved is a CMP system with 8 CPUs which share a NUCA 16 MB L2 cache, 16-way associative, divided in 256 banks with 64KB per bank (Figure 4.1). The switch used into our NoC is 8x8, 256 bits flit and contains both input and output buffer. We obtained the network traffic data using the Simics [50] full sistem simulator with the addition of the Gems [25] module. This tool permits to have vary detailed statistics which show the activity of each switch and the number of messagges travelling the network. Then to study the energetic features of the network switches we exploited the Orion [46] simulator we configured to emulate our network element. Both dynamic and static cache consumption has been estimated through CACTI 5.1 [11]; it is a tool which permits to obtain several parameters about cache consumption, latency and dimensions for each configuration and architecture of memory. Instead, to calculate the wires features we analysed the physical configuration of the entire system because we had to calculate exactly the wires lenght. Then we used the RC model and the PTM tool [49] to obtain the energetic values. At last we evaluated the off-chip consumption for RAM accesses. We got the values from the Micron datasheets [2]. We used our model to study the system using both MESI and

MOESI coherence protocols for static and dynamic NUCA architecture. The results show the main important component of the total energy consumption is always the static power, but we noted the dynamic component is not insignificant and it gets more importance decreasing the temperature. In details, we observed the static component is dominated by cache consumption, whereas the switch leakage is very low. In dynamic consumption the most important componets are the switches activity and the off-chip accesses for S-NUCA, whereas in D-NUCA systems also the contribution of cache accesses becomes important.

6.2 Tools

6.2.1 Simics and GEMS

It is a full system simulator. GEMS (General Execution-driven Multiprocessor Simulator) is a set of modules for Virtutech Simics that enables detailed simulation of multiprocessor systems, including Chip-Multiprocessors (CMPs). It has been developed by the Wisconsin Multifacet Project. Most Multifacet and external Publications use GEMS (http://www.cs.wisc.edu/gems/publications.html)

6.2.2 Orion

It is a power-performance simulator for interconnection networks, developed by Peh and Malik at Princeton University and built atop the Liberty Simulation Environment. It is cited in more than fifty papers and it is integrated in GEMS.

6.2.3 CACTI 5.1

It is a tool for modeling the dynamic power, access time, area, and leakage power of caches and other memories, developed by HP Advanced Architecture Laboratory. It is used in hundreds of Computer Architectures studies, e.g. Kim, Keckler and Burger in NUCA paper [34] (http://www.hpl.hp.com/techreports/2007/HPL-2007-167.html)

6.2.4 PTM

Predictive Tecnology Model (PTM) permits to obtain the features of transistor and interconnect technologies. It is useful to calculate the values of resistence and capacitance for different kinds of wires. It is developed by NIMO Group at Arizona University. With the previous generation of PTM, i.e., BPTM, more than 350 papers have been published by research teams all over the world. As an evolution of previous Berkeley Predictive Technology Model (BPTM), PTM will provide the following novel features for robust design exploration toward the 10nm regime:

- Predictions of various transistor structures, such as bulk, FinFET (double-gate) and ultra-thin-body SOI, for sub-45nm technology nodes.

- New methodology of prediction, which is more physical, scalable, and continuous over technology generations.

- Predictive models for emerging variability and reliability issues, such as NBTI.

6.3 Model

6.3.1 Static energy

$SimulationTime(ST) = \#ClockCycles * ClockPeriod$

$E_{static} = E_sSwitch + E_sCache$
$E_sSwitch = StaticSwitchEnergy/Cycle * \#ClockCycles$
$E_sCache = StaticCachePower/Second * ST$

Static Switch Energy/Cycle obtained from Orion
Static Cache Power/Second obtained from Cacti 5.1

6.3.2 Dynamic energy in D-NUCA cache

$R_{hit} = \#read$
$R_{miss} = 15 * \#read + 16 * \#miss$
$W = 16 * \#write + \#promotion + \#demotion$

- **ReadHit:** Each set is composed by 16 banks and it needs to access them all because the data can migrate.
- **Miss:** If the data is not present into the cache the search performs always sixteen accesses
- **Write:** To write a data it needs to know where the data is. So it performs a read.

