Universitat Politècnica de València

Departamento de Informática
de Sistemas y Computadores

Smart Memory and Network Design for High-Performance Shared-Memory Chip Multiprocessors

A thesis submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy
(Computer Engineering)

Author

Mario Lodde

Advisor

Prof. José Flich

January 2014

A Stefania

Ringraziamenti / Agradecimientos

Vorrei prima di tutto ringraziare le persone che mi sono state vicine in questi quattro anni nonostante le centinaia di km che ci separavano: la mia famiglia in Sardegna, Stefania e tutti i miei amici sparsi tra la Sardegna e la Toscana (e qualcuno ultimamente anche un po' più lontano ;)). Questa tesi è il risultato anche del vostro continuo supporto e affetto.

Volviendo a este lado del Mediterráneo, quisiera agradecer en primer lugar a Pepe por haberme dado la oportunidad de hacer el doctorado en el GAP, por su constante seguimiento y soporte a mi trabajo y por su total disponibilidad para ayudarme con cualquier problema.

También quiero agradecer a todos los compañeros que han trabajado en los dos laboratorios del GAP en estos años, con que he compartido muchas horas de trabajo, comidas y viajes.

Y por último, pero no menos importante, quiero agradecer al resto del personal del DISCA por el gran ambiente de trabajo en que me he encontrado desde el primer día.

Abstract

The cache hierarchy and the Network-on-Chip (NoC) are two key components of chip multiprocessors (CMPs). Most of NoC traffic is due to messages exchanged by the caches according to the coherence protocol. The amount of traffic, the percentage of short and long messages and the traffic pattern in general depend on the cache geometry and the coherence protocol. NoC architecture and the cache hierarchy are indeed tightly coupled, and these two components should be designed and evaluated together to study how varying one's design affects the other one's performance. Furthermore, each component should adjust to match the requirements and exploit the performance of the other one, and vice versa. Usually, messages belonging to different classes are sent through different virtual networks or through NoCs with different bandwidth, thus separating short and long messages. However, other classification of the messages can be done, depending on the type of information they provide: some messages, like data requests, need fields to store information (block address, type of request, etc.); other messages, like acknowledgement messages (ACKs), do not need to specify any information except for the destination node. This second class of messages do no require high NoC bandwidth: latency is far more important, since the destination node is typically blocked waiting for their reception. In this thesis we propose a dedicated network which is able to transmit this second class of messages; the dedicated network is lightweight and fast, and is able to deliver ACKs in a few clock cycles. By reducing ACKs latency and the NoC traffic, it is possible to:

- speed-up the invalidation phase during write requests in a system which employs a directory-based coherence protocol
- improve the performance of a broadcast-based coherence protocol, reaching performance which is comparable to that of a directory-based protocol but without the additional area overhead due to the directory
- implement an efficient and dynamic mapping of cache blocks to the last-level cache banks, aiming to map blocks as close as possible to the cores which use them

The final goal is to obtain a co-design of the NoC and the cache hierarchy which minimizes the scalability problems due to coherence protocols. In this thesis we explore the different design alternatives for fast network delivery and coherence protocol opportunities. The best mechanisms, combined on a final system, allow for a truly dynamic

and customizable architecture in an environment with multiple applications demanding partitioning of resources.

Resumen

La jerarquía de caches y la red en el chip (NoC) son dos componentes clave de los chip multiprocesadores (CMPs). La mayoría del trafico en la NoC se debe a mensajes que las caches envían según lo que establece el protocolo de coherencia. La cantidad de tráfico, el porcentaje de mensajes cortos y largos y el patrón de tráfico en general varían dependiendo de la geometría de las caches y del protocolo de coherencia. La arquitectura de la NoC y la jerarquía de caches están de hecho firmemente acopladas, y estos dos componentes deben ser diseñados y evaluados conjuntamente para estudiar cómo el variar uno afecta a las prestaciones del otro. Además, cada componente debe ajustarse a los requisitos y a las oportunidades del otro, y al revés. Normalmente diferentes clases de mensajes se envían por diferentes redes virtuales o por NoCs con diferente ancho de banda, separando mensajes largos y cortos. Sin embargo, otra clasificación de los mensajes se puede hacer dependiendo del tipo de información que proveen: algunos mensajes, como las peticiones de datos, necesitan campos para almacenar información (dirección del bloque, tipo de petición, etc.); otros, como los mensajes de reconocimiento (ACK), no proporcionan ninguna información excepto por el ID del nodo destino. Esta segunda clase de mensaje no necesita de mucho ancho de banda: la laténcia es mucho mas importante, dado que el nodo destino está bloqueado esperando su recepción. En este trabajo de tesis se desarrolla una red dedicada para trasmitir la segunda clase de mensajes; la red es muy sencilla y rápida, y permite la entrega de los ACKs con una laténcia de pocos ciclos de reloj. Reduciendo la laténcia y el tráfico en la NoC debido a los ACKs, es posible:

- acelerar la fase de invalidación en fase de escritura en un sistema que usa un protocolo de coherencia basado en directorios
- mejorar las prestaciones de un protocolo de coherencia basado en broadcast, hasta llegar a prestaciones comparables con las de un protocolo de directorios pero sin el coste de área debido a la necesidad de almacenar el directorio
- implementar un mapeado dinámico de bloques a las caches de último nivel de forma eficiente, con el objetivo de acercar al máximo los bloques a los cores que los utilizan

El objetivo final es obtener un co-diseño de NoC y jerarquía de caches que minimice los problemas de escalabilidad de los protocolos de coherencia. En esta tesis se exploran diferentes alternativas para una entrega rapida de los ACKs y las oportunidades

que ofrece al protocolo de coherencia. Combinando los mecanismos presentados en un sistema final, se obtiene una arquitectura adaptable dinamicamente a los requisitos de múltiples aplicaciones en un entorno virtualizado.

Resum

La jerarquia de cache i la xarxa en el xip (NoC) són dos components clau dels xips multiprocessador (CMPs). La majoria del trànsit en la NoC es deu a missatges que les caches envien segons el que estableix el protocol de coherència. La quantitat de trànsit, el percentatge de missatges curts i llargs i el patró de trànsit en general varien depenent de la geometria de les caches i del protocol de coherència. L'arquitectura de la NoC i la jerarquia de cache estan de fet fermament acoblades, i aquests dos components han de ser dissenyats i avaluats conjuntament per a estudiar com al variar un afecta a les prestacions de l'altre. A més, cada component s'ha d'ajustar als requisits i a les oportunitats de l'altre, i a l'inrevés. Normalment diferents classes de missatges s'envien per diferents xarxes virtuals o per NoCs amb diferent ample de banda , separant missatges llargs i curts. No obstant això, una altra classificació dels missatges es pot fer depenent del tipus d'informació que proveeixen: alguns missatges, com les peticions de dades, necessiten camps per aemmagatzemar informació (adreça del bloc, tipus de petició, etc.). Altres, com els missatges de reconeixement (ACK), no proporcionen cap informació excepte per l'ID del node destí. Aquesta segona classe de missatge no necessita molt ample de banda: la latència és molt més important, atès que el node destí està típicament bloquejat esperant la recepció d'ells. En aquest treball de tesi es desenvolupa una xarxa dedicada per a transmetre la segona classe de missatges, la xarxa és molt senzilla i ràpida, i permet el lliurament dels ACKs amb una latència de pocs cicles de rellotge. Reduint la latència i el trànsit en la NoC a causa dels ACKs, és possible:

- accelerar la fase d'invalidació en fase d'escriptura en un sistema que utilitza un protocol de coherència basat en directoris
- millorar les prestacions d'un protocol de coherència basat en broadcast , fins a aplegar a prestacions comparables amb les d'un protocol de directoris però sense el cost d'àrea a causa de la necessitat d'emmagatzemar el directori
- implementar un mapejat dinàmic de blocs a les caches d'ùltim nivell de forma eficient , amb l'objectiu d'apropar quant al màxim possible els blocs als cores que els utilitzen

L'objectiu final és obtenir un co- disseny de NoC i jerarquia de cache que minimitze els problemes d'escalabilitat dels protocols de coherència. En aquesta tesi s'exploren diferents alternatives per un lliurament ràpid dels ACKs i les oportunitats que això ofereix

al protocol de coherència. Combinant els mecanismes presentats en un sistema final, s'obté una arquitectura adaptable dinàmicament als requisits de múltiples aplicacions en un entorn virtualitzat.

Contents

List of Figures

List of Tables

Abbreviations and Acronyms

bLBDR	**b**roadcast **L**ogic-**B**ased **D**istributed **R**outing
BN	**B**roadcast **N**etwork
CMP	**C**hip **M**ulti**P**rocessor
CS	**C**ache **S**tealing
D-NUCA	**D**ynamic **NUCA**
flit	**fl**ow control dig**it**
FSM	**F**inite **S**tate **M**achine
GN	**G**ather **N**etwork
HP	**H**ome **P**artition
LBDR	**L**ogic-**B**ased **D**istributed **R**outing
LLC	**L**ast-**L**evel **C**ache
MC	**M**emory **C**ontroller
MPSoC	**M**ulti**P**rocessor **S**ystem **o**n **C**hip
MSHR	**M**iss **S**tatus **H**olding **R**egister
NI	**N**etwork **I**nterface
NoC	**N**etwork-**o**n-**C**hip
NUCA	**N**on-**U**niform **C**ache **A**ccess
pNC	**p**artitioned **N**oC and **C**ache hierarchy
phit	**ph** isical dig**it**
PP	**P**rocessor **P**artition
PTA	**P**arallel **T**ag **A**ccess
RHM	**R**untime **H**ome **M**apping
RTT	**R**ound **T**rip **T**ime
SAF	**S**tore **A**nd **F**orward
S-NUCA	**S**tatic **NUCA**

SR	**S**egment-based **R**outing
SWMR	**S**ingle-**W**riter, **M**ultiple-**R**eaders,
VC	**V**irtual **C**hannel
VCT	**V**irtual **C**ut-**T**hrough

Chapter 1

Introduction

Current manufacturing technologies integrate billions of transistors in a single chip, enabling the production of Chip Multiprocessors (CMPs) with tens of cores and other components (e.g. accelerators, memory controllers, etc.) connected through a high-speed on-chip network (NoC) [1], [2]. Even today, there are several examples of this kind of systems, including prototypes (such as the 48-core Single-Chip Cloud Computer developed by Intel [3]) and real products (such as Tilera's 100-core TILE-Gx100 processor [4] or Kalray's 256-core MPPA 256 [5]).

The trend in CMP systems with a very high core count is to employ simple cores, possibly in-order, since they provide a better performance/power ratio than more complex and larger out-of-order cores. To further reduce the design complexity and provide higher scalability, these systems are usually organized in tiles, modular units which include one or more cores, one or more level of caches and a switch to connect each tile with its neighbors through the network-on-chip (NoC), typically a 2D mesh. Tile-based design allows to focus design efforts on a single tile, which is then replicated to build the final system. These tiled architectures have been claimed to provide a scalable solution for managing the design complexity, and effectively using the resources available in advanced VLSI technologies.

While systems with a limited number of communicating components can rely on a shared interconnect like a bus, as the number of cores and/or other components increases a shared interconnect is likely to become the system bottleneck, so an on-chip network must be implemented to satisfy the higher bandwidth demand. Systems with tens or

hundred of cores are typically connected through direct topologies like a 2D mesh or a concentrated 2D mesh; these topologies are easily scalable and their implementation in a tiled architecture is straightforward.

Parallel applications for multi-core systems can be developed using two different programming models, depending on how the cores communicate between them. On the one hand a message passing model can be used, where cores communicate by exchanging explicit messages; on the other hand, with a shared memory model the cores communicate through shared variables. The shared memory programming model is usually preferred, being easier for programmers as communication is implicit when accessing variables. However, with multiple actors operating on the same memory locations, a set of rules must be defined to determine the order of memory operations that leave the memory in a consistent state; this set of rules is known as Memory Consistency Model [6].

The implementation of a system based on shared memory is further complicated by the presence of cache memories. Cache memories play an important role in a CMP system since they mask and limit the effect of the problem commonly know as the memory wall: the speed of processors is growing faster than the speed of memories, thus making memories the bottleneck of the entire system. Caches are able to strongly reduce the number of off-chip accesses to main memory, resolving the vast majority of data accesses within the chip. The on-chip cache is organized hierarchically, with smaller and faster caches at the higher levels close to the processor and larger but slower caches at the lower levels. The last level is usually banked to overcome the wire delay problem [7].

Since the same data can be replicated in multiple private caches, a cache coherence protocol must be defined to regulate the read and write accesses to cached data and avoid data incoherence due to the wrong management of write operations: any write operation must be indeed eventually visible to all cores and all cores must see the same order of write operations to the same data. Many different coherence protocols have been proposed in the last decades, but the rise of CMP systems introduces new challenges: existing protocols indeed present scalability issues that must be solved to allow their use in a many-core system, so a lot of current research effort is focused on improving the scalability of traditional protocols to make them fit to be used in future many-core systems.

The implementation of coherence protocols in a many-core CMP is further complicated when the protocol is mapped on top of a point-to-point network: some properties of a bus which simplify the implementation of a coherence protocol and limit the number of possible race conditions, such as the total ordering of coherence messages and the inherent broadcast nature, are lost in more sophisticated networks; a regular 2D mesh, for instance, only provides point-to-point message ordering and the coherence protocol must thus be designed keeping in mind this assumption and considering the race conditions that may arise when multiple components (L1 caches, L2 caches, memory controllers, etc.) communicate with the same node simultaneously. Unlike a bus, a 2D mesh delivers broadcast and multicast messages in different times to different nodes, and there is no way to determine whether a node has already received a message or not, so additional control messages (acknowledgement messages) must be used to ensure the correct timing of coherence operations.

Most of the traffic (if not all) on the on-chip network of a CMP system is due to coherence messages exchanged by the various levels of caches and the memory controller according to the cache coherence protocol, so on the one hand the coherence protocol determines the nature of the NoC traffic (percentage of long and short messages, percentage of unicast and broadcast messages, etc.) and on the other hand the NoC determines the overall system speedup that the coherence protocol and the cache hierarchy can achieve. These two components must then be co-designed, tailoring the NoC to meet the requirements of the coherence protocol messages in terms of bandwidth and latency and adapting the coherence protocol to efficiently use the resources provided by the NoC.

Finally, with the core count reaching hundreds of nodes, resource partitioning will be a key feature of future CMP systems. A partitioning scheme to dynamically and independently assign resources to multiple applications which are executed concurrently on the CMP will be essential to efficiently exploit CMP resources and maximize the performance. Some current virtualized systems (e.g. Tilera CMPs) allow to partition the system at the tile-level and define regions in the chip which are assigned to different applications. However, it should be possible to independently partition the CMP components, defining different partitions for different types of resources (e.g. the cores or the last-level cache) to meet the specific needs of each application, which requires additional engineering of the on-chip network and the last-level cache.

1.1 Thesis Contributions

This thesis focuses on the co-design of the NoC and the cache coherence protocol to speedup some control phases of the coherence protocols and improve their scalability. Starting from the baseline CMP with standard NoC and standard cache hierarchy shown in Figure 1.1, different optimizations are proposed at NoC- and cache-level. The first one (Figure 1.2-A) is a dedicated control network to transmit acknowledgement messages; acknowledgements represent indeed a special class of messages which deliver a very limited amount of information, and can be successfully transmitted through a lightweight dedicated network, thus relieving the regular NoC from a percentage of traffic that can be considerable in some classes of coherence protocols. The dedicated network, called the Gather Network, is able to deliver many-to-one acknowledgements to the destination node in a few clock cycles, and it allows to use cache coherence protocols which make an extensive use of acknowledgements even in CMPs with a high core count limiting the traffic penalty on the NoC. It is furthermore used as NoC substrate to enable the second optimization, which is a cache-level technique aiming to reduce the LLC access latency by dynamically mapping cache blocks to the LLC banks (Figure 1.2-B). This cache-level technique, called Runtime Home Mapping (RHM), can also be used to implement the third optimization, a partitioning scheme of CMP resources where cores, cache resources and NoC resources are independently partitioned, as shown in Figure 1.3. Another optimization, orthogonal to previous ones, relies on an L2 bank with an heterogenous entry structure to reduce LLC area and power requirements (Figure 1.2-C).

More in detail, the contributions of this thesis are the following:

- The *Gather Network*, a dedicated control network to transmit many-to-one acknowledgements. It is proposed in two different ways: the first one is fully combinational, has a very low area overhead and a very low latency but may present scalability issues for systems with hundreds of cores; the second one employs dedicated sequential modules at the switches and solves the scalability issues of the combinational implementation at the expense of an increased latency. An extension of the Gather Network, called the *Network of IDs*, is not limited to many-to-one acknowledgements but can be used to transmit unicast acknowledgements as well.

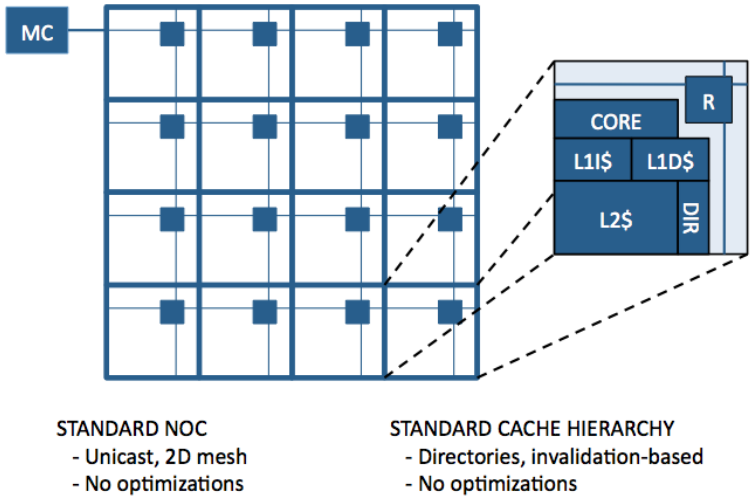


FIGURE 1.1: Baseline CMP system.

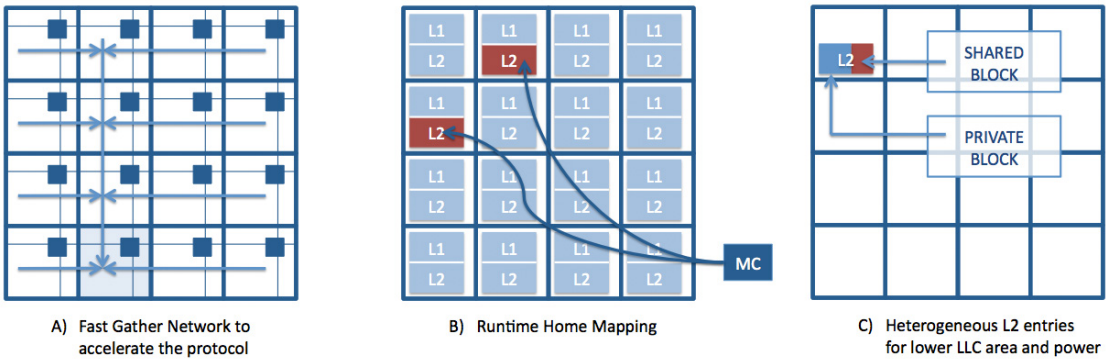


FIGURE 1.2: Contributions of this thesis.

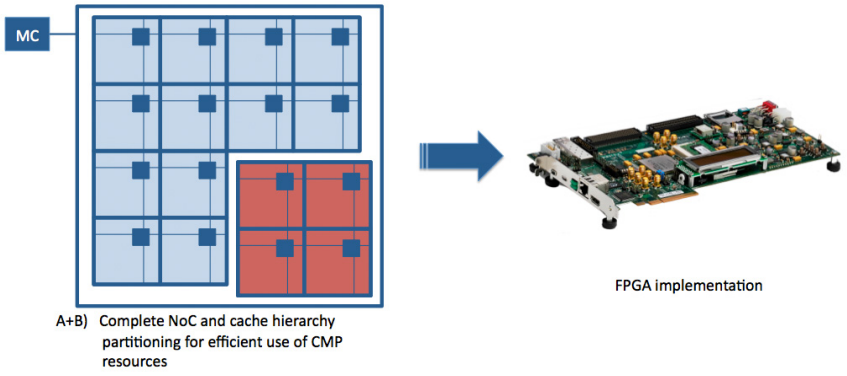


FIGURE 1.3: Final CMP system.

- *Runtime Home Mapping*, a dynamic technique where cache blocks are mapped to the LLC banks at runtime; the aim of RHM is to map each block as close as possible to the core(s) which access it, thus reducing the average LLC access

latency, balancing at the same time the utilization of all the LLC banks, thus exploiting the full on-chip cache capacity.

- The Partitioned NoC-Coherence (pNC), a scheme to independently partition cores, last-level caches and the NoC in a tiled CMP system; it is based on the combination of RHM at the cache level and the previously proposed Logic-Based Distributed Routing (LBDR) [8] at the NoC level.

- A novel cache bank organization with a different entry structure for private and shared blocks; private entries do not include the data field, and blocks are dynamically moved from an entry to another depending on the block state; this organization allows to reduce the cache bank's area and power with a minimum impact on system performance.

- The implementation of the final system's coherence protocol and cache hierarchy in an FPGA board. We have ported the baseline CMP system and the different optimization techniques and coherence protocols into a Virtex 7 evaluation board FPGA. This allows to demonstrate feasibility of the designs on a pre-competitive CMP system product.

With these optimizations and strategies, the CMP system is able to achieve truly partitioning at the NoC and coherence level while saving large percentages of power consumption and keeping maximum performance. The proposed solutions allow the coherence protocol to scale and the on-chip cache hierarchy to be used efficiently and to adapt to the requirements of the application executed by the system. Figure 1.3 shows the system that can be implemented using the contributions of this thesis: a virtualized CMP where the NoC and the LLC layer can be partitioned independently by the core partitioning to meet the requirements of the applications executed in the different regions.

1.2 Thesis Outline

This thesis structures in 7 chapters and 2 appendices. The first chapter is for the motivation of the thesis. Chapter 2 deals with background and related work. This chapter focuses both, in coherence protocols, and NoC architecture. Then, it follows 4 technical

chapters, each presenting a different contribution of the thesis. Then, Chapter 7 concludes the thesis and sets the future directions and describes the scientific contributions out of this thesis. Finally, two appendices describe the coherence protocols of the thesis and the FPGA implementation.

Regarding the technical chapters:

- Chapter 3 describes the Gather Network and illustrates its extension to the Network of IDs.
- Chapter 4 describes RHM and its optimizations to allow block migration, block replication and the particular optimizations that take advantage of a broadcast-based coherence protocol.
- Chapter 5 describes how RHM and LBDR can be integrated to achieve an independent partitioning scheme of the last-level cache and the NoC.
- Chapter 6 describes the heterogeneous structure of the last-level cache bank where different types of entries are used to store private and shared blocks.

Chapter 2

Background and Related Work

This chapter sets the background required for this thesis. It also provides an overview on the state of the art of the two components of a CMP system on which this thesis focuses: the cache hierarchy and the network-on-chip. Both, background and related work, are mixed in this chapter, providing a single description. First, we focus on cache memories and coherence protocols, and then on on-chip networks.

2.1 The Cache Hierarchy

Chip multiprocessor systems (CMPs) may employ a shared memory programming model, thus requiring a cache coherence protocol to keep data coherent along the cache hierarchy. To provide high cache capacity and low access time, the on-chip cache is organized hierarchically, with small, fast caches at the higher levels (closer to the processor) and bigger, slower caches at lower levels (closer to main memory). This provides high on-chip storage capacity without the high access latency a single, large cache would have. To overcome the wire-delay problem [7], the last-level cache (LLC) in CMP systems is usually banked; this configuration is commonly known as Non-Uniform Cache Access architecture (NUCA), initially proposed by Kim et al. for a single core system [9] and then extended to CMPs [10], [11]. While the higher levels of the cache hierarchy are always private to a core, different policies can be implemented as far as the LLC is concerned, but the common choices are two. On the one hand, an LLC bank can also be private to a core, thus extending that core's private cache capacity. This is the best option if the working set of the application which is running in the CMP fits in the LLC bank, since all cached data can be accessed without sending requests over the NoC. However, if the working set does not fit in the LLC bank, this policy generates many LLC line replacements, and therefore off-chip requests. Furthermore, if a set of cores share a cache block, a copy of that block must be present in each private cache, so shared blocks are replicated at different LLC banks.

On the other hand, each LLC bank can be a slice of the shared distributed LLC; shared LLCs are usually preferred because, although this organization has higher access latencies than private LLCs, it provides higher cache capacity, thus avoiding expensive off chip accesses that are more frequent when private LLCs are used. The access latency is variable, depending on the location of both the core and the LLC bank in the CMP, and on the strategy used to map memory blocks to the LLC banks, which will be discussed in Section 2.1.2.

Without any loss in generality, in this dissertation we assume a cache hierarchy organized in two levels, with inclusive[1] L2 caches.

[1] A level of the cache hierarchy is called inclusive if it stores all the data cached in the higher levels.

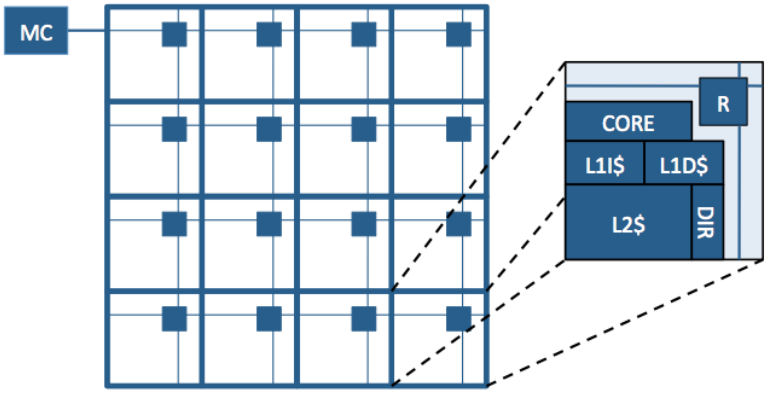


FIGURE 2.1: Baseline tile-based CMP system.

Figure 2.1 shows the reference system we assume throughout this work, unless stated otherwise: it includes 16 tiles organized in a 4×4 2D mesh, each tile including a core, a first-level private cache and a bank of shared LLC. This system will be also extended to a 8×8 configuration, with the same tile organization, in some parts of this thesis.

2.1.1 Cache Coherence Protocols

A cache coherence protocol has the task to keep the data coherent among caches private to different cores and located at the various levels of the cache hierarchy, enforcing the Single-Writer, Multiple Readers coherence invariant (SWMR; for any block, at any given time, one single core with write permissions or one or more cores with read only permissions have a copy of the block) and propagating the new value of a written data according to the memory consistency model.[2]

2.1.1.1 Invalidation-based vs Update-based Protocols

By managing write operations and the correct propagation of the new values, an important design choice is made when determining how the protocol behaves upon a write request on a shared block. Update-based protocols allow a core to write on a shared block, but the new value must be propagated to the copies in the L1 caches of the sharers. This family of protocols has the advantage of immediately providing the new value to the cores which share the block, but the update operation can be difficult to implement

[2]In this dissertation we assume Sequential Consistency; Lamport [12] defines a multiprocessor system sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

and may require a high amount of bandwidth. Furthermore, if the system implements a strict memory consistency model (e.g. sequential consistency) particular care must be taken to perform the update phase without violating the consistency model; this is simpler if the system employs a shared interconnect as a bus but becomes much more complicated with point-to-point topologies such as the 2D mesh.

Invalidation-based protocols invalidate all the shared copies of a block before a write operation is performed. This way, if a core wants to read that block it must issue a new coherence request, thus ensuring that any core will always read the value produced by the last write operation. The performance penalty of invalidation-based protocols is due to the fact that the sharers may still need to access the block which is being invalidated and to the consequent high write miss latency: in case of a write miss, an L1 cache sends a write request to the L2 cache, which in turn sends an invalidation message to each L1 cache holding a copy of the block. The requestor cannot write until it receives the requested data block and all the acknowledgements to the invalidation message.

Hybrid protocols have also been proposed, which behave as invalidation- or update-based depending on the access pattern to each block [13]. These protocols try to reduce the high traffic generated by a pure update-based protocol by determining at runtime which cache blocks should be updated when modified and which ones should be invalidated. However, due to the drawbacks of update-based protocols, almost all current systems use pure invalidation-based protocols, which we will also assume in this thesis.

2.1.1.2 Steady States at L1 Cache Controllers

The typical way of defining a cache coherence protocol is through a finite state machine (FSM) which indicates the evolution of the state of a cache block depending on the access and coherence actions performed to it. One of the design choices of a coherence protocol is the number of steady states the blocks can have in L1 caches; typically the properties of each cache block are encoded using the five states proposed by Sweazey and Smith [14]; focusing on a private L1, the state of a cache block can be one of the following:

- M (Modified): only this L1 cache has a copy of the block with read and write permissions; the copy has been modified and the copy in the L2 cache is thus stale

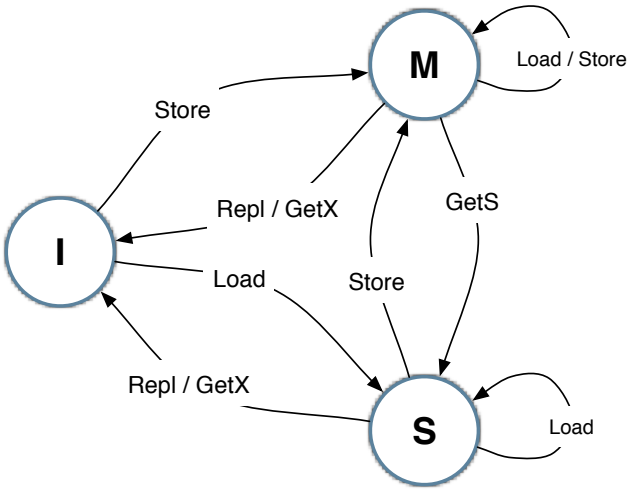


FIGURE 2.2: Simplified FSM for the MSI protocol.

- O (Owned): this L1 cache has a copy of the block with read-only permission and must provide the block when it is requested by other L1s; other L1 caches may have a read-only copy of the block in state S; the copy in the L2 cache may be stale
- E (Exclusive): only this L1 cache has a copy of the block with read and write permissions; the copy has not been modified
- S (Shared): this L1 cache has a read-only copy of the block; other L1 caches may also have a read-only copy
- I (Invalid): the block is either not present in this cache or it is present but not valid

The three states M, S and I are the basic ones and allow to define the simpler MSI protocol, while states O and E are two optimizations which can be used to extend the MSI protocol, thus obtaining MOSI, MESI and MOESI protocols. An L1 cache which has a copy of a block in state M, O or E is referred to as the *owner* of that block, while the set of L1 caches holding a copy of a block in state S are referred to as sharers of that block.

Figure 2.2 shows a simplified FSM of the MSI protocol (transient states are not shown). A cache block is initially in state I; then, when the core issues a read (Load) or write (Store) request, the line goes to state S or M, respectively. The block is requested by sending respectively a GetS or a GetX request through the NoC; the requested data is

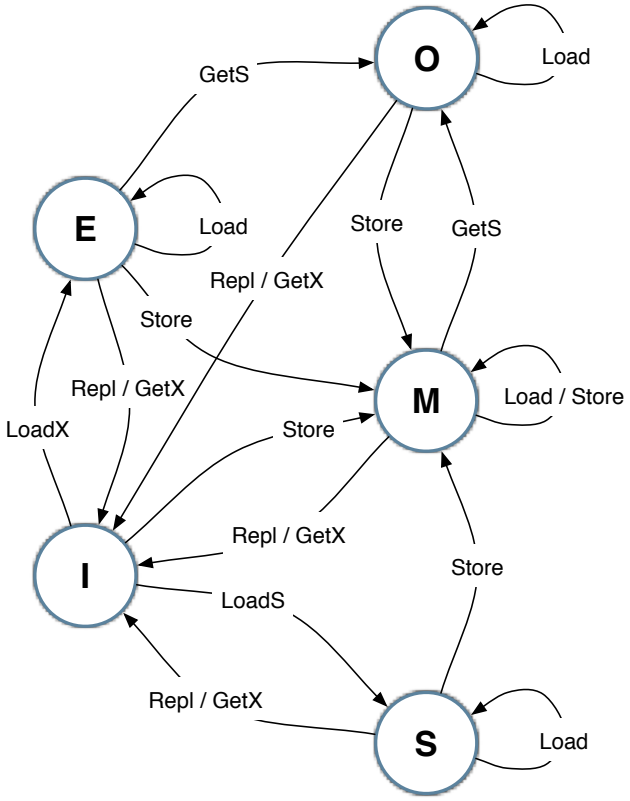


FIGURE 2.3: Simplified FSM for the MOESI protocol.

provided by the LLC or by the owner L1 depending on the state of the block. If the cache line is replaced (Repl), the state switches back to I. The block is also invalidated if the cache receives a write request issued by another core (GetX), to preserve the SWMR invariant. If a write request from the local core is received while the line is in state S, it switches to state M; to do so, the L1 cache must issue a coherence request to obtain write permissions (GetX), and the write operation can be completed only after the copies hold by eventual sharers have been invalidated. If the cache receives a read request issued by another core (GetS) while in state M, it switches to state S, losing write permissions to preserve the SWMR invariant. Since it holds the last recently modified copy of the requested block, the cache must provide the block to the L1 cache which issued the GetS and also has to update the stale copy stored in the LLC (which is in charge to answer to block requests while the block is shared), so two data messages are sent through the NoC.

Figure 2.3 shows the FSM for the full MOESI protocol. A first difference comparing to the MSI protocol is that a cache block can go to state S or to state E when the local core issues a read request: if the block is already present in any other L1 cache of the CMP

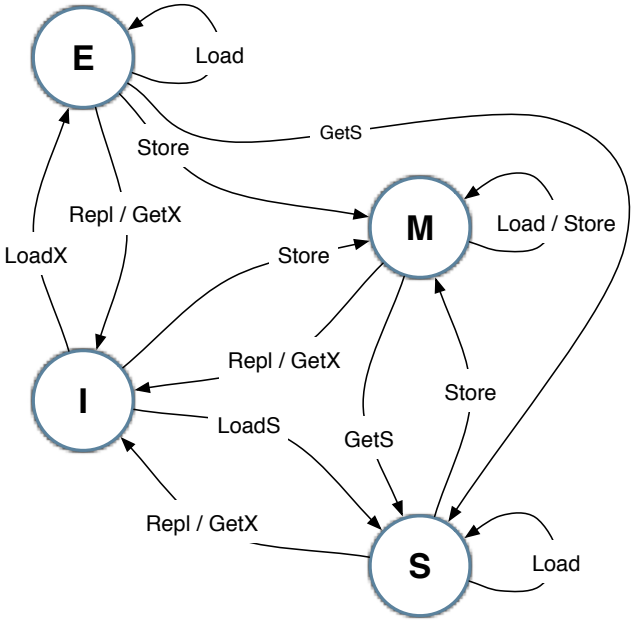


FIGURE 2.4: Simplified FSM for the MESI protocol.

(LoadS), the state switches to S, while if no L1 cache has a copy of the block (LoadX) the state switches to E. A block in state E has read and write permissions, so it can be written without issuing any coherence request (GetX) through the NoC: since only one L1 cache in the system has a copy of the block, the block can be modified preserving the SWMR invariant. Once a block in state E is modified, it goes to state M. The second difference with the MSI protocol is that a block in state M which receives an external read request (GetS) goes to state O instead of state S. Although the block is now shared, the L1 cache which holds the copy in state O is still the owner of the block (the copies in the other L1 caches are in state S) and it is in charge of providing the block when it is requested by other L1 caches. This optimization allows to avoid updating the LLC when a GetS is received while in state M, saving a data message; this is particularly useful if the LLC has high access latencies, as in systems where the LLC is located off-chip.

In CMP systems the LLC latency is relatively low, and the benefits of avoiding updating the LLC are nullified by the additional indirection step which state O introduces in directory-based protocols, so the common choice is to use MESI states at the L1 caches. The simplified FSM of the MESI protocol we assume in this thesis is shown in Figure 2.4.

The actual FSMs of MOESI and MESI protocols also include transient states which are needed to solve the race conditions due to simultaneous accesses to the same block by

different cores; the additional states are not shown in Figures 2.3 and 2.4 due to the complexity they would add to the figures. The protocols are, however, listed in the Appendix A of this thesis.

2.1.1.3 Snoopy and Directory Protocols

Cache coherence protocols can be classified in two families depending on whether the request issued after an L1 cache miss is broadcasted to all the caches in the system or sent to a specific node: in the first case the coherence protocol belongs to the family of snoopy protocols, while in the second case belongs to directory protocols.

Snoopy protocols usually rely on a shared communication medium (typically a bus) with a total ordering of messages. In case of an L1 cache miss, a request is broadcasted to all the caches in the system. Each cache controller's FSM evolves depending on the current cache line state and on the request type, and the protocol is designed to let all the caches independently evolve to a global correct state which guarantees the SWMR invariant. Messages in the interconnection network must be totally ordered to let all the caches see the same order of issued requests. The main drawback of snoopy protocols is due to the shared interconnect which limits their scalability.

In directory protocols the request, in case of an L1 miss, is sent to a single node, referred to as the *home* node, which is in charge of managing all the requests issued by L1 caches to the same block and thus acts as the synchronization point: the requests are managed following the order of their reception at the home node; the home node is typically associated to the lower shared level of the memory hierarchy. Directory protocols rely on a data structure, called the *directory*, to keep track of the cores which have a copy of each block in their private cache. This information is used by the LLC to locate the tiles[3] to which it has to communicate to when a coherence action must be triggered.

In its typical implementation, the directory consists of a bit vector associated to each cache line, with the size of the vector equal to the number of cores in the system. This limits the scalability of directory protocols since the directory introduces an area overhead which grows with system size.

[3]In this thesis we use both terms core and tile interchangeably.

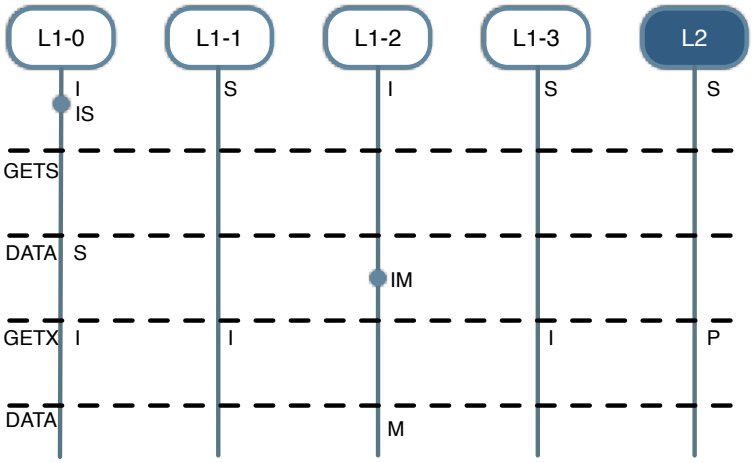


FIGURE 2.5: Snoopy protocol example.

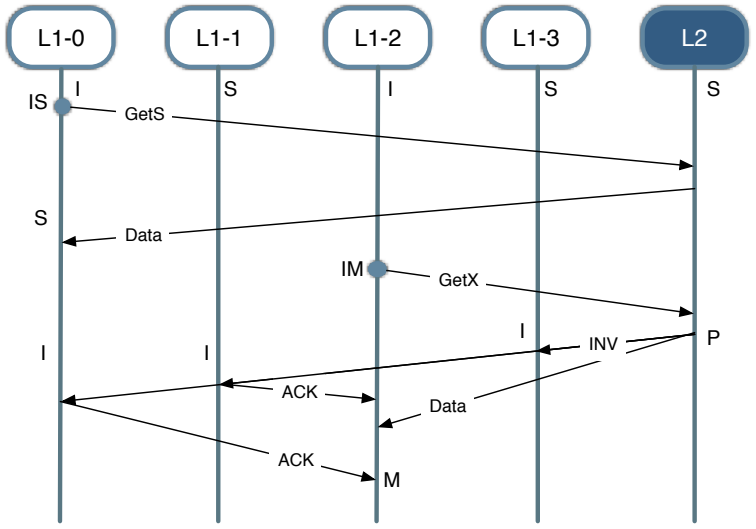


FIGURE 2.6: Directory protocol example.

Figures 2.5 and 2.6 show an example of how the two protocols manage the same sequence of requests generated by L1 caches in a system with 4 cores and one bank of L2 cache. States IS and IM are the transient states used when the block is changing its state from I to S and from I to M, respectively. At the beginning, the block we consider is shared by the L1 caches of cores 1 and 3 (L1-1 and L1-3). Then, a read request is issued by L1-0 and a write request by L1-2.

If a snoopy protocol is used, L1-0 broadcasts its request; when the L2 cache receives the request, it sends the data block to L1-0; the block is now shared by L1-0, L1-1 and L1-3. After a write miss in L1-2, a write request is broadcasted; when the request is received by the nodes which share a copy of the block, they invalidate their copy, while

the L2 cache sends the requested data to L1-2, which will hold the only valid copy of the block. If the coherence protocol is directory-based, L1-0 sends a unicast read request to the home node (the L2 cache bank), which adds L1-0 to the sharers list of that block and provides the requested data block. L1-2 also sends an unicast write request to the L2 bank, which in turn sends an invalidation message to each sharer and the requested data to L1-2; a field of the invalidation message includes the ID of the L1 which issued the request, while the data message indicates the number of sharers that are being invalidated. Each sharer, when it receives the invalidation message, invalidates its copy of the block and sends an acknowledgement message to L1-2; L1-2 can use the block only after the reception of the data and all the acknowledgments, meaning that all the sharers have invalidated their copy of the block.

2.1.1.4 Directory Implementation

In tiled CMP systems the directory structure is distributed in the LLC cache banks, usually included into the LLC tags portion [15]. In this way, each bank keeps the sharing information of the blocks which are cached on it. This sharing information comprises two main components: the state bits used to encode one of the basic steady states the cache controller can assign to a line and the sharing code, that holds the list of current sharers. Most of the bits of each directory entry are devoted to encode the sharing code, and therefore the total size of the directory structure is mainly determined by it. This extra storage adds requirements of area and energy consumption to the final design and could restrict the scalability of future many-core CMPs [16].

As far as the state bits are concerned, we assume four steady states for blocks at the LLC:

- P (Private): an L1 cache has a private copy of the block which may be modified or not (the state at the L1 cache is M or E)
- S (Shared): a set of L1 caches have a shared copy of the block with read-only permissions (the state of the block at each sharer's L1 cache is S)
- C (Cached): no L1 has a copy of the block: there is one copy in the LLC
- I (Invalid): the block is either not present in the LLC or it is present but not valid

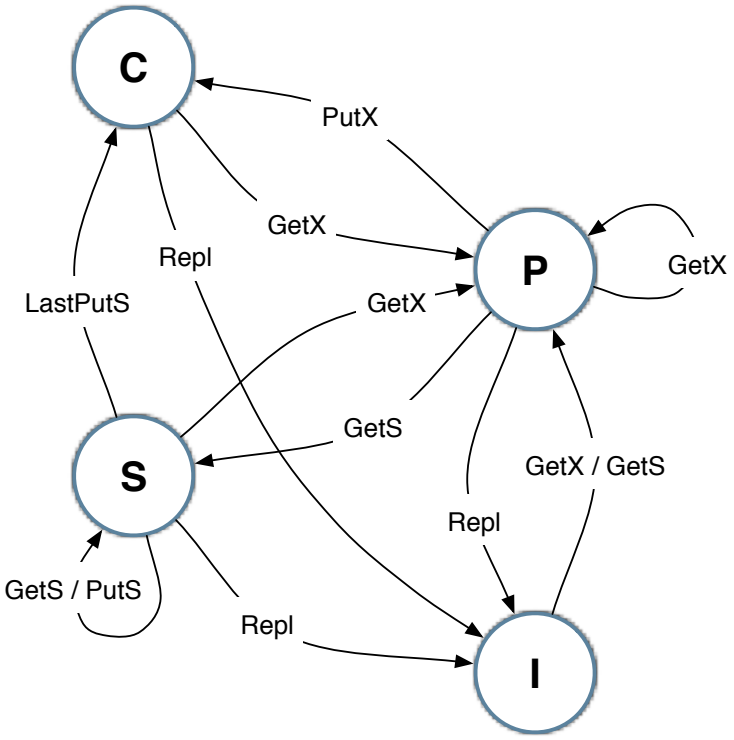


FIGURE 2.7: Simplified FSM for the LLC.

Figure 2.7 shows the simplified FSM for L2 blocks (the complete protocol is specified in the appendix): when a block which is not cached (state I) is requested for the first time, it is fetched from main memory and sent to the requestor with read and write permission independently from the actual request type; the block goes to state P in the L2 bank, while in the requestor L1 will be in state M or E depending on the operation which triggered the L2 request and on further local accesses. Write requests (GetX) received in state P do not change the state: the old owner is removed from the directory and the requestor is added as the new owner. If a read request (GetS) is received in state P, the requestor is added to the directory and the block switches to state S. When read requests are received in state S, the requestor is added to the directory, while if a write request is received an invalidation is sent to all the sharers, which are removed from the directory; the requestor is added as the new owner and the block switches back to state P. When a block is replaced by the owner (PutX in state P) or by all sharers (LastPutS in state S), the L2 bank has the only copy of the block, which is in state C. When any request is received for a block in state C, the data will be sent to the requestor with read and write permission independently from the request type, the requestor is added to the directory as the new owner of the block, and the block state in the L2 switches

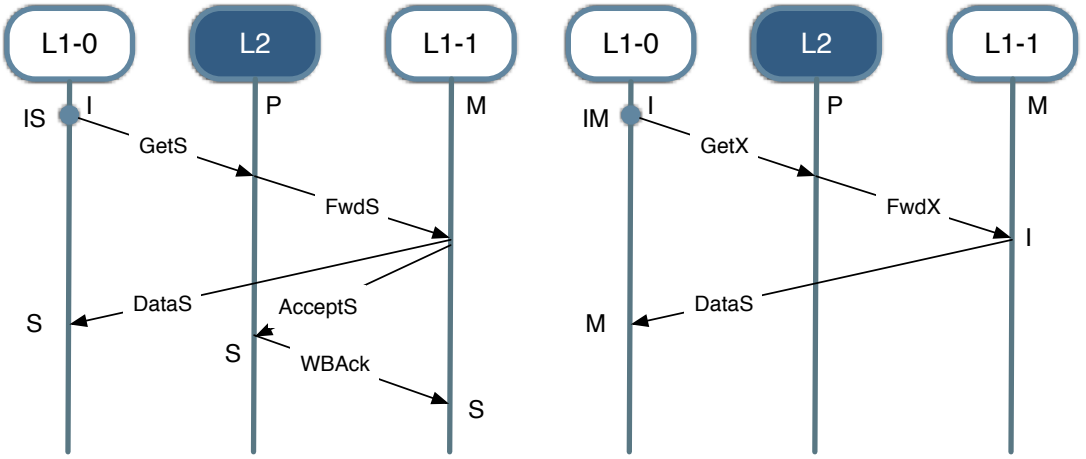
to P.

Many different organizations have been proposed to implement the sharing code, two of which deserve special attention. On the one hand, the sharing code can be implemented as a a bit-vector having one bit for each private cache in the system; if a private cache has a copy of a block, the corresponding bit in the sharing code stored in the directory entry associated to that block is set. This implementation, called *full-map* directory, provides an exact representation of the private caches holding a copy of the block in each moment, but its scalability is limited to tens of cores: for a system having 256 cores, for instance, the sharing code requires 32 bytes; assuming 64 bytes as the block size, this means that the directory increases the cache size by 50%. We refer to these protocols as *directory-based* protocols.

The area overhead can be reduced compressing the sharing code and mapping more than one private cache to each bit; this reduces the accuracy of the directory information, since when a bit of the sharing code is set it is not possible to determine which of the private caches mapped to that bit actually have a copy of the block, so when the LLC has to communicate with L1 caches to manage a request (i.e. it has to forward a request or send invalidation messages), it must send a message to all the caches mapped on that bit. Thus, the more the sharing code is compressed, the more area overhead is reduced, but NoC traffic increases, specially not useful traffic (messages sent to nodes which are not sharers).

By eliminating the sharing code we obtain what is called a Dir_0B protocol. This protocol does not dedicate any bit to the sharing code, so every coherence operation requires a broadcast to all the private caches in the system; we will refer to these protocols as *broadcast-based* protocols. The Hammer protocol employed in systems built using AMD Opteron processors [17], [18] is the most representative example of this family of protocols.

Summarizing, full-map directory protocols do not scale since the size of the directory grows linearly with the number of cores. However, these protocols generate the minimum amount of network traffic on the NoC. On the other hand, broadcast-based protocols completely remove the onerous part of the directory structure (the sharing code) resulting in a low overhead, completely scalable directory structure (just the state bits are needed in each entry). The drawback is that the number of messages on coherence events



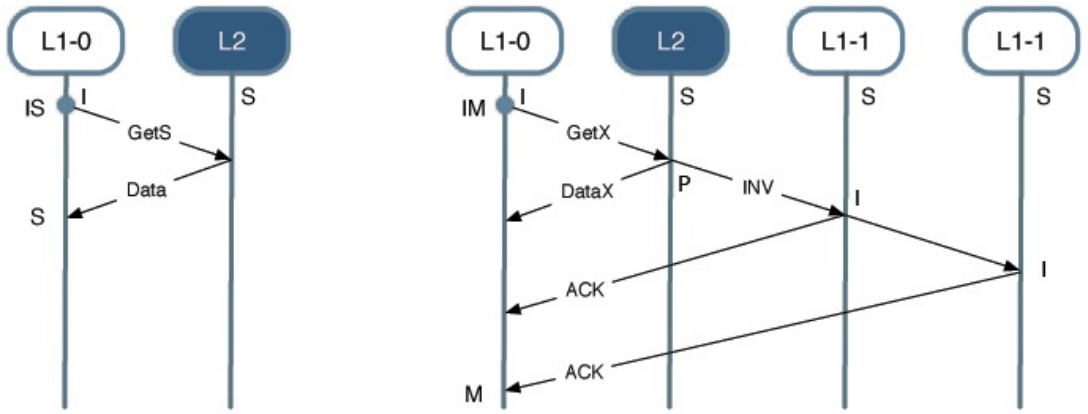
(a) read request (b) write request

FIGURE 2.8: Requests for a private block in full-map directory protocols.

increases linearly with the number of cores, which limits its applicability to systems with a small number of cores.

Focusing on full-map directory protocols, the basic four cases of coherence operations are shown in Figures 2.8 and 2.9. Figure 2.8 shows how an L1 miss is resolved if the requested block is private (owned by another L1 cache). The L1 nodes are identified by the tile number (L1-0 is the L1 at tile 0) and the L2 bank in the picture is the home of the requested memory block. In case of a load miss in L1-0 (Figure 2.8.a), a GetS message is sent to the home L2 bank, which forwards the request to the owner (L1-1) and adds the requestor (L1-0) to the list of sharers. The owner sends a copy of the block to L1-0 (DataS, where S indicates that the block is sent with read only permissions) and updates the L2 entry with the most recent value of the block (AcceptS). In case of a write miss (Figure 2.8.b), L1-0 sends a GetX message to the home, which forwards it to the owner and changes the information in the sharing code by eliminating L1-1 and adding L1-0, which is now the new owner. L1-1 sends the block with read and write permissions (DataX) to L1-0 and invalidates its copy.