$E_{dynamic} = E_dSwitch + E_dCache + E_dWire + E_dOff - Chip$
$E_dSwitch = \#ClockCycle * DynanicSwitchEnergy/Cycle$
$E_dCache = ReadHitEnergy/Op * R_{hit} + ReadMissEnergy/Op * R_{miss} + WriteEnergy/Op * W$
$E_dWire = DynamicWiresEnergy/Flit * \#Flits$

$E_dOff - Chip = RAMAccess\&BusEnergy/access * \#Off - ChipAccesses$

Dynanic Switch Energy/Cycle obtained from Orion
Read Hit Energy/op obtained from Cacti 5.1
Read Miss Energy/op obtained from Cacti 5.1
Write Energy/op obtained from Cacti 5.1
Dynamic Wires Energy/flit obtained from RC model and PTM
RAM Access & Bus Energy/access obtained from Micron datasheet

6.3.3 Dynamic energy in S-NUCA cache for MESI and MOESI coherence protocol

$R_{hit} = \#read$
$R_{miss} = \#miss$
$W = 16 * \#write$

$E_{dynamic} = E_dSwitch + E_dCache + E_dWire + E_dOff - Chip$
$E_dSwitch = \#ClockCycle * DynanicSwitchEnergy/Cycle$
$E_dCache = ReadHitEnergy/Op * R_{hit} + ReadMissEnergy/Op * R_{miss} + WriteEnergy/Op * W$
$E_dWire = DynamicWiresEnergy/Flit * \#Flits$
$E_dOff - Chip = RAMAccess\&BusEnergy/access * \#Off - ChipAccesses$

Dynanic Switch Energy/Cycle obtained from Orion
Read Hit Energy/op obtained from Cacti 5.1
Read Miss Energy/op obtained from Cacti 5.1
Write Energy/op obtained from Cacti 5.1
Dynamic Wires Energy/flit obtained from RC model and PTM

RAM Access & Bus Energy/access obtained from Micron datasheet

6.4 Results

We discuss the results obtained by applying the model to a 8 cores CMP systems with L2 S-NUCA shared 16MB cache where the coherence is managed with MESI and MOESI protocols. The banchmarks we used are Ocean and Barnes which are part of the Splash2 suite and we considered 800 milion of executed instructions for each run. Figure 6.1 shows the total energy consumption in a single

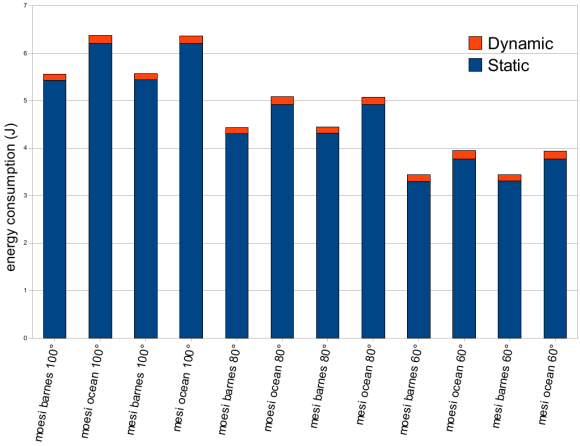

Figure 6.1: *Total energy consumption of S-NUCA cache memory in a system adopting MESI and MOESI protocols and running Ocean and Barnes benchmarks*

run for both the protocols and the benchmarks and for three different value of temperature, because varying the temperature the