Figure 2.9 shows how L1 misses are managed when the requested block is shared by a set of processors. In case of a load miss (Figure 2.9.a), a GetS is sent to the home, which adds the requestor to the list of sharers and sends the requested block back to the requestor with read-only permissions. In case of a store miss (Figure 2.9.b), a GetX is sent to the home bank, which answers by sending the block with read and write permissions to the requestor and by sending an invalidation message to the sharers. At



(a) read request (b) write request

FIGURE 2.9: Requests for a shared block in full-map directory protocols.

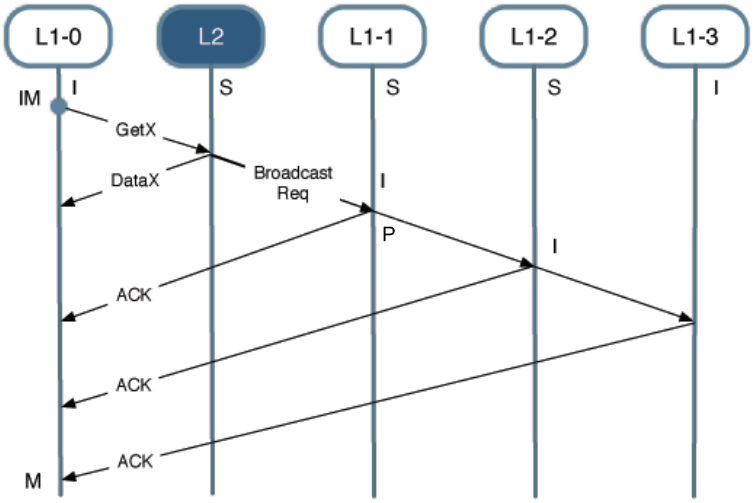


FIGURE 2.10: Write request management in broadcast-based protocols.

the reception of the invalidation message, each sharer invalidates its copy and sends an acknowledgement message to L1-0, which can complete the write operation only when it receives the requested data and all the expected ACKs (a field in the DataX message indicates the number of acknowledgements L1-0 has to wait for).

The broadcast-based protocol does not use the sharing code, so each time an L2 has to forward a request or to send invalidations to the sharers, it issues a broadcast message which is received by all the L1 caches in the system. When this broadcast request is received by the owner L1 cache, it sends the block to the requestor. If it is received by a sharer, it invalidates its copy and sends an ACK to the requestor. If it is received by an L1 cache which does not have a copy of the block, it just sends an ACK to the requestor. So, the broadcast-based protocol behaves as the directory-based protocol in

case of read request for a shared block (Figure 2.9.a) while in all other cases a message is broadcasted to all the L1 caches and the requestor receives a response message from all other L1 caches. For instance, the case of a write request for a shared block in a 4-core system is shown in Figure 2.10. All nodes reply to the broadcast by sending an ACK and invalidating the local copy, if any. L1-0 can write once it receives the data and all the ACKs.

2.1.2 Block Mapping Policies in Shared Banked LLCs

When banked LLCs are shared, the bank that is in charge to host and manage a block is called the *home bank* for that block. The easiest and most common way to map cache blocks to the shared LLC banks is static mapping: the address space is divided into subsets, and all the blocks of a subset are statically mapped to a single LLC bank. This policy is very simple to implement (e.g. the home node is chosen depending on the less significant bits of the block address) and theoretically distributes evenly the cache blocks on the L2 banks (this is however not true in real applications because memory accesses are never uniformly spread over the entire memory address space). The main drawback of this policy is that, since the position of the requesting core in the CMP is not taken into account by the mapping policy, a block can be mapped to a bank which is far from the core which is actually accessing it. A banked LLC using stating mapping is commonly referred to as Static NUCA or S-NUCA [9].

Various alternative policies have been proposed to minimize the access latency of an S-NUCA approach, software- and hardware-based. Cho and Jin [19] proposed an OS-based technique in which the address mapping to the LLC banks is still static but the OS is in charge to load the pages to the main memory depending on the desired policy. If a First-Touch policy is implemented, for instance, the OS takes into account the first thread which requests a block within a page, and that memory page is loaded to addresses which are statically mapped to the L2 bank located in the same tile of the core in which that thread is running. More sophisticated policies than First-Touch can be implemented to achieve a better mapping of the cache blocks to the LLC banks (e.g. Cho et al. [19], Ros et al. [20], Das et al. [21]), but it must be taken into account that complicating the OS mapping policy increases the complexity of the OS paging routine, which is in the critical path of the cache miss management.

Compile-time and data-based techniques have also been proposed in [22] and [23]. OS- and compiler-based techniques, however, rely on static mapping at hardware-level and cannot support hardware runtime techniques to adjust the initial mapping and further reduce the access latency, such as block migration [9] or replication [24] [25].

A first hardware-based policy to improve the performance of an S-NUCA was proposed by Kim [9]; this policy groups the banks in bank sets and maps each subset of addresses to a bank set, allowing blocks to migrate from a bank to another within a bank set to get closer to requesting processors. This policy, called Dynamic NUCA or D-NUCA, achieves lower access latencies than S-NUCA but complicates the process of requesting a block to the LLC, leading to a tradeoff between access time and NoC traffic since all the banks of a bank set must be accessed, leading to either high latency (sequential search) or more traffic (parallel search). Furthermore, it was proposed for a single-core system, and its extension to CMPs does not show the expected performance improvements due to various issues that have to be managed when multiple cores are sharing the same D-NUCA, such as the ping-pong behavior and the race conditions which lead to false and multiple misses [11], [26].

Based on the NUCA architecture, various proposals have been made to reduce the LLC access time through block replication (CMP-NuRAPID [27], Reactive-NUCA [24], Re-NUCA [25]) or using an hybrid policy to exploit the advantages both of private and shared LLCs (ESP-NUCA [28]). As it happens for OS-based policies, these techniques too are all based on static or partially static mapping, so the achieved LLC access latency is reduced but not optimal. Furthermore, some of them require network topologies more complex than the 2D mesh typically used in tiled CMP; Reactive-NUCA, for instance, is designed to run on top of a 2D torus, while CMP-NuRAPID requires a shared interconnect, which does not scale to an high number of cores.

In this thesis we assume static mapping in Chapter 3 and 6, whereas we assume dynamic mapping in Chapters 4, where we describe Runtime Home Mapping, a dynamic policy where block mapping is performed by the memory controller at runtime, while the block is being fetched from main memory, and Chapter 5, where Runtime Home Mapping is used to create a substrate for the independent partitioning of resources in a virtualized CMP system.

2.1.3 Power Implications

As the size of last-level caches is constantly growing, their power requirements represent a significant fraction of the overall system power. Dynamic power-off strategies have been proposed to reduce the leakage by powering entire LLC banks or single LLC entries. Powell et al. [29] propose the gated-Vdd design, in which a transistor is inserted between the ground and each LLC cache line to reduce the leakage current to a negligible level, thus powering down L2 cache lines used for private blocks; the value of these blocks indeed may be stale, since the owner L1 cache has the block with read and write permission and may have modified its value. Similar techniques have been proposed to reduce the supply voltage enough to reduce leakage without destroying the content of the cell, as done with drowsy [30] and superdrowsy [31] caches. The latter techniques have various drawbacks compared to the destructive technique proposed by Powell, being more difficult to implement and saving less leakage since a certain voltage has to be provided to the cell to keep the data.

At higher level, alternative cache architectures have been proposed to reduce static and/or dynamic power. Savings can be achieved by modifying cache parameters like cache size and cache associativity [32], [33]. Other efforts have been made to reduce the number of cache accesses, by using snoop filters [34], [35], way predictors [36] or filter caches [37]. Various proposals turn off L1 or L2 cache ways based on different prediction techniques. As an example, Kaxiras et al [38] propose to turn off L1 cache entries which are not expected to be reused. Abella et al [39] propose a different predictor to turn off unused L2 cache entries. Li et al [40] use different policies to turn off L2 cache entries when block copies are replicated in an L1, and evaluate both conservative and destructive voltage gating techniques.

All these works assume a 1:1 relationship between LLC entries and directory entries. In these thesis we propose a high-level approach where there are more directory entries than LLC entries: depending on the block being private or shared, it is stored in a directory-only entry or in an entry including both the directory and the LLC portion. This way we reduce the L2 cache size (and thus the leakage) by reducing its associativity, while keeping the directory associativity, which is vital to keep track all the on-chip blocks, either shared or private. Our proposal is orthogonal to all these works, and in Chapter 6 we provide results of our technique complemented with the one presented in [29].

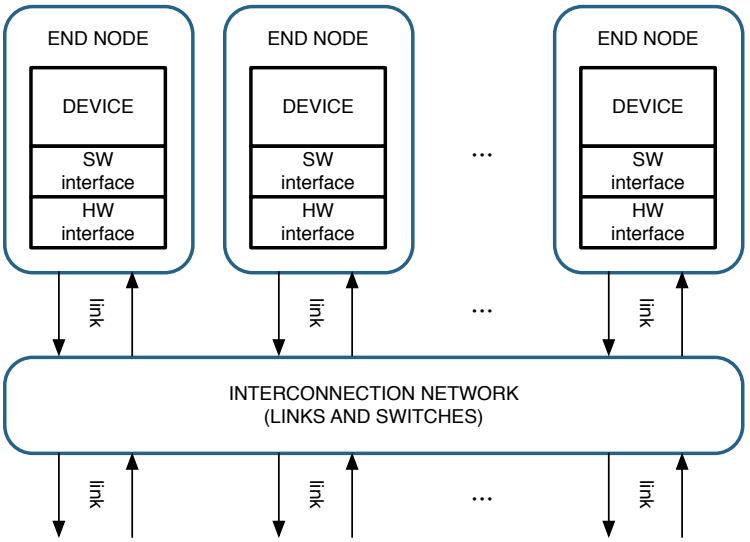


FIGURE 2.11: A general overview of a network architecture.

2.2 The Network-on-Chip

The NoC architecture is the result of several design choices like network *topology*, *switching* and *flow control* techniques and *routing strategies*. The network topology defines the physical interconnection between nodes and other elements. The switching and flow control techniques define how and when the information is transmitted through the network resources. Finally, the routing strategies manage the different path choices of communication between the nodes.

There are some common elements that can be identified in a network architecture. The first ones are the nodes. Nodes are the elements that communicate through the network and perform basically two main tasks: computation and storage. Nodes connect to other nodes through a network interface. Switches (also called routers) connect multiple devices. Links are the elements that connect all the devices (network interfaces and switches) present in the network architecture. Figure 2.11 shows the general structure of a network architecture.

In this section we provide initial background for the NoC components. However, for a full description, the reader is referred to [41].

2.2.1 NoCs Topology

Earlier on-chip communication architectures included buses as the communication subsystem, but the trend nowadays in the industry of high-performance computing is to include a reasonably large number of processing cores inside the chip, and shared-medium networks' poor scalability and bandwidth impacting heavily on the system performance.

NoCs emerged, thus, as a response to effective on-chip communication. As current multicore systems can be seen as a collection of tiles, there is a major taxonomy where chips can be differentiated between those having homogeneous (inducing regular topologies) and those having heterogeneous designs (more suited with irregular topologies). Every tile is connected to a subset of other tiles through the on-chip network. An example of homogeneous configurations are the tiled chip multiprocessors (CMPs) where all the tiles have the same structure as in Figure 2.1. On the other hand, high-end multiprocessor systems-on-chip (MPSoCs) are an example of heterogeneous designs where tiles are different in many aspects: size, functionality, performance, throughput, etc.

A popular choice in NoC designs is the use of orthogonal topologies as most of the direct network architectures are implemented with this property in mind. Orthogonal topologies, which are associated with regular patterns, allocate the nodes in a n-dimensional space, with k nodes along each dimension. Every switch has at least one link crossing each dimension and is labelled with an identifier depending on the coordinates. All links that communicate to other switches are bidirectional (formed by two channels, one in each direction). As the distance between two switches is the sum of the offsets in all dimensions, the routing strategy is usually implemented as a function of selecting the links that decrement the absolute value of the coordinate offsets between a source node and a destination node, a very simple mechanism. The most popular design in NoCs is the n-dimensional mesh, used in most of the commercial and non-commercial (prototypes) NoC designs. The most suitable topology is the 2-dimensional mesh (Figure 2.12). This kind of topology is vastly used (or at least assumed) because it fits the chip We assume this topology throughout this thesis.

As every switch is identified within the network by its coordinates on a n-dimensional space, a switch in a 2-dimensional graph will be numbered by a group of two coordinates, (x, y), one for each dimension. Crossing a link means decrementing or adding an unitary value to the offset of the dimension between the two switches that share the associated

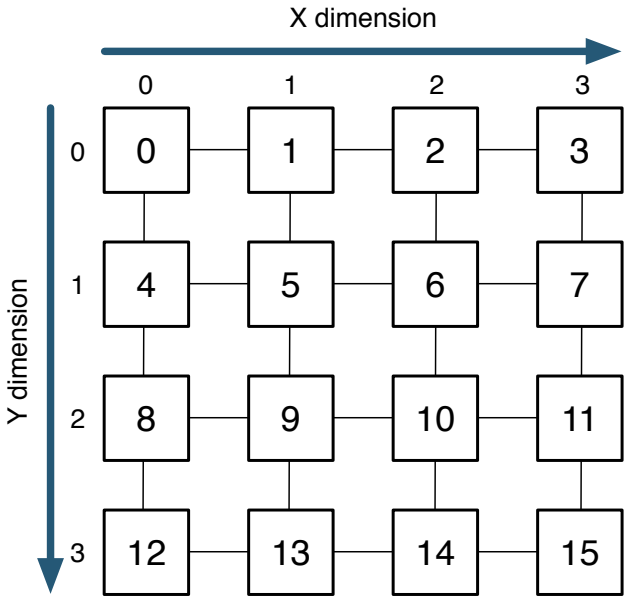


FIGURE 2.12: A 4×4 2-dimensional mesh.

link. Moving from switch 1, with coordinates $(1, 0)$, in $Y+$ direction results in switch 5, coordinates $(1, 1)$. Typically, switches are numbered by a single id, computed as a function of the coordinates and the number of switches per dimension.

2.2.2 The Switch

The switch, or router, is the tile component in charge of the communication between the associated tile and the rest of the tiles through the network layer. Typically, a switch includes the following main modules (Figure 2.13):

- *Buffers:* The task of a buffer is to store temporarily units of information (typically called flits, messages or packets). Buffers are associated to the channels that are connected to the switch. Channels, also called ports, are divided into input ports, streams that receive messages, and output ports, streams the send the messages to other switches or end nodes. Note that, to save area and power, buffers at the output ports are usually not implemented.

- *Crossbar:* The crossbar is the non-blocking switching element that allows the connection of all inputs of the switch to all its outputs. Crossbars are classified by their radix, i.e. the maximum numbers of connections they can make. Since crossbars do not scale, switches with many ports do not scale either.

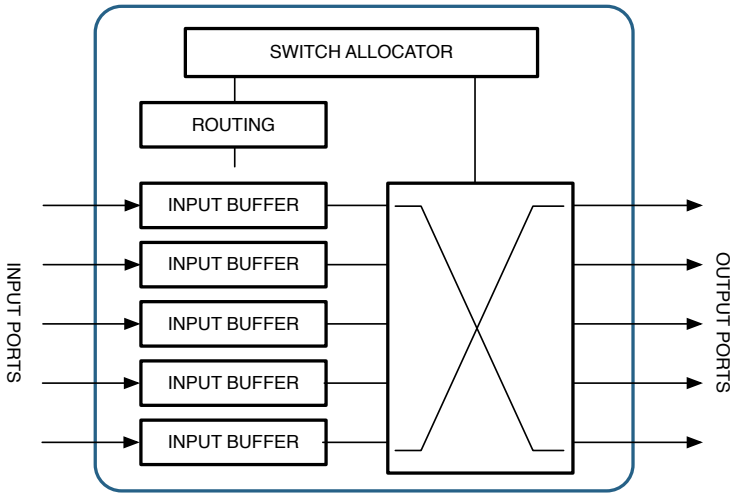


FIGURE 2.13: General structure of a VC-less switch.

- *Routing unit:* This unit is the responsible for decoding the unit of information provided by the incoming message, and based on the routing function and destination of the message, computing the most suitable output ports to transmit the message.

- *Arbiter unit:* This unit feeds from the routing unit and configures the crossbar accordingly to the requests between input and output ports, taking into account *switching* and *flow control* issues (both will be explained next in this Section).

2.2.3 Data Units

In an interconnection network, the general routing unit of information between nodes is the *message* (see Figure 2.14). A message is a collection of bits that the sender wishes to transmit to a destination (or a set of destination nodes), i.e. it contains the data that must be transmitted. A message typically includes a header, which contains the information for routing and control, to be used by the routing devices, a body which contains the data, and optionally a tail, for flow control or arbitration purposes. This information unit, however, due to resource restrictions affected by design choices, may need to be divided into smaller units, called *packets*, through a packetization process (usually performed at the network interface). Often, packet and message terms are interchangeable by the community, when both are equal in size. The term packet is usually employed even when the message has not been packetized.

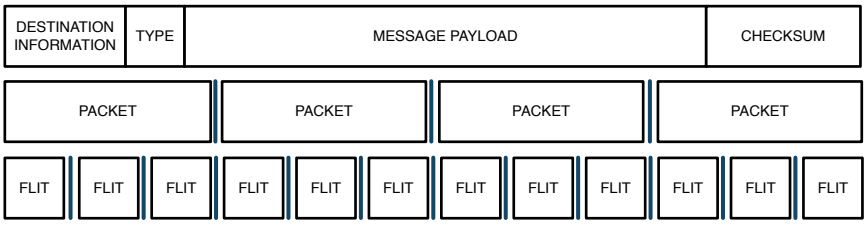


FIGURE 2.14: Data units.

A packet is divided further into *flits* (flow control digits), which are the smallest unit of flow-controlled information. As the width of the link can be lower than the size of a flit, the flit can be further divided at the physical level, into *phits* (physical digits). It is left to the designer and the parameters involved, the size of every unit. However, in on-chip networks, due to the vast amount of bandwidth available, the phit size usually equals the flit size.

2.2.4 Switching

Switching techniques control the allocation of network resources to messages/packets inside the switches. Their basic function is to set the connections between the input buffers and the output ports. The choice imposes several design constraints in the router that impact the performance, fabrication cost and power consumption of the elements in the network. Next we briefly describe the main switching techniques used in NoCs.

If *circuit switching* is used, the network establishes a reserved path between source and destination nodes prior to the transmission of the message. This is performed by injecting in the network a flit header, which contains the destination of the transmission and acts as some kind of routing probe that progresses towards the destination node reserving the channels that it gets. As the path has been reserved for this flow, messages cross the network avoiding buffer needs and collisions with other flows. The circuit is torn down when the transmission finishes. Circuit switching can be very advantageous when messages are very frequent and long. Nevertheless, if the circuit set up time is long compared to transmission time of the data, it will strongly penalize the performance of the network since links will be poorly used.

Instead of reserving all the path for a certain flow, there are some techniques that operate at packet granularity. These techniques are referred as packet switching. The most basic

technique related to packet switching is *store and forward* (SAF). When a packet arrives to a switch, the switch waits to store the whole packet in its input port buffer before the packet is forwarded. So, input port buffers must be large enough to store a whole packet. As can be deduced SAF has higher buffer requirements than circuit switching. In addition, latency of packets is multiplicative with hop count along the path (as the forward operation waits for the completion of the store operation).

The routing process however can be started once the packet header is received at the input buffer, without waiting for the rest of the packet. This is what is done in *virtual cut-through* (VCT) switching. In this case, as packets can advance through the switch once the packet header is received, the base latency for this switching technique is mostly additive to the distance between the nodes (hop count). Despite this, buffer requirements are the same for VCT and SAF. VCT requires there is enough free buffer space to store the entire packet. In fact, VCT behaves like SAF when the output link is busy. The switch needs to completely allocate the entire packet. This is the switching technique commonly used in off-chip high-performance interconnects [42], [43] since buffer size is not as critical as in NoCs.

VCT switching is an improvement over SAF, but in some network architectures, the choice of a buffer size to hold an entire packet could be prohibitive. The requirement to completely store a packet in the buffer of a router may prevent to design a small, compact, and fast router [43]. In *wormhole switching* (WH) buffers at the ports of a router only have to provide enough space to store only few flits, depending on the round-trip time delay (RTT) [4], instead of the whole message. In WH switching, the packet is forwarded immediately before the rest of the packet is entirely received, but as opposed to VCT, there is no need to have enough space for the rest of the message in case the message blocks. In that case, the entire message remains stored through the buffers of several routers. The major advantage of WH switching is the low storage requirements at routers. However, the most important drawback is that WH switching could lead to high contention levels at the network, because a message may block several resources when is traversing the network, causing low utilization of links and buffers.

To overcome the problem of contention induced by wormhole switching, *virtual channels* [44] were proposed. When using virtual channels (VCs) the buffer at the input port is

[4]Round-trip time delay can be defined as the elapsed time between a unit of information is sent and the acknowledgement of that transmission is received

divided into different virtual buffers and the channel is shared by all the virtual buffers. Of course this virtual multiplexing requires some local arbitration and must be taken into account by flow control and switching techniques. VCs can be used to improve message latency and network throughput. Their major drawback is that the available link bandwidth is distributed over all the VCs sharing a physical link, resulting in lower speeds. Again, in the on-chip network domain, the designer must evaluate the trade-off and the impact overhead on the network. VCs are not restricted to wormhole switching, the concept can be extrapolated to other design choices, depending on the need of their functionality (examples are deadlock-free routing algorithms and quality-of-service protocols).

2.2.5 Flow Control

Transmission of a flit between the input and output ports in a switch is a task performed by the switching technique. *Flow control*, however, is in charge of administering the advance of information between switches. Buffers are a temporary resource where to store flits, but they are finite. Flow control techniques are in charge of determining when the flits can be forwarded evaluating the capacity of the buffers and the link bandwidth.

There are three flow control mechanisms that are commonly used: *ack/nack*, *stop & go* and *credit-based*. The ack/nack flow control mechanism is based on data acknowledgements. When a flit arrives to a buffer, if the buffer has space available, then the flit is accepted and an acknowledgement signal (ack) is sent back. Instead, if there is no space available, the flit is dropped and a negative acknowledgement is sent. The flit must be retained at its origin until it receives a positive acknowledgement.

Stop & go emerged as an alternative to reduce the signalling (control traffic) between the sender and the receiver. Stop & go flow control is based on every buffer having two thresholds corresponding to certain sizes computed from the round-trip time. When the space occupied in the buffer reaches the *stop* threshold, a signal is sent back to the sender precisely to stop the transmission, taking into the account that still remains enough buffer space for the flits that are still being transmitted by the sender. When the buffer occupancy diminishes under or equal to the second threshold, *go*, then another signal is sent to reactivate the flow of flits.

With *credit-based* flow control, each sender, at its end of the link, maintains a count of credits, which is equal to the number of flits that can still be stored at the buffer on the receiver side. Whenever a flit is forwarded to the receiver buffer, as it occupies a slot, then the counter is decremented. If the counter reaches zero, it means that there is no available buffer space at the other end, and no flit can be forwarded. On the other hand, whenever a flit is forwarded and frees the associated buffer space, a credit is sent back to increment the counter. The drawback of this flow control mechanism is the significant amount of credit signaling sent backwards.

2.2.6 Arbitration

A router is composed of multiple input and output ports with their associated buffers and channels. Multiple inputs, according to routing decisions, may request the same output port. In this scenario, an arbitration operation is required to decide which one of the requests is allowed to connect to the output port. The arbitration mechanism must ensure to assign the output to only one of the inputs that have requested it, and the others must wait until they are allowed. As the arbitration operation introduces a latency to determine the assignment of the different output ports, it is critical for a NoC environment that these operations are performed fast enough to keep low latencies.

The main goal of an arbitration mechanism is to provide fairness between all the ports while achieving maximal matchings between requests and resources. For a description of different arbitration mechanisms, refer to [41].

2.2.7 Routing

Network topology defines the physical organization of the network composed by the nodes, and thus the available paths between all the nodes. The routing algorithm is the responsible of deciding which path has the message to follow to be effectively routed from its source to its destination. In the NoC domain, and more generally in any interconnection network scenario, the desired behavior is that every generated message from a source node arrives to its destination. However, even in the presence of available physical paths, there are several situations that prevent message delivery:

- Deadlock. A deadlock occurs when a message cannot advance toward its destination because the buffer requested by the message is full, being blocked by another message that is also waiting. A cyclic set of such events could make the messages to get permanently blocked. The two common ways to deal with deadlock events are deadlock avoidance, achieved employing a deadlock-free routing algorithm, and deadlock detection/recovery.

- Livelock. Livelock scenarios are similar to deadlock, but they happen when a message is misrouted and never reaches its destination as the links required to do so are always reserved to other messages. So, there are no permanently blocked messages, but a dynamic condition of blocking. Livelocks arise when non-minimal path routing is allowed and can easily be solved by limiting the number of times a message can be misrouted.

- Starvation. This issue arises when a message is permanently stopped holding a resource and cannot advance because the network traffic is so intense that the resources requested are always granted to other messages with higher priorities. This scenario is the result of an incorrect arbitration. Starvation is easily avoided by a proper design of arbiter and priority mechanisms.

All these issues occur because the number of resources (buffers) is finite, and specially in the NoC domain, is reduced. To face these problems, there are two ways of implementing the routing schemes and algorithms. The most suitable for NoCs is to prevent the formation of such scenarios (acyclic CDGs, no misrouting allowed, fair arbiters). The second one consists on recovery techniques to solve these kind of situations (cyclic CDGs). Recovery techniques also need extra circuitry to detect the presence of these issues. In this thesis we assume a routing algorithm that guarantees acyclic CDGs.

2.2.7.1 Implementation of a Routing Algorithm

Although any implementation is specific to a technology, there are two main trends to implement the routing strategy.

On the one hand we have *table-based* routing. Routing tables are basically composed of row-like structures that match destinations with table entries. So, given the destination

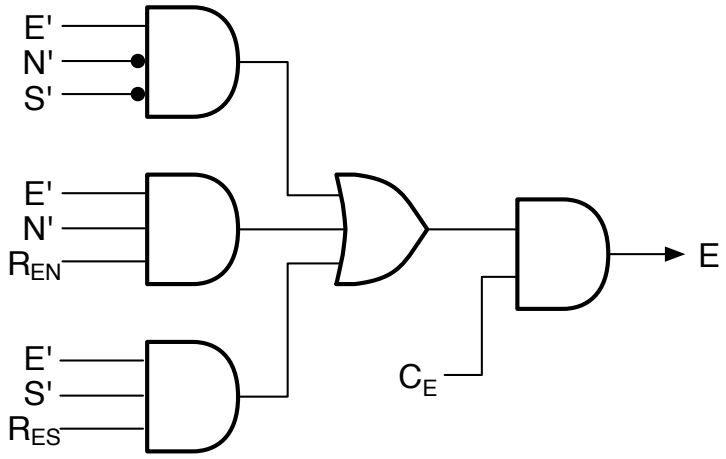


FIGURE 2.15: LBDR logic and configuration bits for east output port.

for a certain message, there is some circuitry associated that decodes this information, and accesses the routing table to find the routing decision associated to that destination. The most conventional way to implement these tables is to use memory structures. The advantage of table-based routing is flexibility, as the information of routing decisions stored on routing tables could be the answer of more complex routing algorithms, that are not only based on logical or arithmetical assumptions. On the other hand, routing tables implementation suffers from scalability, area, power consumption, and latency problems. For example, there is a penalty time (that increases with table size) associated to accessing memory structures.

On the other hand, *logic-based* routing can be used. This kind of routing is the result to translate a logical or arithmetical function of a routing algorithm into the equivalent in circuitry inside the router. So, when the message header is decoded at the input buffers, the output port is computed based on the hardware that represents the routing function. Logic-based routing is a good design choice in terms of delay, area, and power consumption. The main drawback is the lack of flexibility as these implementations could become non-functional if the topology scenario changes due to manufacturing defects, just to name a reason.

In this thesis we use the Logic-Based Distributed Routing (LBDR) approach [8], which tries to obtain the minimum table implementation of a routing algorithm. LBDR relies on the definition of a small set of configuration bits at the switches. For a 2-D mesh, LBDR requires 3 bits per output port at each switch. With these bits, multiple routing

algorithms can be encoded. A small logic block with seven logic gates per output port is needed. Figure 2.15 shows the logic and required bits for the east (E) port of a switch. The R_{en} (and R_{es}) bit encode the turn prohibition that may exist at the next switch for packets leaving the E port and taking at the next switch the north N (south S) port. If the R_{en} (R_{es}) bit is zero, then packets are not allowed to turn north (south). The N', E' and S' signals indicate whether the destination node is in the provided direction (N, E, or S). In this way, routing restrictions (pair of links that can not be used by the same message in order to avoid network deadlocks) can be encoded in the routing bits.

The C_e bit is used to indicate whether the E port can be used or not. It sets the boundary of the network to propagate messages. C_X bits can thus be used to define regions within the network, a feature that is particularly useful to partition network resources, as will be shown in Chapter 6.

Recently the mechanism has been extended to support non-minimal paths in presence of failures. Also, a basic broadcast mechanism has been proposed in [45]. In such broadcast, the connectivity bits can be used to limit the broadcast, constraining it into a partition.

In this thesis we assume that the routing bits are configured to encode the XY routing algorithm. However, in Chapter 6 we will not restrict to XY. We do not take into account the non-minimal path support but we use the broadcasting mechanism. For further descriptions of LBDR and its extensions, see [41].

2.2.7.2 Unicast, Multicast and Broadcast Messages

The nodes of an interconnection network send and receive messages through the switches present in the network. Assumed there is connectivity between all the end nodes, a node (or several nodes) may require to send the same information to several nodes, instead of only one. From the perspective of the sender, and given the amount of nodes that are meant to receive the data, there is a first distinction. If the message is sent to only one destination, we are talking about *unicast* or one-to-one communication ($1:1$). On the other hand, if the message must be sent to several destinations (that could include all the nodes on the network) then we are talking about some types of *collective communication*, again from the perspective of the source node. If the message must be

sent from a source node to the rest of the nodes in a chip then the routing operation is named *broadcast communication*, or one-to-all ($1 : all$). On the other hand, *multicast communication* or one-to-many, occurs when the sender distributes the message to a limited group of destination nodes ($1 : many$). Broadcast communication can be seen as an specific case of multicast communication. Broadcast communication is easier to implement because the incoming message is just replicated to the rest of switch ports but the drawback is flooding the network with unneeded messages. There are three basic methods to implement broadcast or multicast routing. The first one is the *unicast-based approach* or multiple one-to-one communication. This technique implements a collective communication operation by sending, in a sequential manner, a unicast message to every destination. While this solution requires minimum routing infrastructure, it tends to flood the network with many messages, resulting in higher latency communications. Power consumption is also high as there are many redundant messages in the network. The second one is the *path-based approach*. This solution relies on the injection of a single message with as many headers as destinations. The message uses a long path visiting all the destinations sequentially. Its downside is the message header overhead as well as the long path used, which impacts network latency. Also computation of paths is not trivial (so as to avoid deadlock) usually using a Hamiltonian cycle. The last one is the *tree-based approach*. Tree-based multicast or broadcast solutions rely on the use of a spanning tree mapped on the network (typically on a 2D mesh network) or region, providing collective communication to a set of destinations with the minimum amount of time. Routers create replicas when new branches are formed along the tree. This solution minimizes the number of messages sent through the network (the sender only injects one message per tree) with the associated reduction in power consumption and network latency. However, this approach usually requires a costly implementation, and is the one where avoiding deadlock is more complex.

In this thesis we use the bLBDR approach [45], which builds on top of the LBDR approach. bLBDR allows broadcast operations at partition level (a partition is defined with the LBDR bits inside the NoC). For further details, please refer to [41].

2.2.8 NoC and Cache Coherence

Many research efforts have focused on the co-design of the NoC and the cache hierarchy, aiming to optimize the NoC to fit the requirements of the traffic pattern generated by the caches according to the cache coherence protocol.

As a first measure, heterogenous network architectures have been proposed to meet the different requirements of the different classes of messages exchanged by the caches. Cheng et al. [46] leveraged the heterogeneous interconnects available in the upper metal layers of a chip multiprocessor, mapping different coherence protocol messages onto wires of different widths and thicknesses, trading off their latency-bandwidth requirements. Subsequently, Flores et al. [47] propose to combine a protocol-level technique (called Reply Partitioning) with the use of a simpler heterogeneous interconnect. In [48], it is presented a priority-based NoC, which differentiates between short control signals and long data messages to achieve a significant reduction in cache access delay. Additionally, the authors propose to use more efficient multicast and broadcast schemes instead of multiple unicast messages in order to implement the invalidation procedure and provide support for synchronization and mutual exclusion. Walter et al. [49] explore the benefits of adding a low-latency, customized shared bus as an integral part of the NoC architecture. The bus is used for some transactions such as broadcast of queries, fast delivery of control signals, and quick exchange of small data items. More recently, Vantrease et al. [50] advocate nanophotonic support for building high-performance simple atomic cache coherence protocols.

Another interesting research direction aims to raise the level of integration of the NoC and the cache coherence: Eisley et al. [51] proposed In-Network Cache Coherence, an implementation of the cache coherence protocol within the network based on embedding directories in each switch node that manage and steer requests towards nearby data copies. This approach enables in-transit optimization of memory access delay and shows good scalability.

Filters have been proposed as a solution to reduce the traffic generated by the caches and reduce unnecessary cache accesses, thus improving the overall system performance and limiting the energy consumption. JETTY [34] and Blue Gene/P [35] are two proposals to filter the broadcast requests that would miss at destination nodes in order to reduce

energy consumption due to cache look-ups. Filtering has also been proposed at source nodes [52], [53] to save energy and bandwidth.

Due to the high number of collective communication generated by some classes of coherence protocols, several proposals are targeted to provide an efficient support for multicast and broadcast operations, as in [54]. Additionally, it has been evaluated the case of using this kind of support in combination with a cache coherence protocol implementing imprecise directories (the Hammer protocol could be seen as using an inexact directory), demonstrating that multicast support alone is not enough to completely remove the performance degradation that the inexact sharing codes introduce [20]: some actions must be taken indeed to tackle the high percentage of traffic due to the acknowledgements to those multicast messages. The Gather Network, described in Chapter 3 of this thesis, aims to solve this problem.

2.3 Evaluation Platform

In this section we briefly describe the tools used throughout this thesis. Not all the tools are used in all the technical chapters. We select to use them on a chapter, as a function of its suitability and need for the chapter. However, gMemNoCsim is the main tool used in all the chapters.

2.3.1 gMemNoCsim

The main tool used to implement and evaluate the techniques proposed in this thesis is gMemNoCsim, a cycle-accurate NoC and cache hierarchy simulator developed by the Parallel Architecture Group at the Universitat Politècnica de València.

The simulator core is a detailed network infrastructure that simulates cycle-by-cycle the advance of flits through the network from source nodes to destination nodes; this network core has been extended with a memory layer to implement the cache-level proposals presented in this thesis. The current structure of the simulator includes thus two modules as shown in Figure 2.16: a top-level module which implements and simulates the cache hierarchy, and a module containing the original core, which is used by the components of the memory module to send and receive messages.

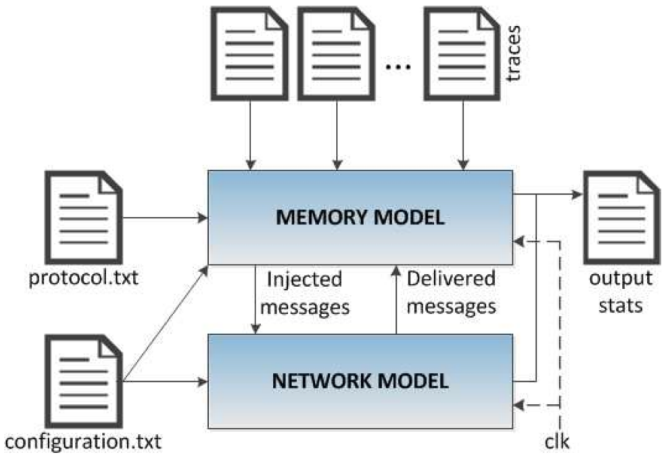


FIGURE 2.16: Structure of gMemNoCsim.

An input file describes the FSM of the L1 and L2 caches, while another input file defines network parameters (topology, routing algorithm, switching, flow control, flit size, ...) and cache parameters (size and associativity of L1 and L2 caches, line size, block mapping policy, tag and cache access latency). L1 cache accesses, which may be read from a trace file or generated by the cores of an external simulator into which our tool is embedded, are sent to the memory model. This module performs a cycle-by-cycle simulation of the cache hierarchy and the coherence protocol; if caches located at different levels of the cache hierarchy or located at different tiles have to communicate, a message is injected into the network model, which simulates the advance of the message through the NoC. When a message reaches the switch connected to the destination node (an L1 cache, an L2 cache or the memory controller) it is delivered to the memory module, and the destination node evolves as established by the coherence protocol. When the simulation ends, a file containing output statistics is generated, including the execution time, cache statistics (hit and miss rate of each cache, load and store latencies of each L1 cache, statistics for each coherence event, etc...) and network statistics (number of injected messages and flits, average latency, number of unicast and multicast messages, etc...).

If it is used in stand-alone mode, gMemNoCsim is trace-driven: one trace file per core lists the L1 accesses issued by that core; each trace includes a time offset (in cycles), the word address and the access type. Four access types are supported: fetch, load, store and barrier. The latter is a special access type which describes a barrier synchronization; when a core reaches barrier in its trace file, the next trace entry is not read until all cores have reached that barrier.

In this thesis work, the NoC (including the Gather Network presented in Chapter 3 and its extensions), the cache hierarchy and the cache coherence protocol of all the baseline and novel systems considered in the evaluation phase have been implemented in gMemNoCsim. In some cases, simulations have been run using trace files, containing synthetic access traces or memory access traces of real applications. In most cases anyway gMemNoCsim has been embedded in an external simulator, such as Graphite, and used for the timing of caches and the NoC.

2.3.2 Graphite

Graphite [55] is a multicore simulator developed by the Carbon Research Group, part of the Computer Science and Artificial Intelligence Lab at MIT. It was designed with the aim to provide a tool for the fast exploration of the design space of multicore architectures containing tens and hundreds of cores. To speedup the simulation, it has a distributed structure: each simulated tile is mapped to a simulator thread, and the threads of the simulated system can be executed in different cores of the host machine, or even in different machines communicating via TCP-IP. The synchronization between threads can be adjusted to tradeoff speed versus accuracy (the more threads are synchronized, the more accurate the simulator is, but slower). In this thesis, Graphite is used to capture the memory accesses of system's cores while executing applications of the SPLASH-2 benchmark suite.

2.3.3 Sniper

Sniper [56] is a multicore simulator developed at Intel Exascience Lab. It is based on the Graphite infrastructure and the principles of *interval simulation* [57]; the aim of Sniper is to provide a fast and accurate tool to evaluate homogeneous and heterogeneous multi-core architecture. Through interval simulation, Sniper achieves high simulation speeds keeping an accuracy which is very close to that of cycle-level simulators. In this thesis, Sniper is used to capture the memory accesses of system's cores while executing applications of the PARSEC benchmark suite.

2.3.4 CACTI

CACTI [58], developed by HP Labs, models area, latencies, dynamic power and leakage of cache memories depending on a wide spectrum of input parameters. CACTI is used in this thesis to calculate the access latencies and the power requirements of L1 and L2 caches; it is also used to evaluate the area reduction due to the technique described in Chapter 6.

2.3.5 Orion-2

Developed by Peh and Malik at Princeton University, Orion [59] is a suite of dynamic and leakage power models developed for various architectural components of on-chip networks. We use Orion to evaluate the power consumption of the basic 4-stage switch assumed in this thesis.

2.3.6 Xilinx ISE

Xilinx ISE is a design suite produced by Xilinx. It includes tools to develop, simulate, debug and synthesize HDL designs. This suite is being currently used to implement a CMP system which includes the main contributions of this thesis, as illustrated in Appendix B.

Chapter 3

Network-Level Optimizations

In this chapter we describe and analyze the different network-level optimizations for the efficient support of coherence protocols. In particular, we introduce the Gather Network, a dedicated control network used to gather and transmit many-to-one acknowledgements in tiled CMPs; two implementations of the Gather Network are described and evaluated, the first one completely combinational and the second one using sequential logic. The Gather Network provides a fast notification infrastructure to the cache coherence protocol that will be used in the following chapters.

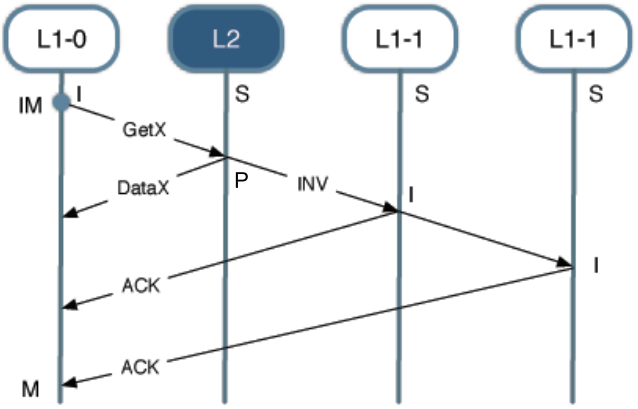


FIGURE 3.1: Write request for a shared block in a Directory protocol.

3.1 Introduction

Among the message types generated by cache coherence protocols, acknowledgement messages (ACKs) represent a special case: these messages indeed do not deliver any information besides the fact that, when the destination node receives an ACK, it is notified about the completion of all the operations established by the coherence protocol at the node which originated the ACK. The relevance of an ACK is therefore temporal: the destination node is typically waiting for its reception to complete further coherence operations on a block. Thus, ACKs are part of an easy way to provide synchronization among the cache controllers of different caches. ACKs are widely used by coherence protocols to manage block requests, writeback operations, block search in D-NUCAs, etc.

Focusing on block requests, the percentage of ACKs over the total number of messages depends basically on the coherence protocol: in a full-map directory protocol, ACKs are used to manage write requests on a shared block, as in Figure 3.1: the home L2 bank sends an invalidation message to the sharers, which invalidate their copy of the block and send an ACK to the block requestor, which cannot complete the write operation until all ACKs are received. Broadcast-based protocols, on the other hand make an extensive use of ACKs: each time the LLC has to communicate with one o more L1 cache to manage an incoming request, a broadcast is sent to all the caches, which acknowledge their reception.

In both cases, ACKs have an impact in overall network traffic and, since they are sent roughly at the same time, they may serialize at the input buffer of the destination

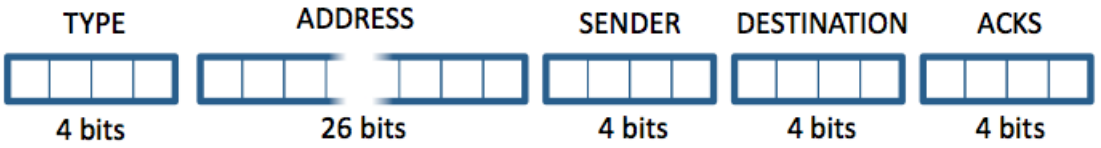


FIGURE 3.2: Format of a short coherence message.

node, thus further increasing the miss latency. The only relevant information in ACK messages is the ID of the destination node; if we assume in-order cores, even the block address is redundant since the requestor is blocked waiting for the reception of the acknowledgments (in case of an out-of-order processor this might change). However, these messages are typically transmitted through the NoC using the same short message structure of requests, thus wasting NoC resources and power to transmit a packet in which most of the fields are not used or redundant. Considering the short message structure of Figure 3.2, the useful information of an ACK message (the destination node ID) is 9.52% of a packet, so on the one hand actually ACKs do not need the bandwidth provided by the regular NoC and on the other hand they require a very low latency and a particular strategy to avoid the serialization at the destination node. These two considerations led to the idea of the Gather Network.

This chapter is organized as follows: Section 3.2 describes the Gather Network logical behavior and its implementation details; Sections 3.3 and 3.4 describe the logic needed to use the Gather Network with Hammer and Directory protocols, respectively; Section 3.4 also explains how the Directory protocol must be modified to be able to transmit invalidation ACKs through the Gather Network. Then, Section 3.5 provides evaluation results of the Gather Network used to speedup the acknowledgement phase in directory-based and broadcast-based coherence protocols. Conclusions are drawn in Section 3.6.

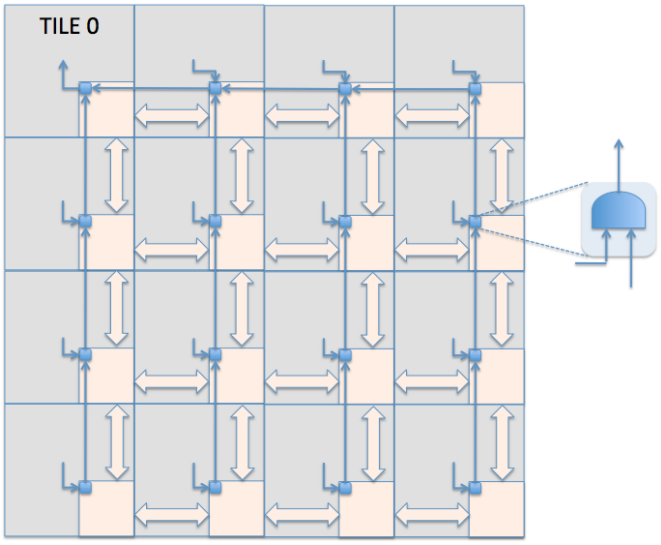


FIGURE 3.3: Gather Network (subnetwork for Tile 0).

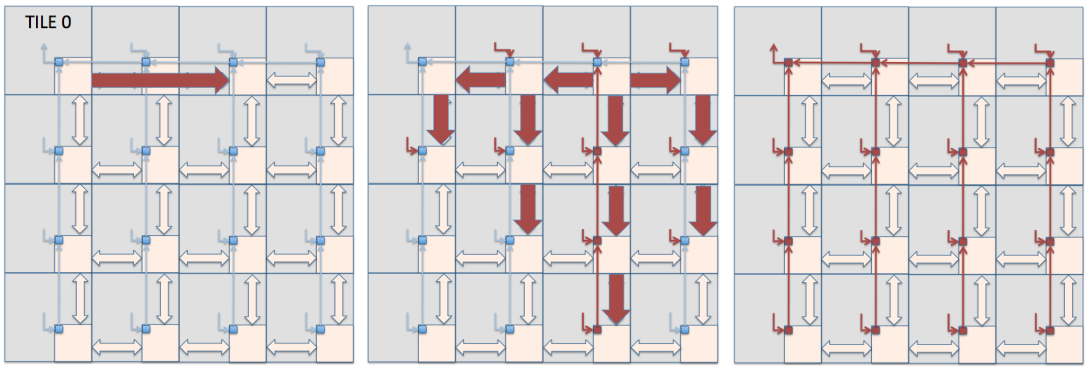
3.2 The Gather Network

The Gather Network (GN) is a dedicated control network which is added to the regular NoC and used to transmit many-to-one acknowledgement messages, thus relieving the NoC from the ACK traffic; ACKs originated by different nodes are gathered to deliver a single acknowledgement at the destination node; this eliminates the serialization of ACKs at the destination node's input buffer. It is logically composed by a set of one-bit wide subnetworks, one for each tile in the system: in our 16-tile reference system we thus define 16 subnetworks, each having 15 sources reaching a single destination. Figure 3.3 shows the subnetwork for Tile 0[1]: it is basically a tree of AND gates with its root in Tile 0 (the destination node) and the leaves in all the other tiles.

At each tile an AND gate combines the signals received from neighboring tiles and the one generated by the local node; the subnetwork routes the signals to Tile 0 according to the YX routing algorithm; this is related to the fact that we assume XY routing in the regular NoC (justification is provided later in this chapter).

Figure 3.4 shows an example how the GN gathers different ACKs to deliver a single global ACK at the destination node: L2-2 receives a write request from Tile 0; assuming Hammer protocol is used, the request is broadcasted to all L1 caches; Figure 3.4.b shows the state of the NoC when the broadcast has reached tiles three hops away from Tile

[1]Tiles are numbered 0 to 15 starting from the upper left corner of the CMP and descending row by row to the bottom-right corner.



(a) TILE 0 sends a block request to the LLC home bank

(b) the request is broadcasted by the home LLC to all other L1 caches

(c) all GN signals are set, TILE 0 receives the global ACK

FIGURE 3.4: Gathering ACKs in a broadcast-based protocol. Thick arrows indicate messages sent through the NoC, thin arrows indicate GN signals.

2: the L1 caches in the tiles which have already received the broadcast perform the coherence actions, if any, established by the coherence protocol and enable their outputs for the subnetwork with Tile 0 as destination node. When the AND gate at a tile has all its inputs set, the signal is propagated to the next hop of the GN: this happens for the tile in the same column of Tile 2; at Tile 2 the signal is not propagated since the west input is still not active (it will be active when Tile 15, in the lower right corner, receives the broadcast message). Once the broadcast message reaches all the tiles and all the input signals of the GN are set, the output at Tile 0 will also be set, thus notifying Tile 0 of the reception of the global acknowledgement (Figure 3.4.c).

If, instead, a directory protocol is used, the GN must be configured at each acknowledgement phase to enable only the signals of the tiles which have a copy of the block; this requires an additional logic at the AND gates and some coherence protocol modifications, as will be explained in Section 3.4.

3.2.1 Description of a Logic Block

The GN logic block at each switch is connected to the logic blocks of its neighbors with dedicated wires. Each logic block has the same general structure (a wire shuffling stage at the input side, a set of 16 AND gates and a wire shuffling stage at the output side) but the actual layout of the signal distribution and the number of inputs at each AND gate depends on the position of the tile in the 2D mesh. Figure 3.5 shows the logic at switch 5, assuming YX mapping for each subnetwork.

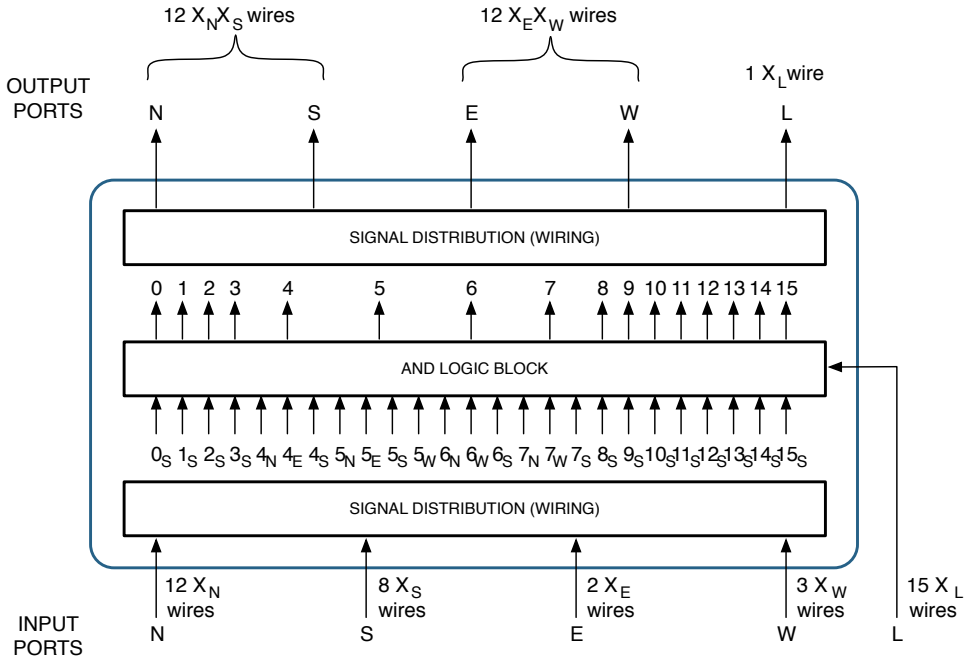


FIGURE 3.5: Logic block at Tile 5.