consumption changes considerably. First we notice that for every architecture the most importat component is the leakage power (static energy) and his value always grows with the temperature. Instead, the dynamic consumption rappresent about five percent af total energy and it is quite constant. Then, we observe that the choice of coherence protocol doesn't influence the power consumption and the energy value for MESI and MOESI is the same for each configuration. Finally, the benchmark affects the results; as shown in Figure 6.4 the Barnes IPC is higher than the Ocean one, so to commit 800 milion of instructions the system need more cycles running Ocean respect to Barnes. The power consumption increases because the leakage component grows for every Ocean runs. In Figure 6.2 we show the staic consumption breakdown changing the benchmark, the protocol and the temperature. The most important component of leakage power comes from cache banks consumption, whereas the contribution of switches is very low, about five percent at 100Â°C As we have already observed before for total consumption, there aren't important differences still for leakege power using MESI and MOESI protocols. Instead, the choice of benchmark is significant because varying the execution time (Figure 6.4) both the components increase for Ocean which need higher time to complete. At last, the temperature influences the power consumption directly, i.e. when the system runs at higher temperature the energy grows. Figure 6.3 shows the dynamic consumption breakdown changing protocols and architecture. The temperature in this case isn't influential on energy. The dominant components are the switches activity and the consumption of off-chip accesses, whereas the energy spent to access the cache banks and to travers the network is quite negligible. The protocol does not affect the consumption, whereas the choice of benchmark does it. In fact, Ocean shows higher miss rate (Figure 6.4) which causes the increment of off-chip accesses and dynamic

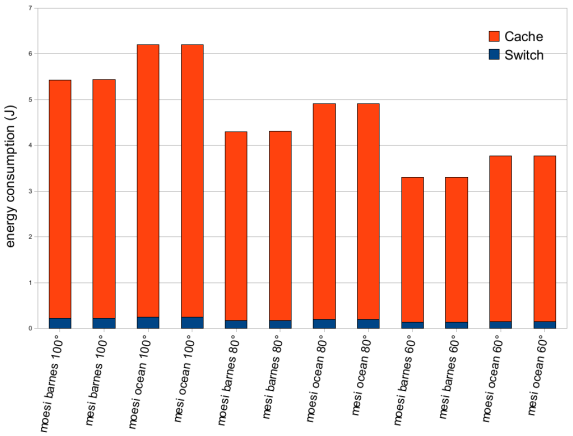

Figure 6.2: *Staic energy consumption of S-NUCA cache memory in a system adopting MESI and MOESI protocols and running Ocean and Barnes for different temperature, 100Â°C, 80Â°C and 60Â°C*

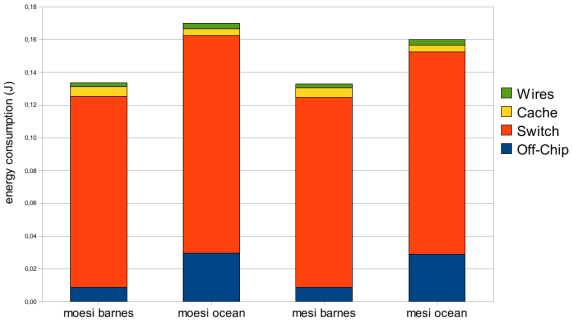

Figure 6.3: *Dynamic energy consumption of S-NUCA cache memory in a system adopting MESI and MOESI protocols and running Ocean and Barnes benchmarks*

energy consumption.

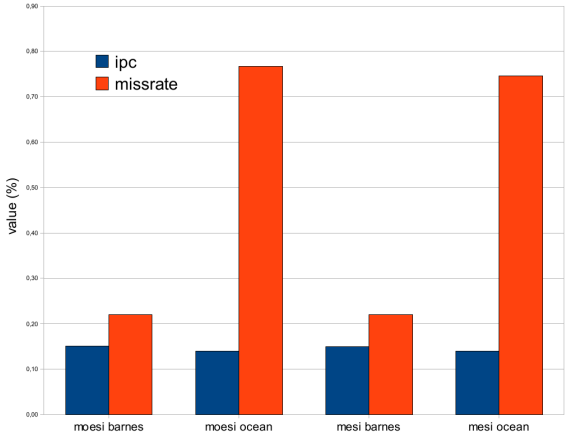

Figure 6.4: *IPC and miss rate of S-NUCA cache memory in a system adopting MESI and MOESI protocols and running Ocean and Barnes benchmarks*

After this analysis we applied our model to dynamic NUCA memory and we compared it to S-NUCA cache with MESI protocol we showed before. We studied two configurations 4.1 moving four cores on the other side of the memory and we performed an analysis on a 16MB L2 shared cache using the Barnes benchmark. The results for static energy consumption are similar to those we got for S-NUCA only, in fact, the leakage depends to simulation time. Instead, the dynamic component of power consumption behaves differently, as shown in figure 6.5. The total dynamic consumption is higher for D-NUCA than for S-NUCA and this is due to migration mechanism which increases the number of cache accesses and the traffic on the NoC. Then we observe that moving from 8p to 4+4p

configuration the switch component increases for both S-NUCA and D-NUCA because the NoC traffic grows. So, for D-NUCA architecture the switch component is dominant, but the cache accesses contribution becomes quite important.

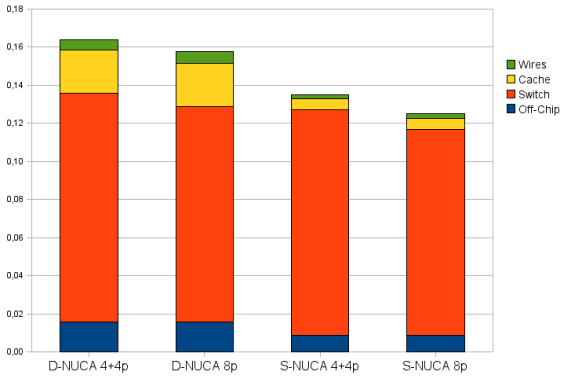

Figure 6.5: *Dynamic energy consumption of D-NUCA cache memory in a system running Barnes benchmark in 8p and 4+4p configuration and dynamic energy consuption of S-NUCA cache memory in a system adopting MESI protocol and running Barnes benchmark in 8p and 4+4p configuration*

Chapter 7

Conclusion and future works

Contents

7.1 Conclusions

In modern CMP system the cache memory implementation rapresents one of the most important issues to consider. In fact, the miss rate and the access time play a main role in performances increase and power consumption reduction. It is possible to adopt shared or private LLCs memory i CMP systems, but it is fundamental to optimize both latency and data rate. In order to improve CMP cache performance, this dissertation focused on the design of cache hierarchy adopting a Non-Uniform Cache Access (NUCA) architecture as the shared Last-Level-Cache of a CMP system. NUCA were proposed by Kim et al. as a solution to the wire-delay problem; studies performed so far demonstrate that NUCA caches are able to hide the wire-delay effects on the overall system performance. We showed three different analysis, the first about design tradeoff in S-NUCA

based CMP system, the second on latency reduction adopting D-NUCA system and the third on power consumption analysis of these architecture. We presented an evaluation for two different coherence strategy, MESI and MOESI, in a 8-cpus CMP system with a large shared S-NUCA cache where the topology vary across two different configurations (i.e. 8p and 4+4p). Our experiment show that CMP topology has a great influence on performances, instead the protocol has not. We also show that bandwidth utilization depends on the topology: this is a central design point, as bandwidth utilization is tired to dynamic energy consumption. Then we show our implementation of the migration mechanism in D-NUCA architecture realized avoiding the effects of "false miss" and "multiple misses" phenomena. We resolved the "multiple miss" using the Collector mechanism, which delegate just one bank of each bankset to be the manager for off-chip accesses. Instead, the "false miss" has been resolved adopting the FMA protocol, that guarantees that during migration there is at least one bank that knows that the block is actually on-chip. Our evaluation for D-NUCA architecture shows the migration mechanism is able to move the most accessed data toward the CPUs and it redeuces considerably the access latency to cache banks. Finally, we present a power consumption model which is adaptable to both S-NUCA and D-NUCA architecture. The results shows that the static energy is the dominant component of power consumption and the dynamic component rapresents a not negligible element which grows when we consider D-NUCA scheme.