The logic receives 15 input control signals from the local core, each of which is addressed to a different destination node. X_L indicates a control signal coming from the local port and addressed to destination X. Thus, we have from 0_L up to 15_L signals (excluding the one for the local core, 5_L in this case). In addition, if we assume YX routing for all the subnetworks, there are up to 20 control signals coming from either north (N) or south (S) input ports and up to 5 control signals from either east (E) or west (W) input ports. In the case of Tile 5, 12 signals are received at the N port, 8 at the S, 2 at the E and 3 at the W. Switches at the boundaries of the mesh have lower number of input control signals. These signals are then distributed (based on the location of the switch in the mesh) and assigned to the corresponding inputs of the AND gate array. To simplify the picture, Figure 3.5 does not show the local inputs at the AND gates, which are present at every AND gate except for the one that generates the signal of the local node's subnetwork. Notice that each AND gate can have a maximum of 4 inputs, but in most cases only the local input and one or two more are present. The outputs of the AND gates are then distributed over the output ports, again depending on the location of the tile in the CMP and the routing layout; 16 output control signals are generated, 15 of which are distributed at the output ports and one is sent to the local node. Notice that the logic at each switch simply includes 16 AND gates for this first implementation. Later we will see some multiplexers and configuration bits will be needed. The signal

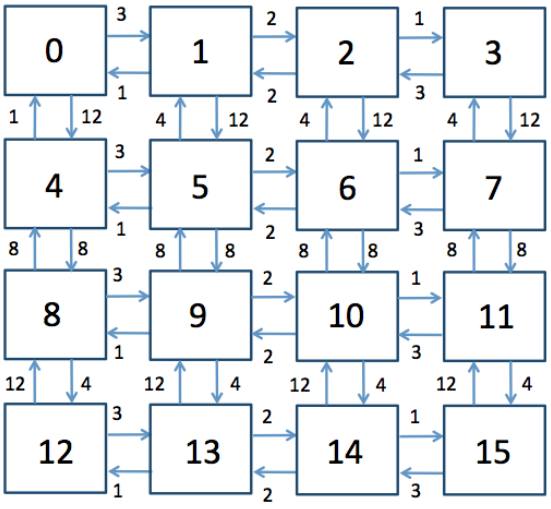


FIGURE 3.6: Control signals distribution (XY layout).

distribution blocks are simply a rearrangement of the input and output control signals to the appropriate inputs and outputs of the AND gates.

3.2.2 GN Wiring Layout

One important aspect of the GN is the floorplan of the wires over the NoC area. Figure 3.6 shows the number of wires of the GN wiring between the logic block modules at the switches. Each module handles both input and output control signals through all its ports. For a $N \times N$ mesh NoC, the number of outgoing control signals through all the output ports of a module is $N^2 - 1$, in our case 15. Each control signal belongs to a different one-bit subnetwork addressed to a different destination ($N^2 - 1$ destinations). Notice that some output ports handle more control signals than others. This is due to the mapping we assumed so far for the control signals, following the Y-X layout.

To better balance the GN wiring, it is possible to use a different mapping strategy with a mixed approach, where wires for half the destination nodes are mapped YX and wires for the other half of destination nodes are mapped XY. The latency of each subnetwork does not change as the path follows the same manhattan distance. Figure 3.7 shows the case where the subnetworks for tiles with odd ID number follow the YX mapping and the rest follow the XY mapping. In this case we achieve a perfect distribution of wires, where each bidirectional port handles 10 wires for a 4×4 mesh network. As will be shown later, this mapping cannot be used with Directory protocol, since the GN will need to follow the same path of the multicast invalidation message (but in opposite

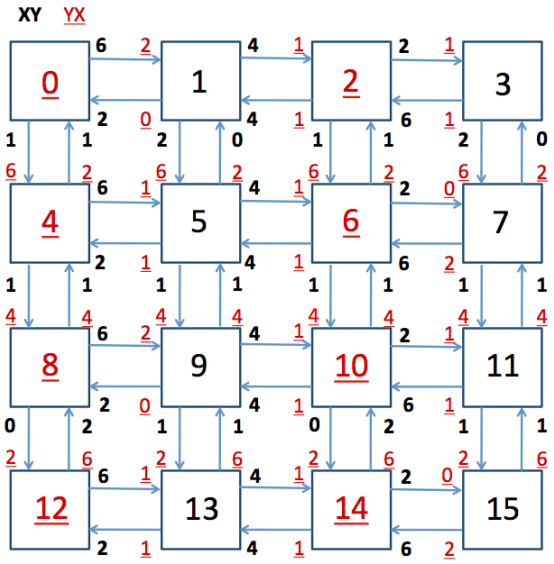


FIGURE 3.7: Control signals distribution (mixed layout).

direction) for a proper configuration of the GN. However, it can be used with Hammer protocol, which does not require to configure the GN.

As the system size increases, the number of wires also increases: for a $N \times N$ network, the mixed mapping strategy requires $(N^2 + N)/2$ wires per direction and dimension. For a 8×8 system, the number of wires per port is 36, well below the size of a typical NoC port, however.

3.2.3 Implementation Analysis

To evaluate the overhead of the GN we must first define the basic NoC switch; in this work we assume a basic 4-stage pipelined switch with 5 ports (north, east, west, south and local) and wormhole switching; the outputs of the switch are registered; each input port has a 4-flit-wide buffer. Link width and flit size are set to 8 bytes. Flow control is Stop&Go and routing is dimension-order (XY). A round-robin arbiter according to [60] is used. No virtual channels are implemented.

A basic switch with these configuration settings has been implemented using the 45nm technology open source Nangate [61] library with Synopsys DC. Cadence Encounter has been used to perform the Place&Route. Table 3.1 summarizes the delay and area for each of the modules of the switch.[2] Notice that these values do not take into account

[2] Area numbers in the table are for a single instance of each module, thus some of them are replicated in the complete switch.

module	area (mm^2)	critical path (ns)
Input port	3.08×10^{-3}	0.58
Routing	8.91×10^{-5}	0.30
Arbiter	1.09×10^{-3}	0.74
Crossbar	4.47×10^{-3}	0.43

TABLE 3.1: Area and delay for the switch modules

Critical path (ns)	Non Registered Switch		Registered Switch	
link length (mm)	1.2	2.4	1.2	2.4
conventional 2D mesh	1.86	2.17	1.35	1.75

TABLE 3.2: Conventional 2D mesh critical path.

the link delay neither the control logic needed to implement the communication between switches (these latencies are taken into account in Table 3.2).

The basic switch has been used to implement a 4×4 and an 8×8 2D mesh network, which is then analyzed with and without the GN.

When analyzing a conventional 2D mesh, although the arbiter stage is the slowest one in a switch, the critical path of the whole network is fixed by the delay of transmitting a flit through a link. This delay involves the XB delay of the upstream switch, the delay of the link, and the delay of the logic to select the input VC at the receiving switch. This delay is reduced when the output ports are registered. This design option enlarges the pipeline but reduces the critical path of the whole network. As the buffering and the operating frequency is increased, the power consumption of the network is also increased.

Table 3.2 shows the critical path of the conventional 2D mesh network. Two link lengths have been analyzed: 1.2mm and 2.4mm.

Table 3.3 shows the critical path (end to end delay) of the GN analyzed independently of the rest of the network. To compute the combinational GN critical path, each block must be properly placed besides the switch it is connected to. Then, on the implementation process some constraints must be forced to the placement&route tools. First, the highest metallization layers must be used. By doing this, lower metallization layers get free, and hence, other logic as SRAMs could be placed under GN wires. Repeaters are inserted by the own tool in order to fulfill delay constraints imposed by the designer. The critical path of the GN is fixed by the GN logic that connects the two nodes in the chip with the higher physical distance. Notice that the latency of the control network depends on

Critical path (ns)	4x4 Network		8x8 Network	
link length (mm)	1.2	2.4	1.2	2.4
Basic switch	1.35	1.75	1.35	1.75
Gather Network	1.23	2.20	2.65	4.32

TABLE 3.3: GN critical path.

the mesh radix. The table also shows the delay of a single switch. Two link lengths are analyzed: 1.2 mm and 2.4 mm.

For a 4×4 network with a link length of 1.2 mm, the GN critical path is smaller than the delay of a single switch, and hence, it is able to work at the same operating frequency than the switch (in a switch cycle the GN is able to notify all the nodes about possible *ack* messages). In contrast, if the link length is increased, the GN has a higher critical path. However, only two cycles are needed. For the 8×8 network, it can be seen that the GN does not scale as well as the point-to-point communication protocol of the NoC. However, it can be noticed that the worst case is for a GN with a delay of 4.32 ns (3 clock cycles when compared to the switch). The area of the switch is $20.418 \times 10^{-3}\ mm^2$, while the area of the GN in each switch is $0.28\ \times 10^{-3}\ mm^2$, being a 1.3% overhead. Notice that if virtual channels are to be added the area overhead will be much lower.

3.2.4 Sequential Implementation of the Gather Network

The combinational design described in the previous section is simple, fast and very efficient in area. However, the wiring requirements of the GN increase with the number of cores, leading to an unacceptable number of wires between switches as the system size is increased to hundreds of cores. Furthermore, the combinational design can handle only one request per core (in-order cores); to handle multiple requests at the same time it would be necessary to have more than one subnetwork per core, thus increasing the number of wires even more.

The implementation described in this subsection reduces the wiring requirements to a logarithmic scale and allows to handle multiple requests to the same tile at the cost of increased latency: while the combinational implementation is not bound to the clock frequency of the switches, this implementation uses sequential logic at each switch and has a latency of 1 cycle per hop. As will be shown in the evaluation section, this is not an issue, since the latency of the combinational GN can be increased up to tens

of cycles without affecting the system performance. On the other hand, the number of wires between switches is drastically reduced, providing a scalable solution for larger systems (beyond 64 tiles).

Figure 3.8 shows the sequential implementation of the GN at each switch. Each port has its own decoder and encoder, through which it can receive and send the IDs of the ACKs destination node. This reduces the number of wires, since each port will only need, in an system with N cores, $\log_2 N$ input wires and $\log_2 N$ output wires. The received IDs are decoded and saved in the input registers. When all expected ACKs are received for a node, an arbiter for each output port selects one output ID (since at the same cycle more than one AND gate can be activated for two different destinations which are reached through the same output port). The selected output ID is then encoded and transmitted to the next switch (or to the local node) and the reset logic sets to zero all the bits in the input registers belonging to the subnetwork of the destination node. Notice that the AND logic block at each switch is the same for both the combinatory and the sequential implementation. The number of wires in each connection shown in Figure 3.8 refers to a 4×4 system with a mixed XY-YX control signal distribution. The number of wires connecting the AND logic block with the input and the output blocks varies from 0 (in case the tile does not have any connection through that specific direction) to 6 depending on the tile position in the 2D mesh.

To allow multiple requests, new bits can be added to the ID field of the GN: this way with $\log_2 N$+R bits per port per direction it is possible to handle 2^R requests per core in a system with N cores. This means, for instance, that a sequential GN can handle 4 requests per core in a 4×4 system using 6 bits per port per direction, which adds a low overhead to the typical NoC ports of 128 or 256 bits.

The sequential control logic described above has also been implemented with Synopsys DC. This implementation has a higher area overhead than the combinatorial one, being basically the same circuit with decoders at the input ports and encoders at the output ports. The encoders at each output port also include an arbiter in case more than one ACK signal is generated at the AND logic block in the same cycle. Since the control logic area varies depending on the tile position, the worst case was considered, which in a 4×4 system with mixed XY/YX mapping corresponds to the module at one of the switches located in the central tiles. These switches indeed are connected with another

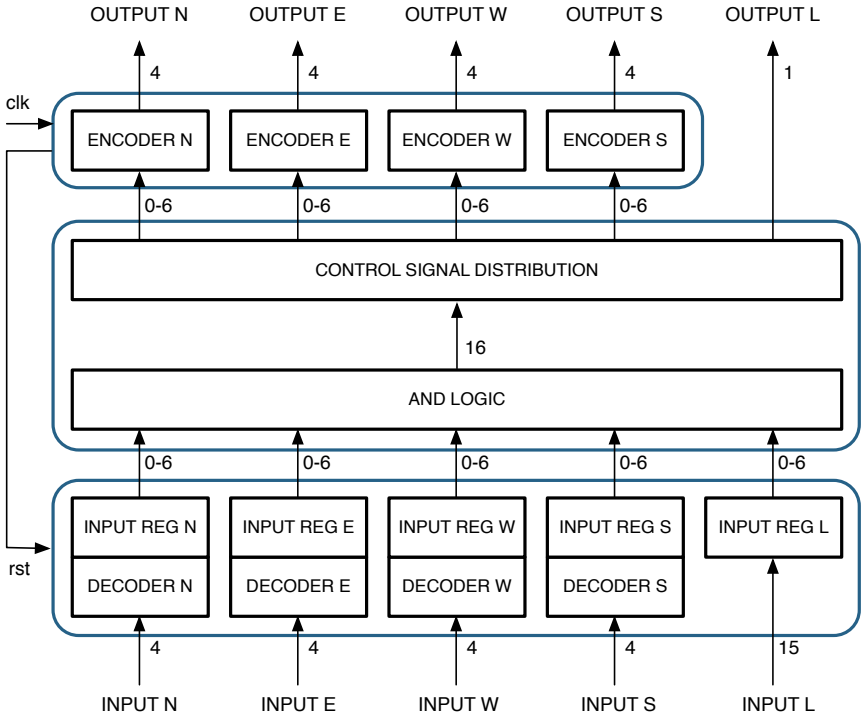


FIGURE 3.8: Structure of a sequential GN module.

	Area (mm^2)	Latency (ns)
Worst case (center tile)	0.472	$0.52 * 10^{-2}$
Best case (corner tile)	0.153	$0.21 * 10^{-2}$

TABLE 3.4: Area and latency results of the sequential GN module.

switch in each direction and are crossed by more signals than the switches located on the border. The area of the control logic in these tiles is $0.472 \times 10^{-3} mm^2$, which is 2.3 % the area of the basic switch. The critical path of the control logic in the worst case is 0.52 ns, which is lower than the critical path of the slower module of a switch (0.74 ns for the arbiter, as shown in Table 3.1). This means that at each cycle the control logic can propagate up to 5 ACK signals (one per port). The control logic area is reduced to $0.153 \times 10^{-3} mm^2$ for the switches located at the corners of the 2D mesh, which are connected only in two directions.

3.3 GN Applied to Hammer Protocol

Since the directory is completely eliminated in Hammer protocol, the LLC sends a broadcast to all L1 caches each time it has to communicate to one or more lower-level caches to manage a request (e.g. sharers must be invalidated due to the reception of a

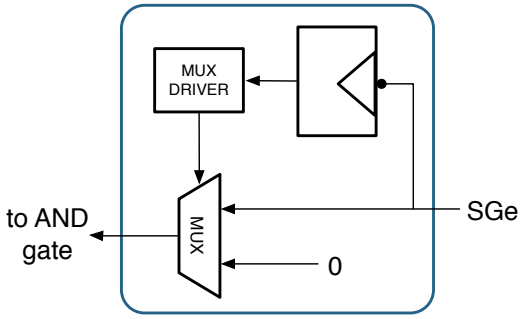


FIGURE 3.9: Logic at the inputs of each AND gate when the system implements Hammer coherence protocol.

write request, or a request for a private block is forwarded to the owner). The L1 caches, at the reception of the broadcast message, must send an ACK to the requestor to avoid race conditions between two subsequent requests. All the nodes located at the leaves of the requestor's AND tree participate thus in the acknowledgement phase. Therefore, the application of the GN to the hammer protocol seems straightforward. Whenever a node receives a coherence action, it acknowledges the requestor through the GN. However, between two consecutive GN operations (on the same destination node) a reset needs to be performed.

3.3.1 Reseting The GN Wires

Indeed, the inputs of the GN need to be reset every time a broadcast request is propagated through the NoC: the inputs indeed may be already set due to a previous request. Considering first the combinational implementation, Figure 3.9 shows the additional logic needed at the inputs of each AND gate to reset a subnetwork each time it is used. A mux is placed at each input of the AND gates; the output of the mux can be forced to 0 or be equal to the actual input of that module, which may be coming from the local node or from a tile located at lower levels of the AND tree; in the example shown in Figure 3.9, the input signal is marked as SGe, indicating that it is received from the tile connected to the east port.

When a broadcast request is received at switch A, the multiplexer output is forced to 0 if SGe is set, thus preventing the output of the AND gate from being set to one too early; then, as the broadcast message travels down to the next switch B, the input signal will be also set to zero at that switch; the output of the AND gate at switch B that

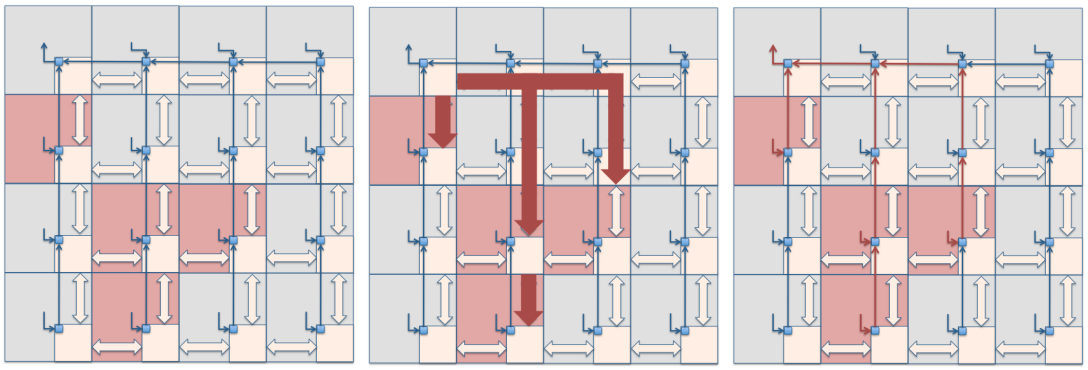
generates the value of SGe will then be zero. As the new value of SGe is received at switch A, the logic detects the transition of the input signal through the flip-flop and restores the multiplexer entry to be equal to SGe.

Considering the sequential implementation, the inputs of each AND gate are registered: when an ID is received, it is decoded and the register at an input of the AND gate associated to the destination node is set; all the registers at the input of an AND gate are reset when the output of the AND gate becomes 1 and the destination ID is propagated at the output port. Therefore, there is no need to temporarily set to zero any multiplexer output.

3.4 GN Applied to Directory Protocol

When the system implements a directory protocol, the subnetwork of the requestor must be configured at each invalidation phase to activate only the input signals of the sharers. This is done by exploiting the NoC's hardware multicast support: a single invalidation message is injected in the NoC by the home L2, which replicates at the proper switches to reach only the sharers. When the invalidation is forwarded through an output port, the corresponding input of the requestor's subnetwork is activated by setting a configuration bit (seen in the next section), while the subnetwork inputs corresponding to the output ports through which the invalidation is not propagated are disabled. At the end of the configuration phase (when the multicast invalidation message has reached all the sharers), the activated inputs of the requestor subnetwork will form a tree which has the same structure of the path followed by the invalidation message.

Figure 3.10 shows the configuration of the Tile 0's subnetwork in case the home L2 bank is also located at Tile 0. Figure 3.10.a shows the set of sharers marked in red; the subnetwork is configured through the multicast invalidation sent by L2-0 (Figure 3.10.b) and the AND tree resulting by the subnetwork configuration is the one marked in red in Figure 3.10.c, which follows the same path of the invalidation multicast but in the opposite direction.



(a) block sharers (marked in red) (b) multicast invalidation message (c) active GN inputs after the configuration

FIGURE 3.10: Configuration of the Gather Network.

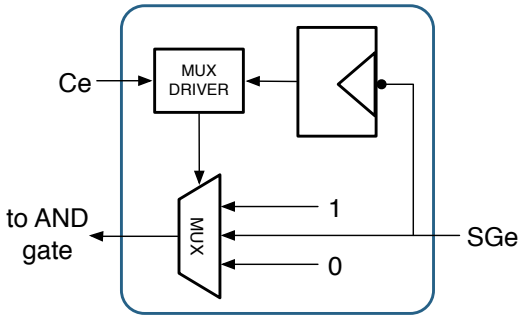


FIGURE 3.11: Logic at the inputs of each AND gate when the system implements Directory coherence protocol.

3.4.1 Reseting the GN Wires

Considering the combinational GN, the logic needed at the inputs of each AND gate when the system implements a directory protocol is shown in Figure 3.11. In this case an extra bit is needed to enable/disable that input signal (C_e in the picture). If the input signal is disabled, the logic forces the output of the mux to 1. In the other case, the multiplexer's output is forced to 0 until the reset of SGe, as described for Hammer protocol.

With the sequential implementation, the registers at the input of the AND gates are reset when the destination ID is propagated at the output port, as described for Hammer protocol. However, the configuration bit is still needed.

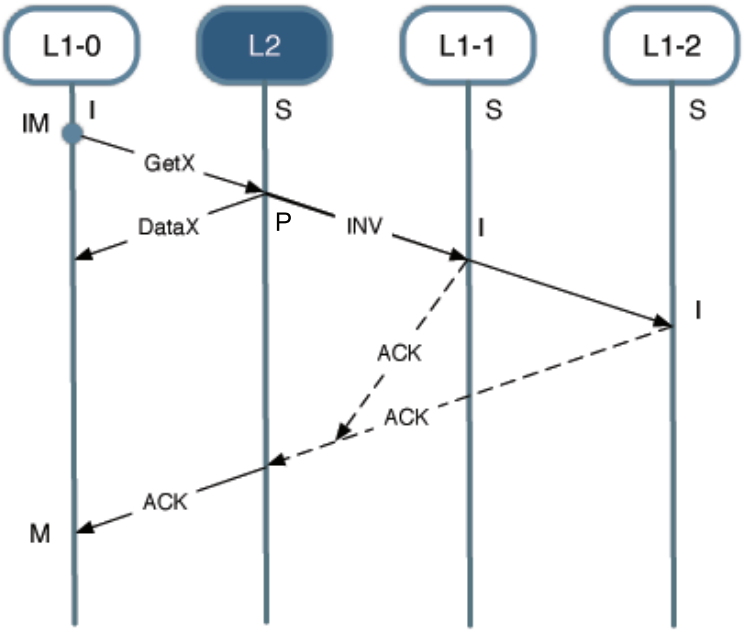


FIGURE 3.12: First alternative to let the GN work with directory-based protocols: acknowledgements are sent to the home L2.

3.4.2 Protocol Modifications

The directory protocol described in Chapter 2 must be modified to allow the correct configuration of the GN each time an invalidation multicast message is sent by the L2 cache. The multicast invalidation is used to configure the subnetwork through which the sharers will send the ACKs: the routing stage of every switch activates the local input of the GN if the local node is one of the destination nodes of the multicast invalidation, and the inputs from neighboring tiles if the multicast message is propagated to other switches.[3] Once the subnetwork is configured, the enabled signals will form a tree structure that reflects the propagation of the multicast invalidation, but in the opposite direction.

Since the configuration is done through the multicast invalidation, the source node of the invalidation message must also be the destination node of the ACKs. This is not true for the basic directory protocol: invalidations are sent by the home L2 bank, while the acknowledgements are sent to the L1 which originated the request that triggered the invalidation process.

[3]We assume hardware multicast and broadcast support at the switches.

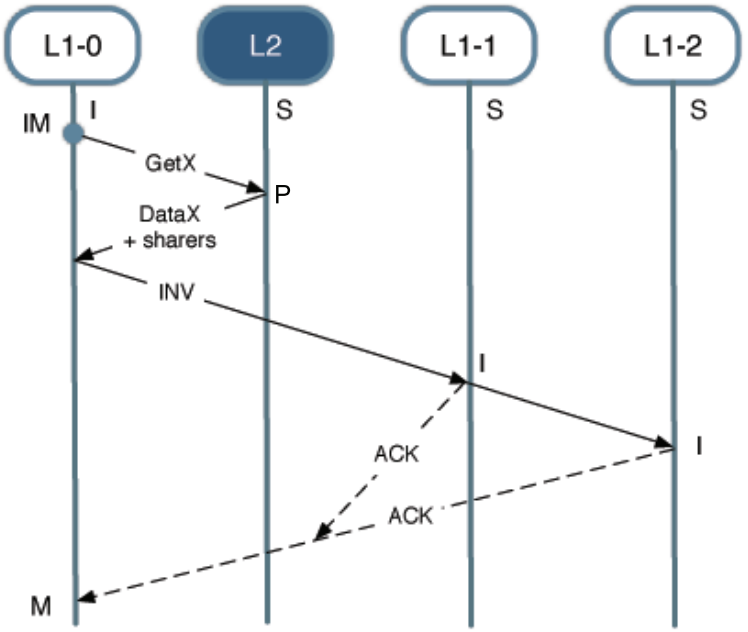


FIGURE 3.13: Second alternative to let the GN work with directory-based protocols: the L1 invalidates the sharers.

There are two alternative simple modifications that can be done to adapt the directory protocol to work with the GN: in the first one, shown in Figure 3.12, the home L2 bank sends a multicast invalidation message to all the sharers, which send the ACKs back to the L2 node through the GN. Once the home receives all the ACKs (the GN output is set), it sends a unicast ACK message through the regular NoC to the L1 requestor. In the second alternative protocol, shown in Figure 3.13, the home L2 does not invalidate the sharers, but it piggybacks the sharers list to the data message sent to the requestor. The requestor is in charge to send a multicast INV message to the sharers, which acknowledge the reception of the invalidation message through the GN.