7.2 Future works

The aim is to find an architecture which is mapping independent for general purpose applications and to exploit the different mappings strategy to increase performances in the case of specific applications,

however there are several directions we can exploit to improve our work.

- **Mapping strategy:** we want to implement a compiler level mapping strategy to optimize the distrubution of data inside the cache memory to increase the performances and to reduce the wire-delay effetcs.
- **Tiled architecture:** we work on a new tiled architecture where single tile is rappresented by a CMP system
- **D-NUCA insertion policy:** different insertion policies of memory blocks could be designed and evaluated in the D-NUCA scheme. In particular, such policies would aim to reduce the conflict probability that affect a class of application.
- **Way adaptable mechanism:** it si possible to adopt the way adapting technique [8] to CMP system in order to switch off the unused NUCA banks and to reduce both static and dynamic power consumption as we proposed in [23]

Bibliography

[1] International technology roadmap for semiconductors. semiconductor industrial association, 2005. 1, 56

[2] Micron datasheet. http://www.micron.com/. 96

[3] A. Bardine, M. Comparetti, P. Foglia, G. Gabrielli, and C. A. Prete. Nuca caches: Analysis of performance sensitivity to noc parameters. *Proc. of the Poster Session of the 4th Int.Â Summer School on Advanced Computer Architecture and Compilation for Embedded Systems*, 2008. 43

[4] A. Bardine, M. Comparetti, P. Foglia, G. Gabrielli, and C. A. Prete. On-chip networks: Impact on the performances of nuca caches. *Proceedings of the 11th EUROMICRO Conference on Digital System Design*, 2008. 43

[5] A. Bardine, M. Comparetti, P. Foglia, G. Gabrielli, and C. A. Prete. Performance sensitivity of nuca caches to on-chip network parameters. *Proceedings of the 20th International Symposium on Computer Architecture and High Performance Computing*, 2008. 43

[6] A Bardine, M Comparetti, P Foglia, G Gabrielli, and C A Prete. A power-efficient migration mechanism for d-nuca caches. *In Proceedings of the Design, Automation and Test in Europe 2009 (DATE 09)*, 2009. 26, 88

[7] A. Bardine, M. Comparetti, P. Foglia, G. Gabrielli, C. A. Prete, and P. Stenstrom. Leveraging data promotion for low power d-nuca caches. *In Proceedings of the 11th EUROMICRO Confer-*

ence on Digital System Design, September 2008,Parma, Italy. 93

[8] A Bardine, P Foglia, G Gabrielli, C A Prete, and P. Stenstrom. Improving power efficiency of d-nuca caches. *ACM SIGARCH Computer Architecture News*, 35:53–58, September 2007. 109

[9] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sanoand C. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocesing. *Proceedings of the 27th annual International Symposium on Computer Architecture, ISCA 00*, 28:282–293, 200. 10

[10] B.M. Beckmann and D.A. Wood. Managing wire delay in large chip-multiprocessor caches. *IEEE Micro*, Dec. 2004. 29, 56, 57

[11] Cacti 5.1: cache memory model. http://quid.hpl.hp.com:9082/cacti/. 58, 96

[12] J. Chang and G.S. Sohi. Cooperative caching for chip multiprocessors. *Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 264–276, 2006. 56

[13] Z. Chishti, M. Powell, and T.N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. *Proc. 36th Annual International Symposium on Microarchitecture (MICRO-36)*, pages 55–66, 2003. 20

[14] Z. Chishti, M.D. Powell, and T.N. Vijaykumar. Optimizing replication, communication, and capacity allocation in cmps. *Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 357–368, 2005. 26, 56, 57

[15] Dally and Towels. *Principles and Practices of Interconnection Networks*. Morgan Kauffmann,Elsevier, 2004. 6, 42, 43, 56, 58

[16] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks an Engineering Approach*. Morgan Kauffmann,Elsevier, 2003. 6, 42, 56, 58

[17] P. Foglia, G. Gabrielli, F. Panicucci, C. A. Prete, and M. Solinas. Reducing sensitivity to noc latency in nuca caches. *3rd Workshop on Interconnection Network Architectures: On-Chip, Multi-Chip (INA-OCMC'09)*, January 25, 2009. 43

[18] P. Foglia, D. Manganoa, and C.A. Prete. A nuca model for embedded systems cache design. *Proceedings of the 3rd Workshop on Embedded Systems for Real-Time Multimedia*, 2005. 24

[19] P. Foglia, F. Panicucci, C.A. Prete, and M. Solinas. Facing the false miss problem in d-nuca based cmp systems. *Proceedings of the Poster Session of the 4th International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES08)*, 1:99–102, 2008. 80

[20] P. Foglia, F. Panicucci, C.A. Prete, and M. Solinas. Investigating design trade-off in s-nuca baseb cmp systems. *In Proceedings of the Workshop on UNIQUE CHIPS and SYSTEMS (UCAS-5)*, Boston, 26 April 2009, to appear. 55

[21] P. Foglia, F. Panicucci, C.A. Prete, and M. Solinas. Techniques for reducing power consumption in cmp nuca caches. *In Proceedings of the ACACES 2007*, 1:5–8, July 2007. 93

[22] P. Foglia, F. Panicucci, C.A. Prete, and M. Solinas. Cmp l2 nuca cache energy consumption model. *Proceedings of the ACACES 2008*, 1:111–114, July 2008. 93

[23] P. Foglia, F. Panicucci, C.A. Prete, and M. Solinas. Cmp l2 nuca cache power consumption reduction technique. *Proceed-*

ings of IEEE Symposium on Low Power and High-Speed Chips (COOLChips XI), page 163, Yokohama, Japan, April 16-18 2008. 93, 109

[24] S.J. Frank. Tightly coupled multiprocessor system speeds memory access times. *Electronics*, 57(1):164–169, Jan. 1984. 5

[25] Winsconsin multifacet gems simulator. http://www.cs.wisc.edu/gems/. 58, 96

[26] K. Gharachorloo, M. Sharma, S. Steely, and S. Van Doren. Architecture and design of alphaserver gs320. *Proc. of the 9th int. conf. ASPLOS*, pages 13–24, 2000. 42, 43, 57

[27] R. Giorgi and C.A. Prete. Pscr: a coherence protocol for eliminating passive sharing in shared-bus shared-memory multiprocessors. *IEEE Transaction on Parallel and Distributed Systems*, 10(7):742–763, July 1999. 5

[28] S. Gochman, A. Mendelson, A. Naveh, and E. Rotem. Introduction to intel core duo processor architecture. *Intel Technology Journal*, 10(2):89–98, 2006. 13

[29] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and Olukotun K. The stanford hydra cmp. *IEEE Micro*, 20(2):71–84, 2000. 8

[30] L. Hammond, B.A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9), 1997. 56

[31] Mai Ho and Horowitz. The future of wires. *Proc. of the IEEE*, 89:490–504, 2001. 56

[32] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A nuca substrate for flexible cmp cache sharing. *Proc.*

of the 19th annual int. conf. on Supercomputing, pages 31–40, 2005. 25, 56, 57

[33] R.H. Katz, S.J. Eggers, D.A. Wood, C.L. Perkins, and R.G. Sheldon. Implementing a cache consistency protocol. *Proc. 12th Int'l Symp. Computer Architecture*, pages 276–283, June 1985. 5