In both cases, invalidating memory copies will take four steps, one more than the basic protocol. However, due to the fast reception of acknowledgements, both alternative protocols achieve better performance than the basic protocol, as shown next in the evaluation..

3.5 GN Performance Evaluation

This section provides evaluation results of the performance improvements obtained when a system employs the GN to transmit many-to-one acknowledgements. First, the GN

is evaluated in a system which implements a directory protocol, where many-to-one acknowledgements are used to manage the invalidation phase of shared blocks. Then it is evaluated with a broadcast-based protocol, which makes an extensive use of acknowledgements to manage coherence requests.

3.5.1 Directory Protocol with GN

Four versions of a system employing the directory protocol have been implemented using gMemNoCsim:

- basic: a configuration with the regular NoC and the basic Directory protocol
- mc: a configuration with a NoC-level multicast support and the basic Directory protocol
- mc+gL1: a configuration with the NoC-level multicast support, the GN and the custom protocol shown in Figure 3.13. The L1 cache is in charge of invalidating the sharers.
- mc+gL2: a configuration with the NoC-level multicast support, the GN and the custom protocol shown in Figure 3.12. The L2 bank is in charge of invalidating the sharers and then to notify the L1 requestor cache.

The last two versions have been evaluated considering a combinatorial GN with 1-cycle and 2-cycle latency. Other network and cache parameters are shown in Table 3.5. Each tile has two 64KB L1 banks (instruction and data) and a 512KB L2 bank. Tag access latency is set to 1 and 2 cycles respectively for L1 and L2 cache, while cache access latency is set to 2 and 4 cycles respectively for L1 and L2 cache.

To simulate actual applications on our system, gMemNoCsim was embedded in Graphite simulator, which allowed to run various applications of the SPLASH-2 benchmark suite. Since all applications generated a very low percentage of write accesses on shared variables (0.4% of total L2 accesses on average), the effects of the GN were quite limited, as shown in Figure 3.14.

Four sets of synthetic memory access traces have then been generated and fed into the simulator. Each set is made of 200,000 random accesses to 500 different addresses. The

Routing	XY	Coherence protocol	Directory
Flow control	credits	L1 cache size	64 + 64 kB (I + D)
Flit size	8 byte	L1 tag latency	1 cycle
Switch model	4-stage pipelined	L1 data latency	2 cycles
Switching	virtual cut-through	L2 bank size	512 kB
Buffer size:	9 flit deep	L2 tag latency	2 cycles
Virtual channels:	4	L2 data latency	4 cycles
GN delay	1 cycle / 2 cycles	Cache block size	64 B

TABLE 3.5: Network and cache parameters (GN with Directory protocol).

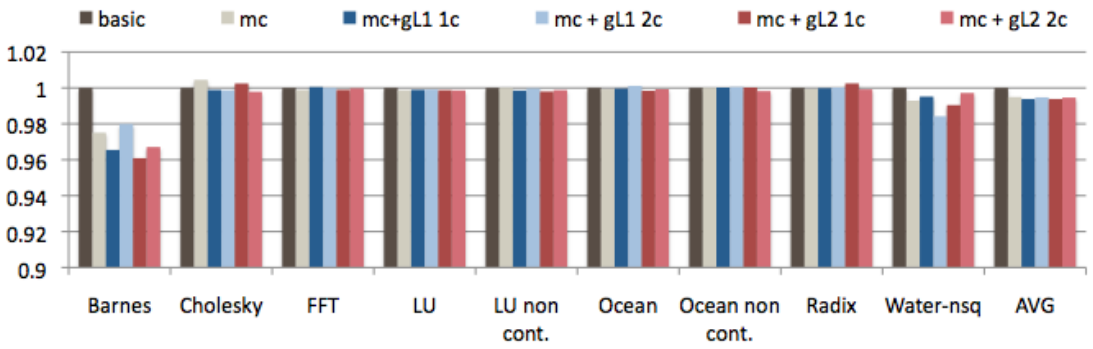


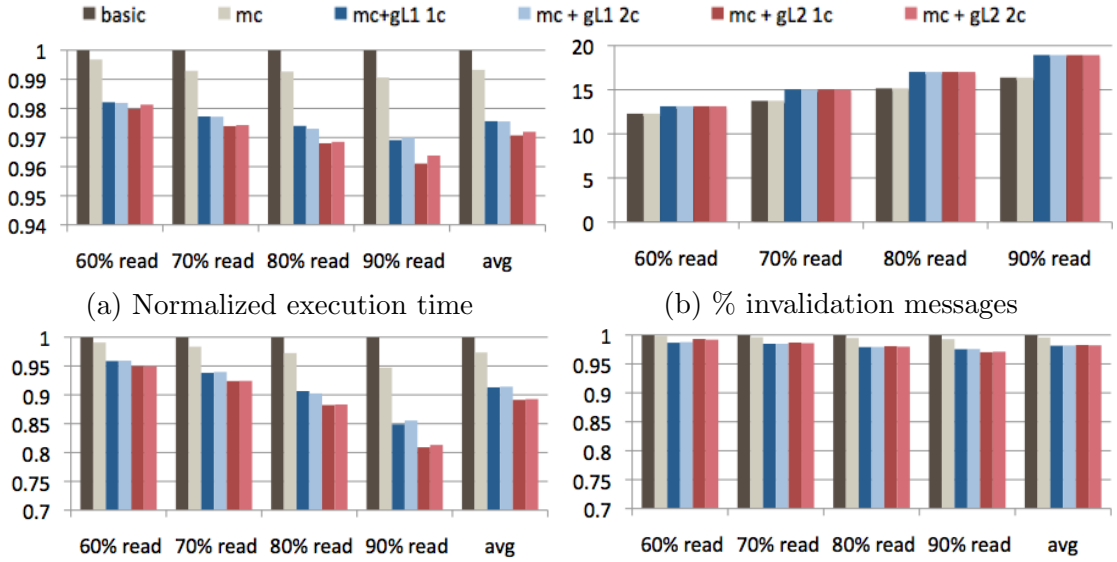
FIGURE 3.14: Normalized execution time (SPLASH-2 applications).

sets differ in the percentage of read and write operations (from 60% read operations to 90% read operations).

Figure 3.15.a shows the execution time for each set, normalized to the case of the basic configuration. The multicast support alone (*mc*) slightly reduces the execution time as INV messages are sent with a single message and less contention is incurred in the network. With the alternative protocols and the Gather Network, execution time is further reduced up to 4%, depending on the set of traces.

Figure 3.15.b shows the percentage of invalidation messages over the total number of messages for each set. The improvement in execution time is tightly coupled with the percentage of invalidations. The more invalidation messages are sent, the higher the benefits obtained by the GN. The percentage of invalidations grows with the percentage of read operations in the traces, since each write operation has to invalidate more sharers, so the performance improvement due to the GN becomes more evident with traces with a high percentage of reads.

Figure 3.15.c shows the average store miss latency normalized to the basic configuration. Again, the multicast support combined with the control network helps in lowering the store miss latency up to 20%. In particular, when the L1 nodes send the invalidation



(a) Normalized execution time

(b) % invalidation messages

(c) Normalized store miss latency

(d) Normalized load miss latency

FIGURE 3.15: Evaluation results with synthetic access traces.

messages (*mc+g L1*), up to 15% reduction in write miss latency is achieved. When the L2 nodes take care of invalidations (*mc+g L2*) an extra reduction is achieved obtaining up to 20%.

Since the directory-based protocol only uses the GN to collect the ACKs in store misses, its effect on the load miss latency is negligible. This effect can be seen in Figure 3.15.d. When using the alternative protocols, the load miss latency is slightly reduced (2%) with respect to the basic protocol. It should be noted also that the latency of the GN has a very low impact in the results. Execution time, miss load latency, and miss store latency, are practically the same when the control network has a delay of one (mc + gnL1 1c and mc + gnL2 1c) or two cycles (mc + gnL1 2c and mc + gnL2 2c).

Optimizing only one case out of the four exposed in Figures 2.8 and 2.9, the impact of the GN in systems which implement a directory-based protocol is quite application-dependent: if the application generates a high percentage of write accesses on widely shared variables, the GN will be effective, otherwise it is not used.

3.5.2 Hammer Protocol with GN

As expected, the GN is more effective if it is used in a system which implements a broadcast-based protocol like Hammer. As explained in Chapter 2, the LLC in a

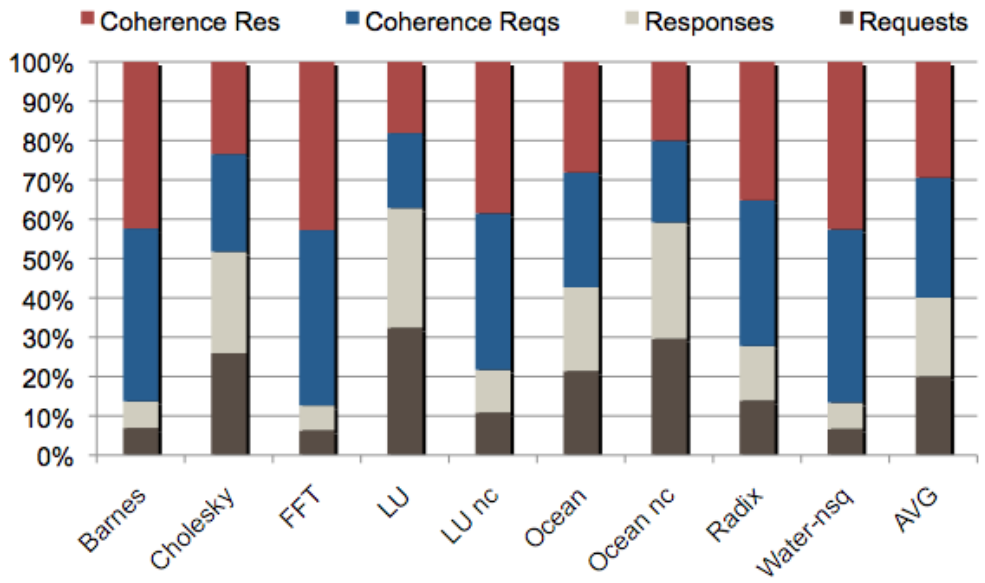


FIGURE 3.16: Breakdown of network messages in Hammer protocol.

broadcast-based protocol injects a broadcast in the NoC every time a request is received (except for the case of a read miss on a shared block) and each node except for the owner of the block must send an acknowledgement to the requestor, so acknowledgements are much more common in broadcast-based protocols than in directory protocols. Indeed, the results obtained running various SPLASH-2 applications on Graphite + gMemNoCsim with a Hammer-like protocol show that the percentage of ACKs over the total number of messages is 30% on average (reaching 43% for some application like Barnes, FFT and Water-nsquared), as shown in Figure 3.16, where broadcasts are labeled as Coherence Reqs and acknowledgements as Coherence Res.

A broadcast-based protocol like Hammer has been implemented in gMemNoCsim and evaluated with three different network configurations:

- Hammer: a basic configuration with no hardware broadcast support and no GN
- Hammer BC: a configuration with NoC-level broadcast support and no GN
- Hammer BC GN: a configuration with broadcast support and the GN

The results obtained with these configurations have been compared also to those of a full-map directory protocol with a regular NoC. Network and cache parameters are shown in Table 3.6; we assume a combinational GN with latency of 2 cycles.

Routing	XY	Coherence protocol	Hammer / Directory
Flow control	credits	L1 cache size	64 + 64 kB (I + D)
Flit size	8 byte	L1 tag latency	1 cycle
Switch model	4-stage pipelined	L1 data latency	2 cycles
Switching	virtual cut-through	L2 bank size	512 kB
Buffer size:	9 flit deep	L2 tag latency	2 cycles
Virtual channels:	4	L2 data latency	4 cycles
GN delay	2 cycles	Cache block size	64 B

TABLE 3.6: Network and cache parameters (GN with Hammer protocol).

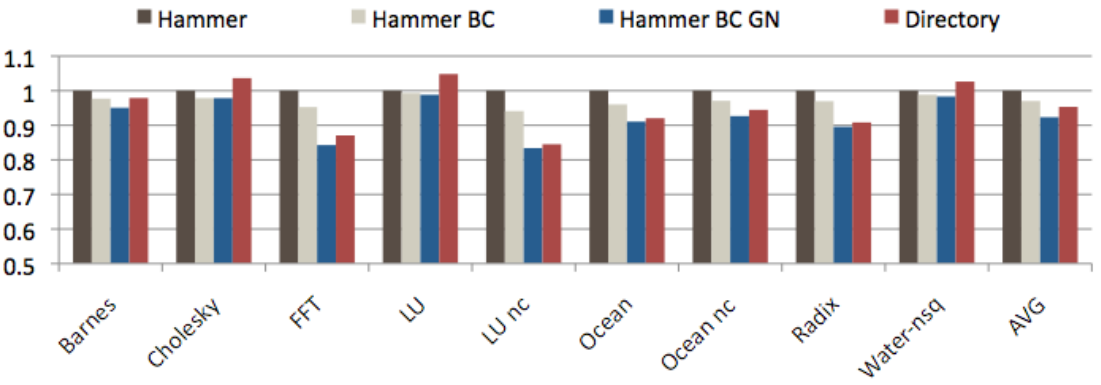


FIGURE 3.17: Normalized execution time (Hammer).

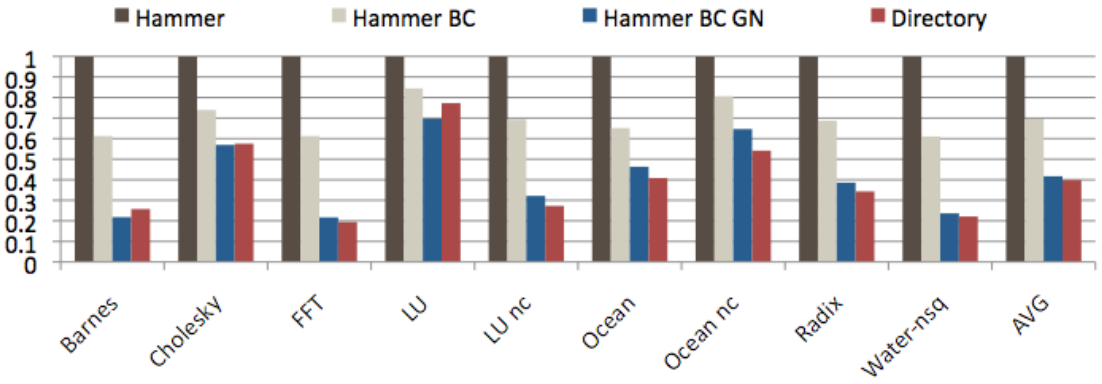


FIGURE 3.18: Normalized number of injected messages (GN signals are not included).

Figure 3.17 shows the normalized execution time of various SPLASH-2 applications with the four configurations listed above. Without further support at network level, Hammer performs worse than Directory due to the amount of traffic generated each time a request is sent to the home L2 bank and due to the serialization of ACKs at the requestor node's input port. Hardware broadcast support reduces the execution time, but still the performance of Directory is better than those of Hammer BC. The GN further reduces the execution time, reaching an average execution time for Hammer BC GN which is 8% lower than Hammer and 3% lower than Directory.

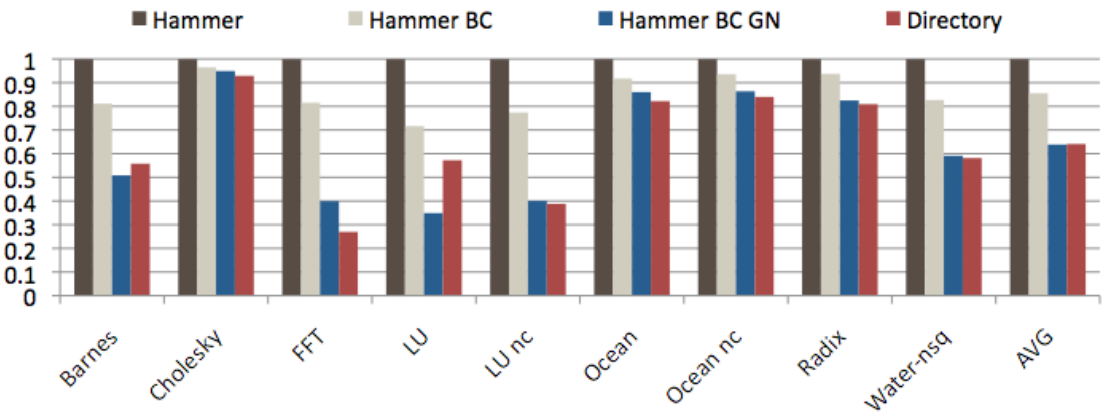


FIGURE 3.19: Normalized store miss latency (Hammer).

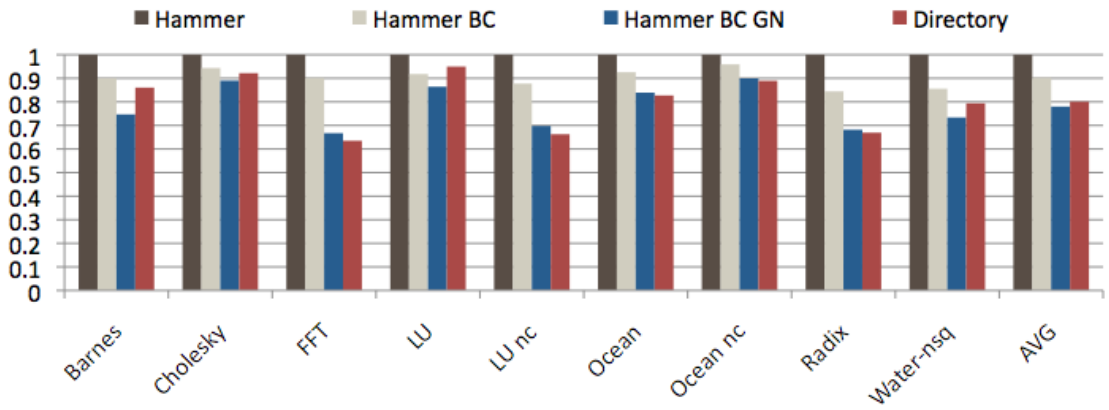


FIGURE 3.20: Normalized load miss latency (Hammer).

Network traffic is also drastically reduced. As shown in Figure 3.18, combining NoC broadcast support and the GN the number of injected messages in Hammer BC GN is reduced up to 60% on average and 80% for some applications. This means that Hammer BC GN reaches better performance than Directory, clearing the area/traffic tradeoff: typically, directory-based protocols have a high area overhead (due to the sharing code) but generate low traffic, while broadcast-based protocols have a very low area overhead and generate much more traffic. The GN allows Hammer BC GN to overcome the performance of Directory with a lower chip area overhead an generating the same amount of traffic.

The impact of the GN on store and load miss latency when using Hammer protocol is higher than that shown in Figures 3.15.c and 3.15.d for Directory protocol. As shown in Figure 3.19, the combined effect of broadcast support and the GN reduces the store miss latency of Hammer protocol by 40% on average (and up to more than 60% for some applications). The impact on load miss latency is lower, although still noticeable: as

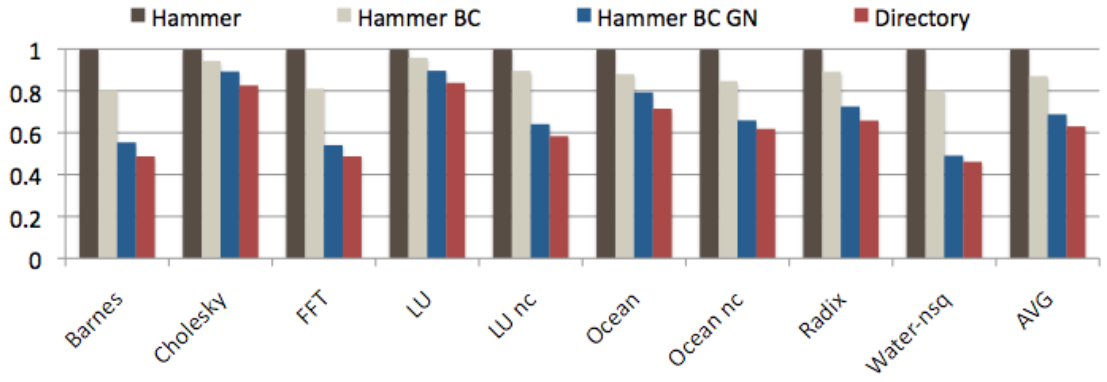


FIGURE 3.21: Normalized NoC dynamic energy.

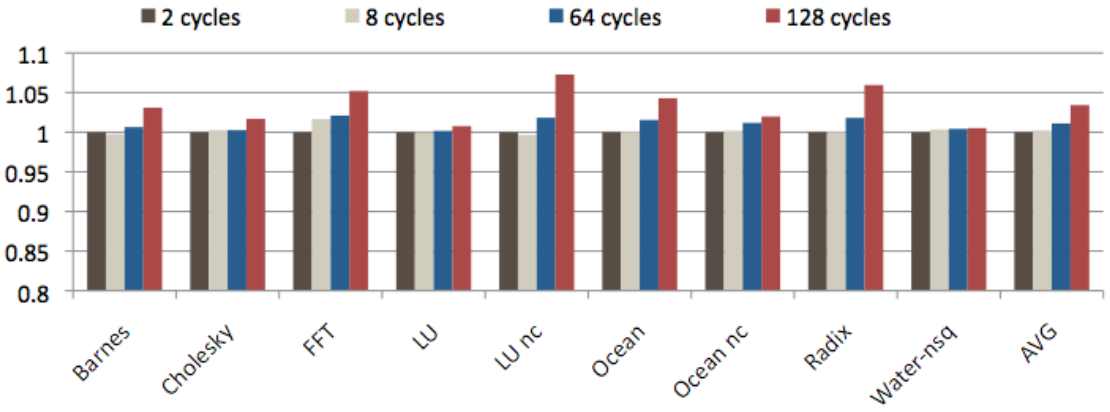


FIGURE 3.22: Normalized execution time with different GN delays.

shown in Figure 3.20, the load miss latency is reduced by 20% on average, due to the fact that a percentage of read requests (those received for a shared block) are managed by the L2 cache without broadcasting any message to L1 caches.

Figure 3.21 shows the NoC dynamic energy consumed during the entire execution of each application; Orion 2 has been used to calculate the energy consumption of the regular NoC, while the power requirements of the GN sequential module were obtained with Cadence Encounter. The energy consumed by the GN however is a very small fraction of the total NoC energy. The hardware broadcast support and the GN combined are able to reduce the dynamic energy by 32 % on average, getting close to the values obtained with Directory protocol.

Figure 3.22 shows the impact of the GN delay on the system performance. The SPLASH-2 applications were run on a system with Hammer BC GN, varying the GN delay from 2 to 128 cycles. As shown, the performance is not significantly affected with delays up to 64 cycles: the average execution time increases by 1% on average. For this evaluation

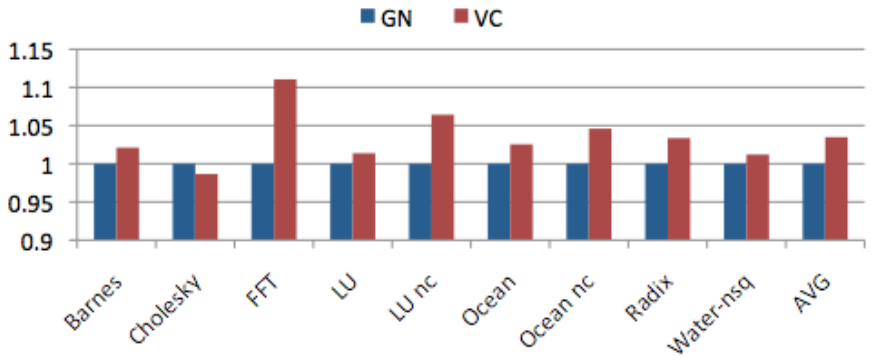


FIGURE 3.23: Normalized execution time compared to a NoC with an high priority VC for the ACKs.

we assumed a combinational implementation of the GN with a delay of 2 cycles; as exposed in Subection 3.2.4, the latency of a sequential implementation would be higher (1 cycle per hop). As shown in Figure 3.22, this latency increment would not affect the performance; Subsection 3.5.3 provides a more detailed evaluation of ACK latencies when a sequential implementation of the GN is used.

To conclude this subsection we compare the performance of a system using the GN with those of a system where a dedicated high-priority virtual channel is used to transmit the ACKs (VC). Figure 3.23 shows the normalized execution time when the two configurations are used; GN has better performance than VC since it completely relieves the NoC from the large amount of traffic due to the ACKs. Notice that the VC configuration needs more switch resources contrary to the Gather Network solution as implementing buffering for the extra VC is costlier than implementing the control logic and additional wiring of the GN.

3.5.3 Sequential Gather Network

In the previous section we assumed a combinational implementation of the GN. Now, its performance is compared to that of a sequential implementation. As explained in Subsection 3.2.4, in the sequential implementation the ACKs are transmitted through the GN hop by hop and cycle by cycle, and the wiring is no more dedicated (in the combinational implementation each wire is dedicated to a subnetwork) so there could be contention if two different ACKs must be transmitted through the same output port of a GN module at the same cycle.