[34] C. Kim, D. Burger, and S. W. Keckler. Nonuniform cache architectures for wire-delay dominated on-chip caches. *IEEE Micro*, Nov./Dec. 2003. 16, 26, 56, 58, 98

[35] C. Kim, D. Burger, and S.W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. *In Proceedings of the 10th international conference on Architectural support for Programming Languages and Operating System*, 2002. 26

[36] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005. 2, 56

[37] K. Krewell. Ultrasparc iv mirrors predecessors. Microprocessor report, November 1997. 1, 2, 56

[38] J. Laudon and D. Lenoski. The sgi origin:Â a ccnuma highly scalable server. *Proceedings of the 24th international symposium on Computer architecture*, pages 241–251, 1997. 6, 32, 37, 55

[39] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the

dash multiprocessor. *Proceedings of the 17th international symposium on Computer Architecture*, page 148, 1997. 6, 32, 36, 52, 55, 56

[40] Milo M. K. Martin, Mark D. Hill, and David A. Wood. Token coherence: decoupling performance and correctness. *In Proceedings of the 30th annual international symposium on Computer architecture*, 2003. 37

[41] E.M. McCreight. The dragon computer system: An early overview. *NATO Advanced Study Institute on Microarchitecture of VLSI Computer*, July 1984. 5

[42] C. McNairy and R. Bhatia. Montecito: A dual-core dual-thread itanium processor. *IEEE Micro*, 25(2):10–20, 1997. 1, 56

[43] Mendelson, Mandelblat, Gochman, Shemer, Chabukswar, Niemeyer, and Kumar. Cmp implementation in systems based on the intel core duo processor. *Intel Technology Journal*, 10, 2006. 2, 56

[44] 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*, 10(2):99–108, 2006. 13, 15

[45] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. pages 2–11, 1996. 56

[46] Orion: Power-performance simulator for interconnection networks. http://www.princeton.edu/ peh/orion.html. 96

[47] M. Papamarcos and J. Patel. A low overhead coherence solution for multiprocessors with private cache memories. *In Proceedings of the 11th International Symposium on Computer Architecture*, pages 348–354, June 1984. 5

[48] C.A. Prete. Rst cache memory design for a tightly coupled multiprocessor system. *IEEE Micro*, 11(2):16–19, 40–52, Apr. 1991. 5

[49] Predictive technology model. http://www.eas.asu.edu/ ptm/. 96

[50] Simics: full system simulation platform. http://www.simics.net/. 58, 96

[51] B. Sinharoy, R. Kalla, J. Tendler, R. Eickemeyer, and J. Joyner. Power5 system architecture. *IBM Journal of Research and Development*, 49(4), 2005. 1, 2, 56

[52] D.J. Sorin, M. Plakal, A.E. Condon, M.D. Hill, M.M.K. Martin, and David A. Wood. Specifying and verifying a broadcast and multicast snooping cache coherence protocol. *IEEE Transaction on Parallel and Distributed Systems*, 13(6):556–578, 2002. 42, 43

[53] P. Sweazey and A.J. Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. *In Proceedings of the 13rd International Symposium on Computer Architecture*, pages 414–423, June 1986. 5

[54] C. Thacker, L. Stewart, and E. Satterthwaite. Firefly: A multiprocessor workstation. *IEEE Trans. Computers*, 37(8):909–920, Aug. 1988. 5

[55] Woo, Ohara, Torrie, Singh, and Gupta. The splash-2 programs: characterization and methodological considerations. *roceedings of the 22th Internationl Symposium on Computer Architecture*, pages 24–36, 1995. 58

[56] M. Zhang and K. Asanovic. Victim replication:Â maximizing capacity while hiding wire delay in tiled chip multiprocessors. *Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 336–345, 2005. 56