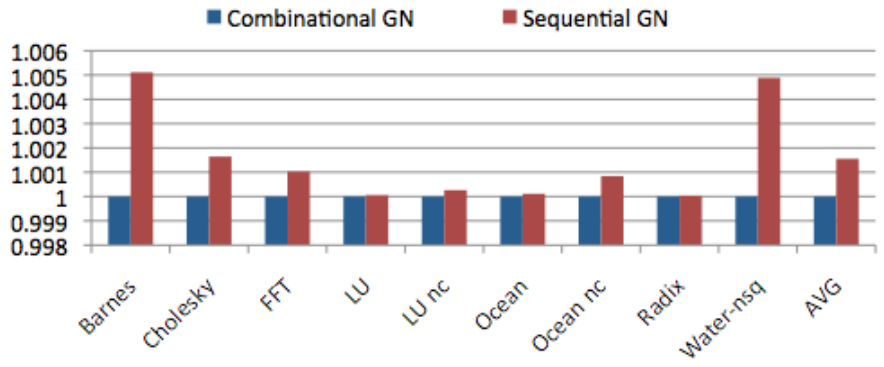


FIGURE 3.24: Normalized execution time with the two implementations of the Gather Network.

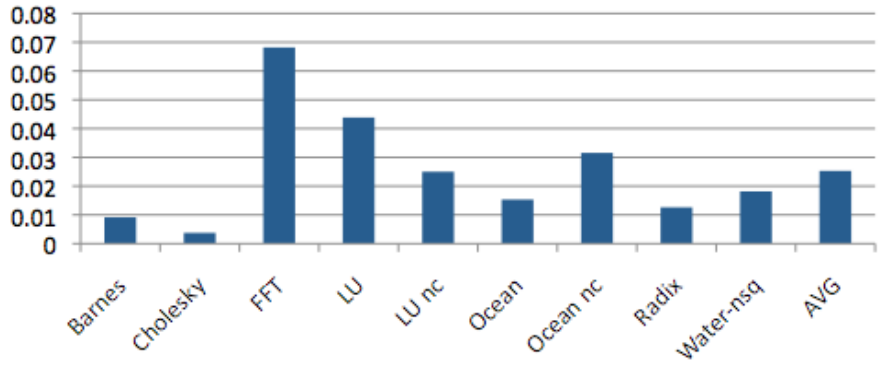
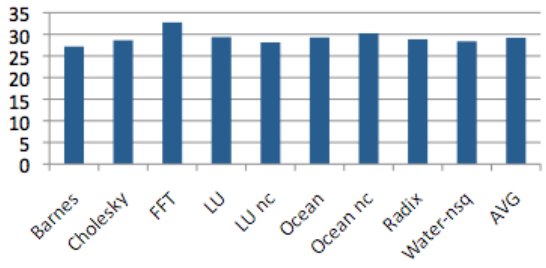


FIGURE 3.25: Number of conflicts per gather message received at destination node.

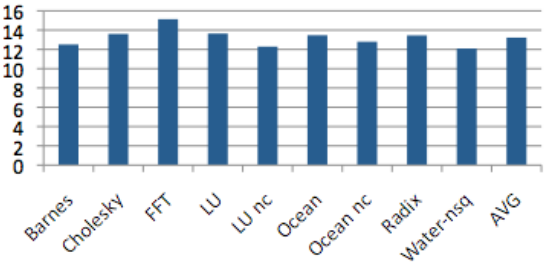
Figure 3.24 shows how the execution time is affected when the sequential implementation is used. The increased latency in delivering the ACKs has a very low impact on overall performance: the execution time increases by a 0,15% on average and 0,5% in the worst case (Barnes an Water-nsq). For some applications no performance degradation can be noticed.

The increased execution time is due to conflicts in the GN modules and to the increased latency of ACKs. Figure 3.25 shows how many conflicts occur at the output ports of all GN modules for each ACK received at the destination node. The percentage of conflicts is quite low: on average, 25 conflicts occur during the transmission of 1,000 ACKs. Notice that we are considering the mixed XY-YX mapping of Figure 3.7 to achieve a balanced distribution of ID transmissions through the different output ports of each module.

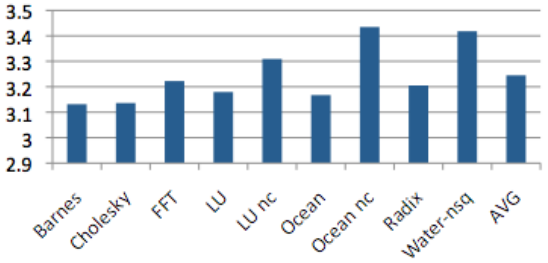
Figure 3.26 shows the average latency of ACKs with the sequential implementation, measured in three different ways: Figure 3.26.a shows the elapsed time between the triggering of the first ACK and the reception of all the ACKs; Figure 3.26.b shows the



(a) between the first ACK and the reception of the global ACK



(b) between the last ACK and the reception of the global ACK



(c) between the ACK of the furthest node and the reception of the global ACK

FIGURE 3.26: Average GN latency (sequential implementation).

elapsed time between the triggering of the last ACK and the reception of all the ACKs, and Figure 3.26.c shows the elapsed time between the triggering of the ACK by the node located farther from the requestor and the reception of all the ACKs. Notice that the last node to trigger the ACK is not always the one that is located more distant to the requestor. The latter case therefore is the fittest to be compared to the latencies of the combinational version shown in Subsection 3.2.3, since it represents the latency of the sequential GN in transmitting the ACK when the last signal is triggered. On average, the GN latency is thus increased to 3.2 clock cycles when the sequential implementation is used in a 4×4 system.

The moderate latency increment, which is well below the 128 cycles slack shown in Figure 3.22, and the low number of conflicts at each module lead to a negligible performance degration when using the sequential version instead of the combinational one.

At the expense of a slightly increased latency, the sequential module has a better scalability and enables more sophisticated uses of the GN, which will be exposed in the rest of this dissertation. From now on, we'll thus assume the sequential implementation.

3.6 Conclusions

In this chapter we presented the Gather Network, a lightweight dedicated network which can be used to transmit many-to-one acknowledgements in tiled CMP systems. Two versions of the GN have been implemented and evaluated:

- a combinational one, which has a very low area overhead and can deliver ACKs in 1 to 4 clock cycles in systems with tens of cores. The drawback of this implementation is the high wiring requirements as the number of nodes increases, since a one-bit subnetwork must be dedicated to each destination. Wiring requirements also increase if the system employs out-of-order cores: in this case, indeed, each core needs a number of subnetworks equal to the maximum number of pending cache accesses it can issue.
- a sequential one, which employs sequential modules at each switch; modules communicate by exchange the ID of the destination tile, thus lowering the wiring requirements to a logarithmic dependency from the number of nodes. This implementation has higher area overhead and latency than the combinational one, but still low compared to the regular NoC: each module indeed increases the area of the basic switch by 2.3% and is able to propagate ACKs to the neighbors in 1 clock cycle.

The GN has been used to transmit invalidation ACKs in Directory protocol and coherence ACKs in Hammer protocol. Directory protocol requires some modification to employ the GN for invalidation ACKs, since a subnetwork must be configured before it is used to enable exclusively the inputs of the sharers. These modifications, which add indirection stages, and the low number of invalidations generated by SPLASH-2 applications, result in low performance improvements for the set of applications we simulated. Results with synthetic traces with an high percentage of invalidations, however, show a higher performance improvement.

Hammer protocol on the other hand does not require the configuration phase, and, due to its the broadcast nature, ACKs represent 30% on average of the total messages when SPLASH-2 applications are executed. Combining the LBDR broadcast support and the GN we achieved to lower the miss latencies (and then the execution time),

the NoC traffic and the NoC energy, reaching values which are close to those obtained with Directory protocol. With the GN, Hammer protocol is thus able to overcome the area-traffic overhead tradeoff, reaching performance comparable to those of Directory protocol without having the area overhead of the directory structure.

The GN sequential module can be extended to transmit also unicast (one-to-one) ACKs after a few simple modifications. This allow to use the GN also to transmit, for instance, writeback acknowledgements sent by the LLC to a private cache which is replacing its line, or directory unblock messages; this also allow to use the GN and Directory protocol without any modification: the ACKs sent by the sharers are sent as unicast messages rather than be gathered as we assumed in this chapter. A study about this extension of the GN module has been published recently [62].

In the rest of this thesis the GN is used as a substrate to speedup the delivery of all-to-one ACKs generated by the LLC-level technique described in Chapter 4. This technique indeed requires to broadcast requests and acknowledge those broadcasts; since all nodes participate in the acknowledge phase, the logic to configure the GN is not needed. The sequential modules, however, need some modification to transmit flags associated to the gathered ACK.

Chapter 4

Runtime Home Mapping

In this chapter we present an optimized strategy to allocate memory blocks on LLC banks. The idea is to put memory blocks close to the L1 caches that consume those blocks. This is done at runtime with an algorithm implemented on the memory controller of the chip. In this chapter we also adapt the Gather Network and propose further optimizations to make the whole system more efficient and scalable.

4.1 Introduction

The reference system considered throughout this thesis uses shared LLC banks, since this is the most common configuration for the LLC in CMP systems. As explained in Chapter 2, the LLC bank that hosts a block is called the *home* bank, and the common policy to decide which LLC bank is the *home* for a block is static mapping. A banked cache which uses static mapping is called S-NUCA: the address space is divided in subsets and all the blocks of a subset are statically mapped to a bank. This policy is very simple to implement but can be inefficient as blocks may be mapped to banks which are far away from L1 requestors.

Another option is to perform the mapping dynamically (D-NUCA): each subset of blocks is mapped to a group of banks, or bank set, and blocks can migrate within a bank set to move as close as possible to the requestor's tile. This policy has lower miss latencies but is more complex to implement. Furthermore, the process of finding a block within a bank set leads to a tradeoff between access time and NoC traffic since all the banks of a bank set must be accessed, leading to either high latency (sequential search) or more traffic (parallel search).

Figure 4.1 shows the average distance of accessed L2 banks by their L1 requestors in a 4×4 tiled configuration. The figure shows results when using private L2 caches, shared L2 caches with S-NUCA approach and shared L2 caches with D-NUCA approach, where each bank-set is a column of four banks. As expected, private caches achieve the lowest hop count, thus having a reduced access latency. However, this is achieved by highly restricting the L2 cache capacity to a single bank per core. In contrast, neither S-NUCA (due to its static nature) nor D-NUCA (due to its static bank set configuration) are able to achieve reduced hop counts. The goal of the technique described in this chapter is to achieve similar results in hop distance to the private configuration, keeping the high capacity of the shared configuration of L2 banks.

The technique is called Runtime Home Mapping (RHM), and is a new dynamic approach where the LLC home bank is determined at runtime in hardware by the memory controller (MC). While in D-NUCA the mapping is partially static, in RHM a block can be mapped to any L2 cache bank, aiming to reduce the distance from L1 requestors to the L2 home bank. Figure 4.2 shows the basic steps of the RHM mechanism comprising

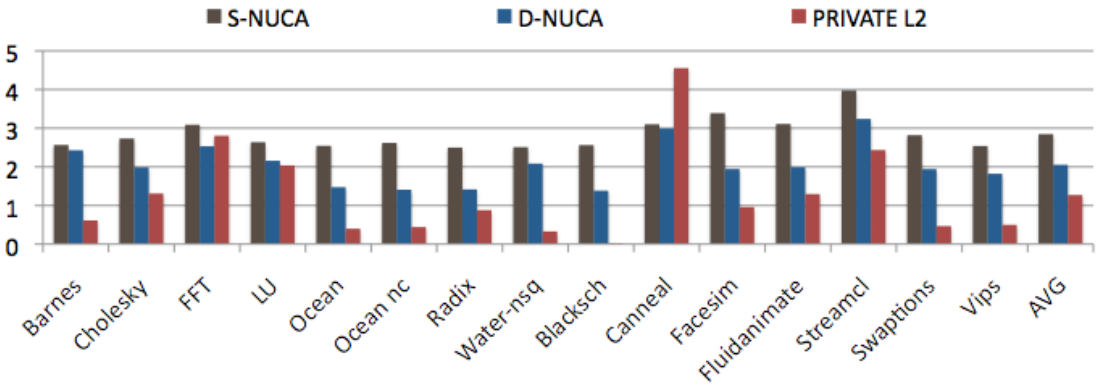


FIGURE 4.1: Average distance of L2 banks to their L1 requestors for different mapping policies.

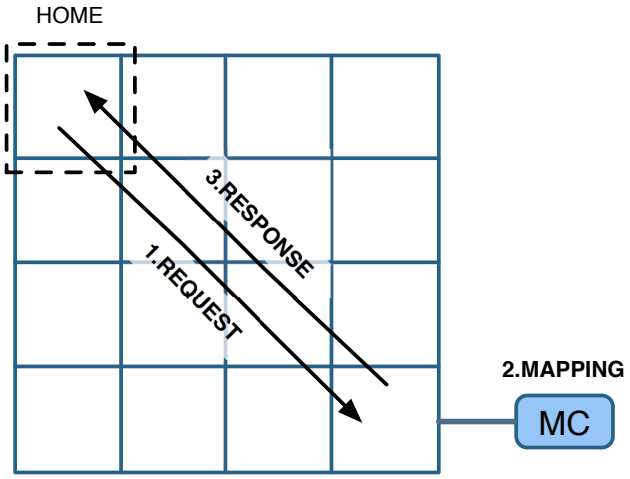
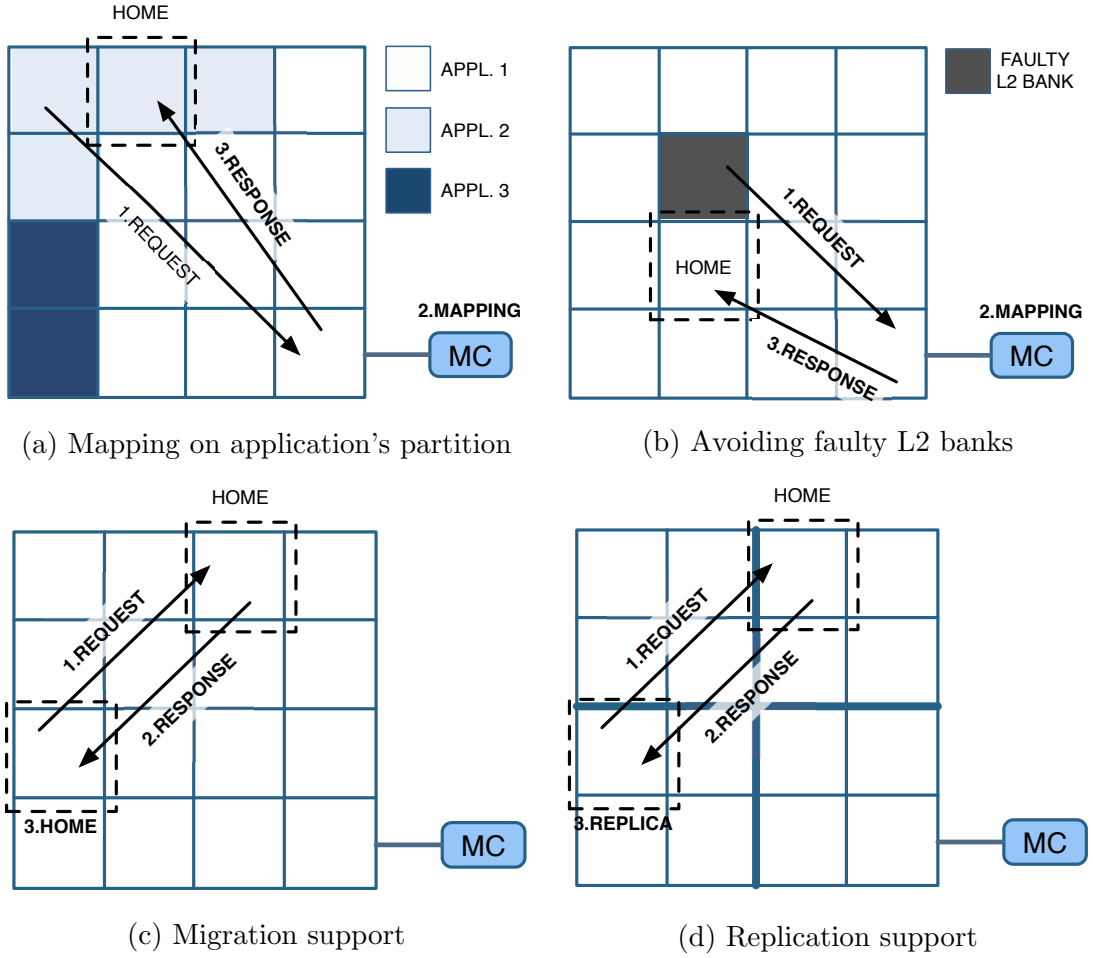


FIGURE 4.2: RHM example (block is mapped on requestor's tile).

the request of a block to the MC, the selection of the home L2 bank while fetching the block from memory, and the forwarding of the block to the home L2 bank and to the requestor. In this example, the selected home is the local L2 bank of the requestor's tile.

RHM can be viewed as a D-NUCA configuration with a single bank set which includes all the L2 banks. However, RHM differs from D-NUCA in the sense that it enables further optimization opportunities such as partitioning/virtualization, thread migration, and fault-tolerance. As an example, in Figure 4.3.(a) three applications are mapped on different resources of the chip. The MC, by using RHM, is able to map memory blocks belonging to an application to the L2 banks mapped to that application, thus guaranteeing network-level and memory-level partitioning. In this example, the selected L2 home bank is not located at the requestor's tile. Also, Figure 4.3.(b) shows the case where one L2 bank has been disabled possibly due to some manufacturing defects. In



(a) Mapping on application's partition

(b) Avoiding faulty L2 banks

(c) Migration support

(d) Replication support

FIGURE 4.3: Runtime Home Mapping. Different scenarios.

that situation, the MC, by using RHM, is able to filter out the failed bank and thus to map blocks only to functioning L2 banks. In Figure 4.3.(c) we can see the case of a block migration in RHM. In this case, one requestor solicits a copy of a private block, triggering the block migration process, at the end of which the L2 bank in the requestor's tile is the new home bank. To further reduce hop distance from L1 requestors to L2 home banks, RHM allows the replication of shared blocks. Figure 4.3.d shows the case where a home decides to replicate a block in another L2 bank.

All the previous examples hide two potential problems that need to be solved. The first one is the potential incoherence of blocks involved in migration or replication processes. Multiple race conditions can arise when several processors trigger load and store requests to the same memory block, so the coherence protocol must be carefully designed. This is more difficult to achieve when memory blocks can migrate within LLC banks or when multiple LLC banks can have replicas of the same block. The second problem is the

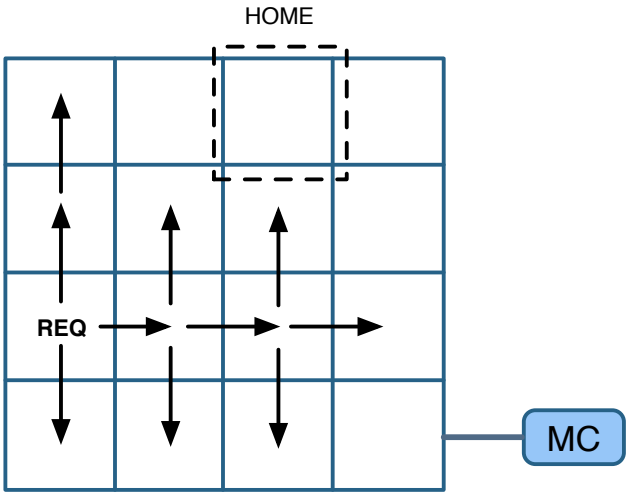


FIGURE 4.4: Home search phase.

efficiency of the coherence protocol, in particular of the home search phase and of the migration and replication processes. Indeed, the principal source of inefficiency is that L1 caches do not know which L2 bank is the home for a particular block. Since the *home* bank is not known a priori, a search must be performed each time an L1 miss occurs. This is shown in Figure 4.4, where a local miss triggers a broadcast action to search the home bank for the requested block. This problem affects the network infrastructure providing connectivity support to RHM (the underlying on-chip network) and requires NoC-level measures to prevent the search phase to become a bottleneck.

More in detail, three different NoC mechanisms can be used to optimize the RHM search phase:

- bLBDR tree-based broadcast mechanism to broadcast the search message, used in conjunction with the Gather Network to collect the acknowledgements of L2 banks; the GN must be extended to support basic signaling between tiles.
- Parallel access of L2 tags: the tag array of L2 banks is tightly coupled to the NoC router design. At the same time the broadcast message enters the router, the tag array is accessed.
- A router mechanism aimed to reduce the broadcast messaging. On an L2 tag hit, the broadcast message is cancelled, thus reducing traffic by chopping broadcast branches.

These optimizations at NoC level provide an efficient support to implement the cache-level optimizations of RHM:

- Dynamic mapping of blocks to the L2 banks performed by the memory controller during an off-chip memory request.
- Migration and replication of cache blocks to correct the initial placement.
- Tight coupling of the block search phase and the coherence actions in broadcast-based protocols.

The rest of this chapter is organized as follows: Section 4.2 describes the basic RHM mechanism and the modifications needed to adapt the Gather Network to transmit basic signaling together with the global ACK; Section 4.3 describes the block migration and replication mechanisms, how RHM can be further optimized when the system implements a broadcast-based coherence protocol and the parallel access to L2 tags and to the router pipeline to speedup the search phase and allow the cropping of useless broadcast branches; Section 4.4 provides the evaluation results of RHM and its optimizations with both directory-based and broadcast-based coherence protocols. Conclusions are drawn in Section 4.5.

4.2 Runtime Home Mapping

Runtime Home Mapping (RHM) aims to map blocks to L2 banks at runtime, in order to allocate them as close as possible to the requesting cores, preferably at the L2 bank of the requestor's tile. The mapping is performed by the MC each time it receives a request.

Figure 4.5 shows the global overview of the RHM protocol. In case of an L1 miss, a request is sent to the local L2 bank in the same tile. If the block is found on the local L2 bank (the local bank is the *home* for that particular block), coherence actions are triggered according to the coherence protocol and the block is sent to the L1 cache. The specific coherence actions are detailed later in this section.

On a miss on the local L2 bank, a broadcast is sent to all other L2 banks. When an L2 bank receives this broadcast request, it checks its tag array. In case of a hit, it completes

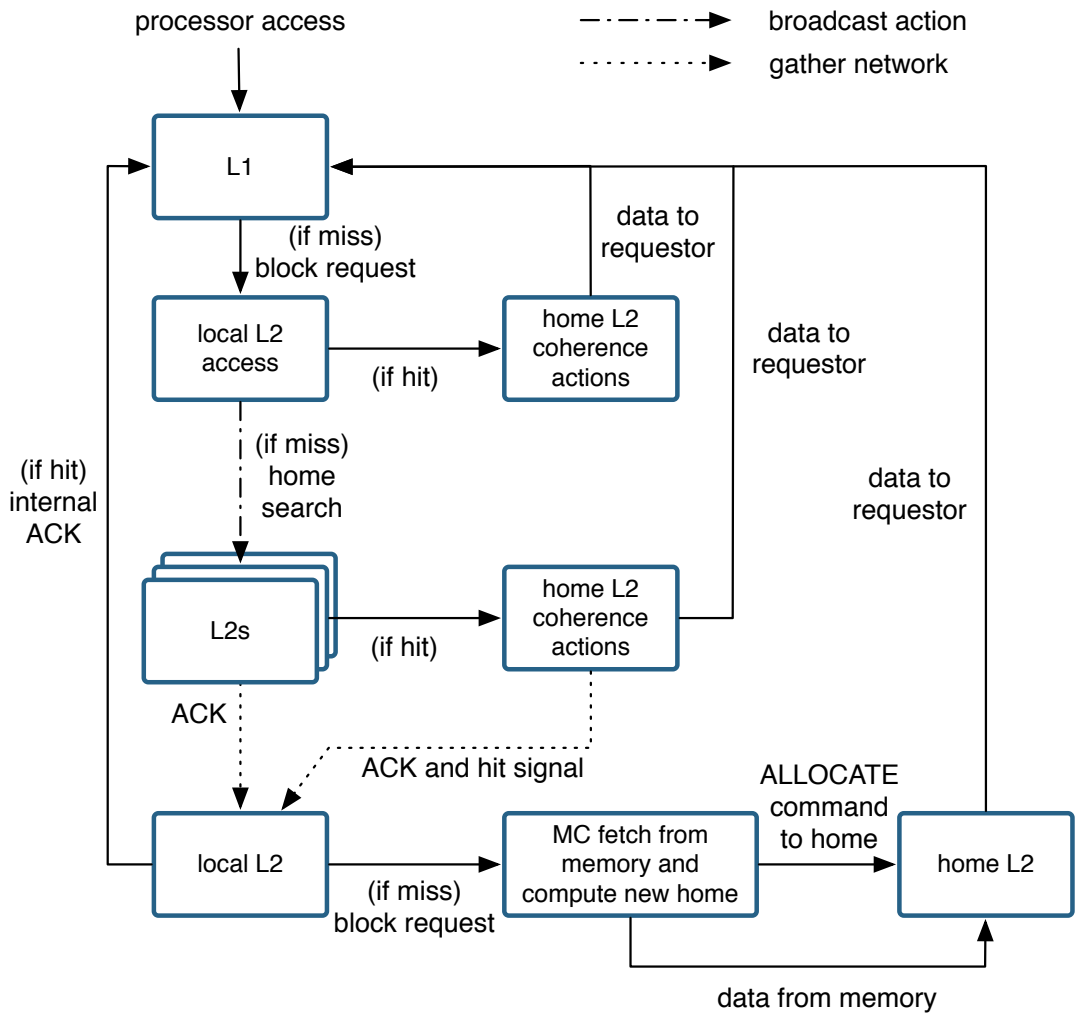


FIGURE 4.5: RHM global overview. From processor access to MC access.

the coherence actions specified by the coherence protocol, sends the data back to the L1 requestor and a *hit* signal to the L2 bank that originated the broadcast. Furthermore, all the L2 banks must acknowledge the reception of the broadcast message to the L2 bank which issued the broadcast. When the local L2 bank has received all the ACKs it checks the *hit* signal. If the *hit* signal has not been received it means the block is not cached on chip, so a request is sent to the memory controller (MC), which in turn fetches the block from main memory. If the *hit* signal has been received it means the block is on its way to the L1 requestor. Thus, an internal ACK signal is sent to notify the local L1 that the search phase is complete and a new request can be sent to the local L2 as soon as it receives the requested data to complete the current request.

Upon receiving the request from the L2 bank, the MC triggers the access to main memory to fetch the block. Meanwhile, it computes which L2 bank will be the home for

the incoming block (the mapping policy and algorithm are described in Section 4.2.3). Once an L2 bank is chosen as the *home* for a particular block, the MC notifies the bank so it can start replacing a cache line, if needed, and allocate a new line while the MC is still waiting to receive the block from main memory. When the block is received at the MC, it is sent to the chosen *home* L2 bank, which in turn will forward the block to the requestor L1.

Upon a hit, either on the local L2 bank, or on a remote L2 bank, RHM follows the typical MESI protocol. Figure 4.6 shows the details of the coherence actions at the home L2 bank when a read request is received.[1] For read accesses (left hand side of the figure), different actions are performed depending on the current state of the block in L2. If the block is in *private* state, then the request is forwarded to the *owner* L1 which in turn sends the block to the *requestor*. The L2 home updates the sharing list accordingly and the new state of the block (*shared*). If the block is in *cached* state, then it means there is only one copy of the block and lies in the home L2 bank. Thus, the block is sent to the requestor and the state is changed to *private*. If the block is in *shared* state, then the requestor is added to the sharing list and the L2 bank sends the block to the requestor.

Similar actions are taken when the access is a write operation (Figure 4.7). In this case, if the block is in *private* state, the request is forwarded to the *owner* which in turn sends the block to the requestor. The block is invalidated in the owner. Also, the owner pointer in the L2 bank is updated. If the block is in *cached* state (only one copy in the chip), then is simply forwarded to the requestor and the state and owner pointer are updated. If the state of the block is *shared* then a multicast message is sent to the sharers in order to invalidate their copy. Sharers in turn send an acknowledgment to the requestor. In addition, the L2 bank sends the block to the requestor and changes the state back to *private*.

In addition to the previous basic coherence actions, RHM supports migration and replication of blocks. In particular, blocks in *cached* or *shared* state for *read* requests and blocks in *private* state for *write* requests can migrate. On the other hand, blocks in *shared* state for *read* requests can be replicated over the chip. In those cases, and when a given threshold is reached (the migration and replication processes are described in

[1]The figure, for the sake of description, does not show the additional states and messaging needed to solve the race conditions. They have been carefully analyzed and solved in the protocol implementation in gMemNoCsim. The complete protocol is shown in the Appendix.

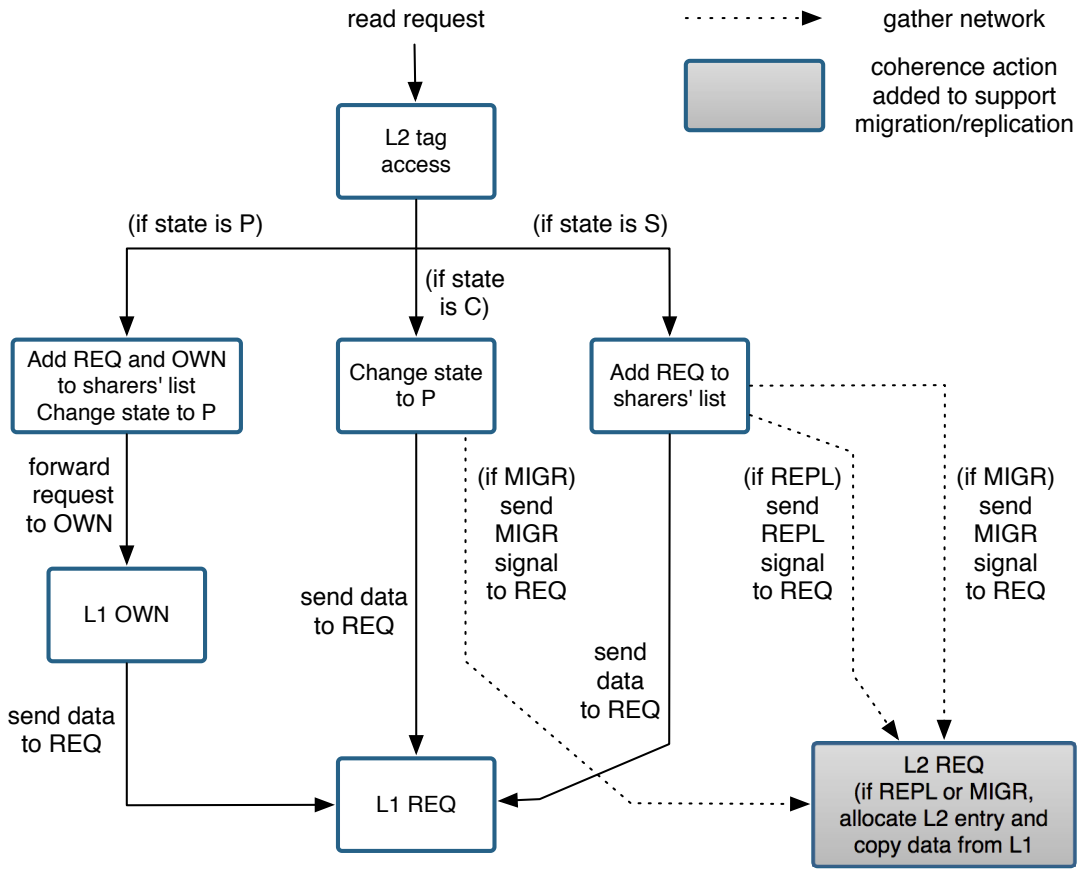


FIGURE 4.6: RHM coherence actions (read request).

Section 4.3), the L2 bank triggers a signal (either MIGR or REPL) to the L2 bank that originated the broadcast request (the one in the same tile of the L1 requestor). When the block is received by the L1 requestor, and if the signal has been received, the block is also copied on the L2 bank. Further descriptions are given in Section 4.3.

From Figures 4.5, 4.6 and 4.7 we can deduce the network latency when accessing blocks. If the block is mapped in the L2 bank in the local tile then no access to the network is made. This will be the frequent case as the MC will try to map most of the blocks to the requestor's tile. However, when the block is mapped in an L2 bank on a different tile then different accesses to the network will be made. First, a broadcast to find the L2 bank. Then, all the L2 banks sending an ACK signal to the requestor. Also, the L2 home probably will send new messages to other nodes to manage the request (as in Figures 4.6 and 4.7) and will finally send a hit signal to the local L2 bank and the block to the requestor. In case the block is not mapped on cache then the MC is accessed and a new L2 home is computed. This again means more messages through the NoC.

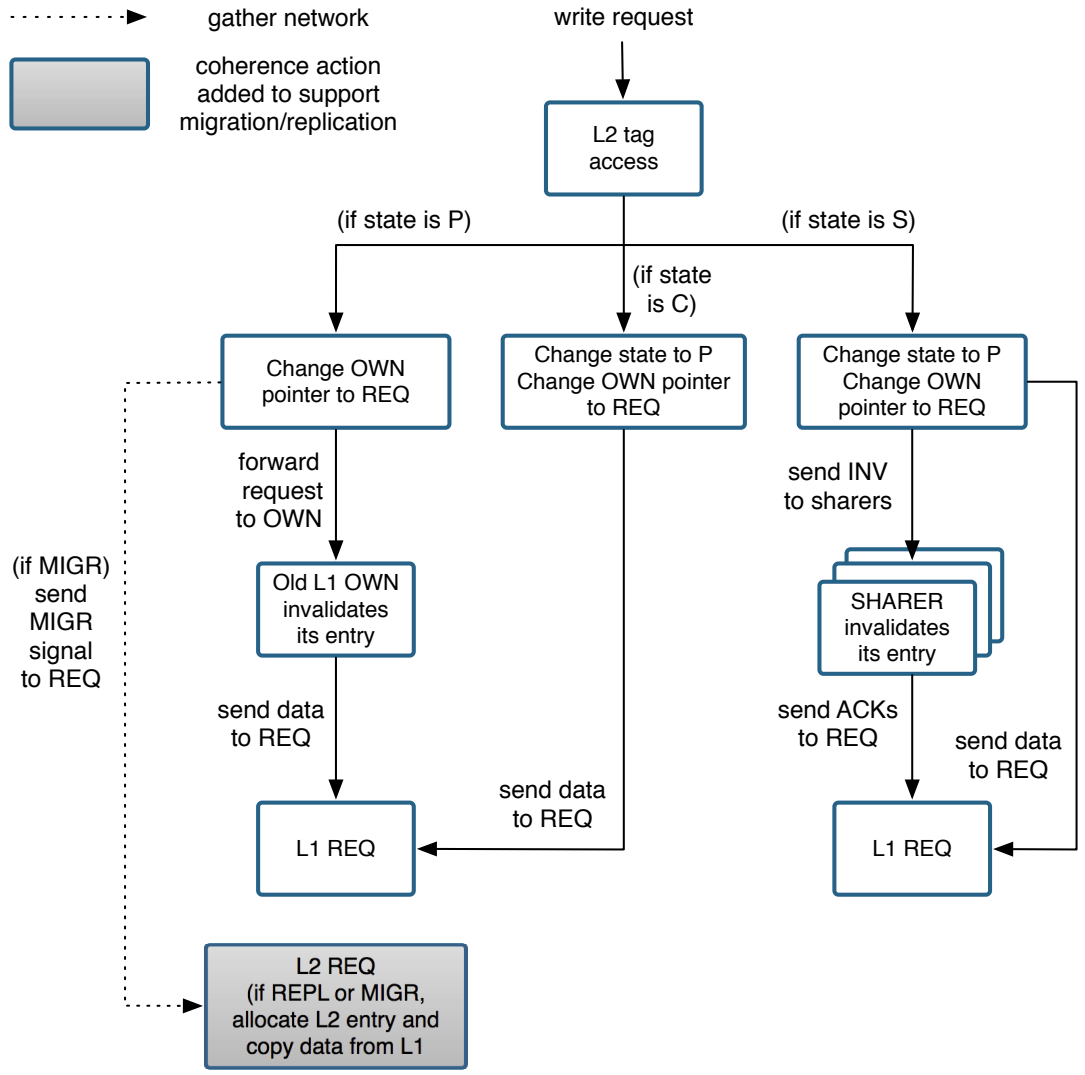


FIGURE 4.7: RHM coherence actions (write request).

The L2 home search policy described above has high network resources demand: every time a request misses in the local L2 bank, a broadcast is issued. Also, all other banks must answer to the broadcast with an ACK or with the data. The network requirements of the broadcast phase can be attenuated by using the hardware tree-based broadcast mechanism already used to reduce broadcast traffic in Chapter 3: a broadcast is sent as a single message, that replicates at the switches to reach every L2 bank. This reduces NoC traffic and eliminates the serialization of multiple copies of the same request (one per destination). The GN described in Chapter 3 can be used here to collect the ACKs generated during the home search phase. In addition, different signals (*hit*, *MIGR*, and *REPL*) are sent from the home L2 to the L2 bank in the requestor's tile. These signals need to reach the requestor at the same time ACK signals do. To reduce the network

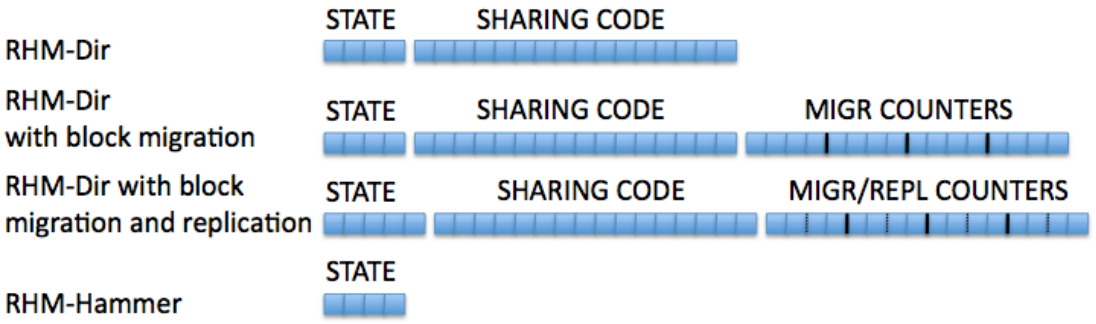


FIGURE 4.8: Control info required for each version of RHM.

impact of all those signaling commands, the GN can be enhanced to transmit those signals as flags of the global acknowledgement. Section 4.2.2 describes the modifications needed at the GN modules to support RHM.

Figure 4.8 shows the control information required at each L2 cache line by the coherence protocol. Different versions of RHM will be described in the following sections: a version which uses a directory-based coherence protocol (RHM-Dir), a version using a broadcast-based coherence protocol (RHM-Hammer) and two optimizations where blocks can migrate from a bank to another and be replicated in different banks; although these optimizations are implemented on the directory-based version, they are orthogonal to the coherence protocol. For each cache line, the L2 bank will keep the state of the block. Four bits are needed to encode all the possible states (the L2 state machine has less than 16 states) except for the version with block replication and migration support, which needs five bits (up to 27 states are used to correctly manage the possible race conditions). All the versions, except RHM-Hammer need the sharing code, to reflect the list of sharers of a block. Also, migration and replication support need counters to trigger the migration and replication processes. As can be deduced, the only field that grows with system size is the sharing code while the size of the counters only depends on the thresholds which are chosen to trigger the migration and replication of a block.

4.2.1 Avoiding Multiple LLC Misses

When the home bank in a NUCA cache is not statically known, race conditions may lead to the *Multiple Miss* problem: two or more different L1 caches issue a request for the same block simultaneously, those requests miss in the LLC and all the requestors send a message to the memory controller to retrieve the block; since the block can be cached

in different LLC banks, this may lead to multiple copies of the same block entering the on-chip cache and evolving independently and incoherently in each LLC bank. To avoid this problem, an additional *RETRY* signal can be sent to the requesting L2 together with the ACK. A fixed priority is assigned to the LLC banks, depending on their tile ID; when an L2 bank which has issued a request receives a broadcast request from another L2 bank with lower priority, the *RETRY* signal is sent to the requestor; once an L2 bank receives all the ACKs to a request and the *HIT* signal has not been received, the block is requested to the memory controller only in case none of the other L2 banks answered with a *RETRY* signal. If a *RETRY* signal is received, then the broadcast is issued again. In case the *HIT* signal is received, then *RETRY* signals are ignored. With this priority-based method the multiple miss problem is avoided.

4.2.2 Adapting the GN Module to Support RHM

The GN can be used to relieve the NoC from the ACKs generated during the home search phase and to speed-up their transmission. Some modifications are, however, required to the sequential module described in Section 3.2.4, since in the home search phase of RHM an ACK may include additional information such as the *HIT* signal, the *RETRY* or the *MIGR/REPL* signals. Figure 4.9 shows the basic 5-port switch with the added logic for the GN module. The additional logic basically manages what is indicated as the signal table, a set of registers which store the value of the *HIT*, *RETRY*, *MIGR* and *REPL* flags associated to the global acknowledgement sent to destination node.

The input logic (Figure 4.10) receives the IDs and four control signals: *HIT*, *RETRY*,*MIGR*, and *REPL*. The received IDs are stored in the input register, which has one bit (implemented as a flip-flop) per identifier, building the *input ID bitmap*. Thus, the bitmap register is of size n. Whenever an ID is received the associated bit is set. Also, the incoming ID and the control signals are forwarded to the central logic.

The central logic (Figure 4.11) is in charge of two actions. First, it has to AND the input identifiers with the same value coming from different input ports. This is done at the matrix of AND gates. The signals coming from the input ports are reorganized appropriately at the previous *wiring shuffling* stage. Second, the central logic has a signal table ($n \times 4$ matrix) combining all the control signals coming from the input ports. Whenever an input signal is received with its value set to one, it is stored in the

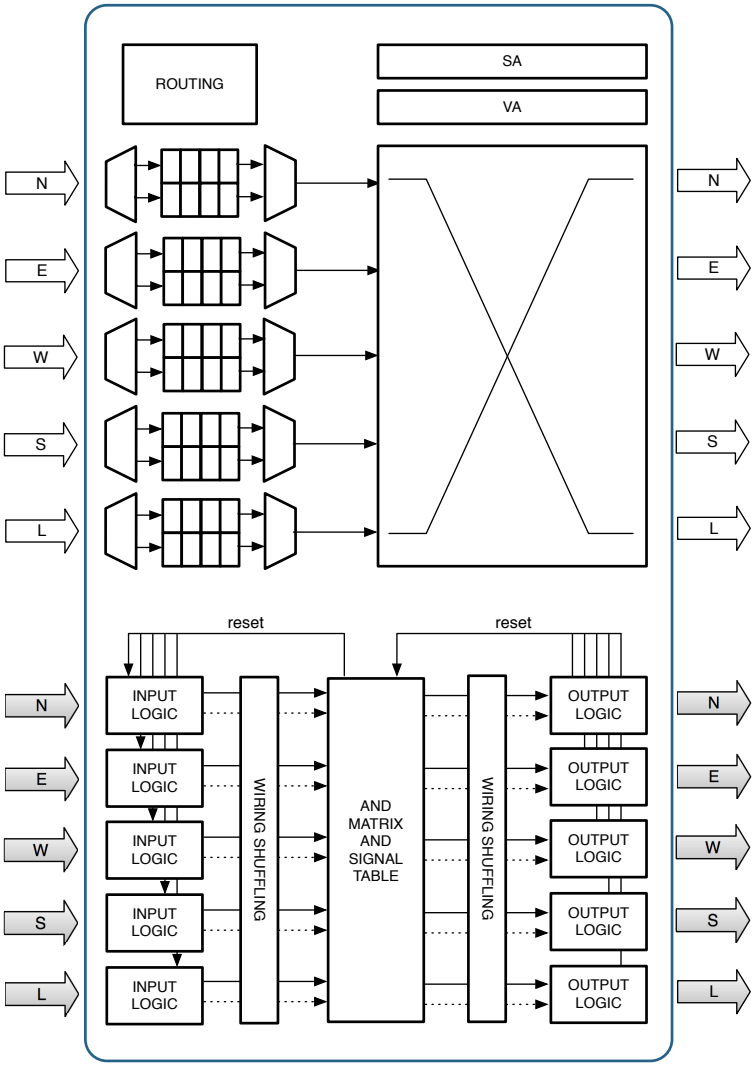


FIGURE 4.9: Switch with a GN module adapted to RHM.

signal table. Notice that input signals coming with a value set to zero do not reset the value in the signal table. We can view this table as an OR operation of all received input signals with the same IDs. The flag values at the signal table are reset when an ID is propagated to the next switch (all the ACKs for that node have been received at the current module).

Finally, each output port has an output logic (Figure 4.12). The main function of this logic is to forward IDs that have been combined by the central logic. To do this, IDs are stored in a bit vector (*output ID bitmap*) and encoded when forwarded. A priority encoder is used. The output of the encoder also selects the control signals stored in the signal table, thus the output port forwards the ID with its combined control signals.

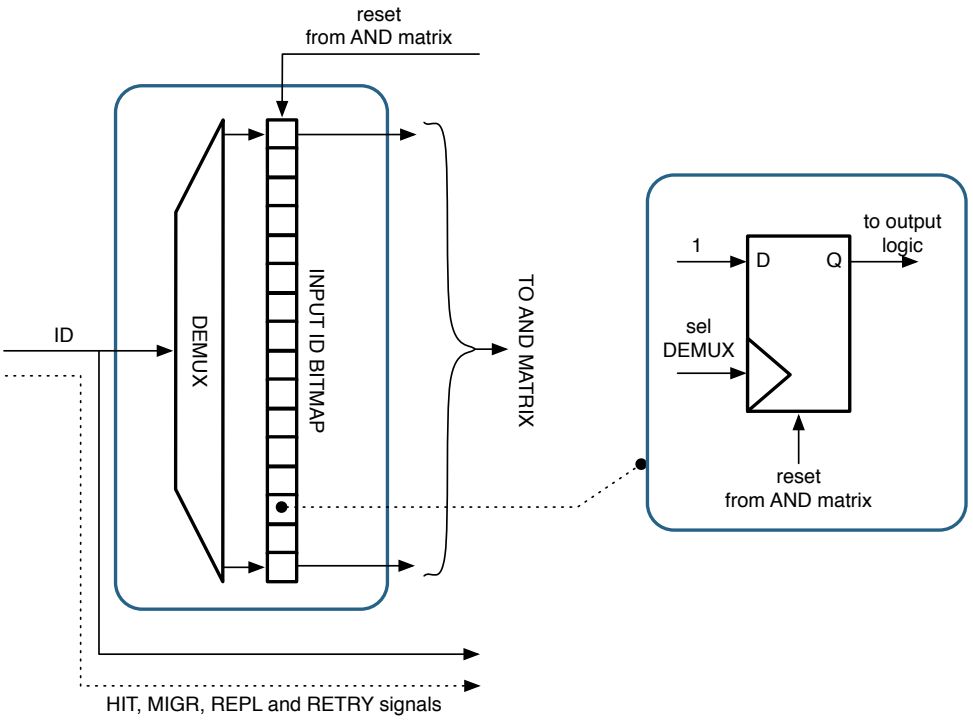


FIGURE 4.10: GN input logic adapted to RHM.

As we can see, IDs are stored both at the input and at the output ports, while signals are centralized in the central logic. Those IDs and signals need to be reset every time the ID is collected and forwarded through the output port. To do this, reset signals are triggered from the central logic to the input logic (to reset IDs) and from the output logic to the central logic (to reset the signals). The IDs at the output logic are reset whenever the ID is forwarded through the port.

Notice that the size of registers at each input port will vary from switch to switch in a 2D mesh. This depends on the mapping strategy used to collect IDs (e.g. they can be collected following X-Y patterns or Y-X patterns). In particular, we follow the mapping strategy shown in Figure 4.13. IDs associated with tiles with odd identifiers follow XY routing whereas IDs associated with tiles with even identifiers follow YX routing. This creates a balanced distribution and reduces the number of IDs per output port. In particular, the maximum number of IDs at an output port (and input port) is 9 (e.g. north output port at tile 13). In contrast, there are output ports forwarding only one ID (e.g. west output port at tile 1). Therefore, the size of the registers at the input and output logic will vary between 0 and 9. By collecting all the output IDs in a tile through all the output ports, we can see that the sum is always 15. Indeed, each tile will send one

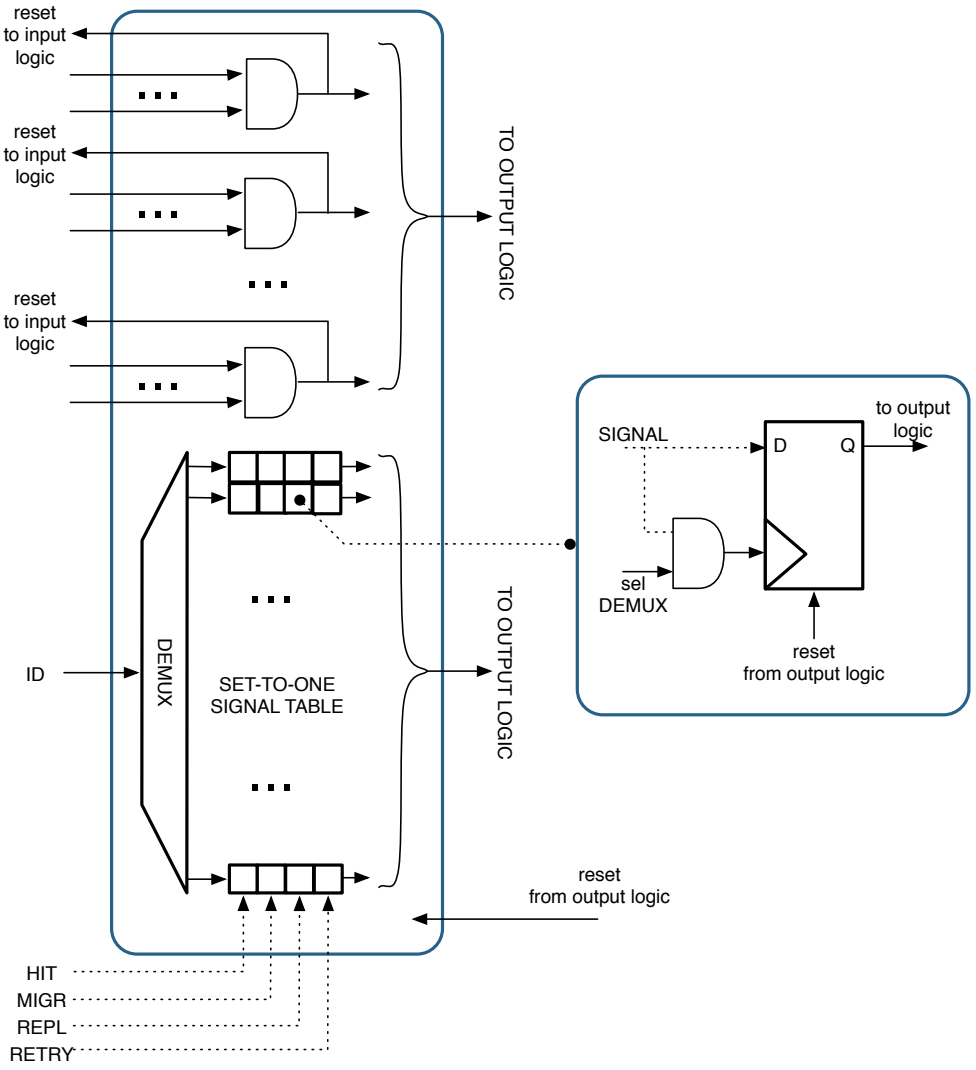


FIGURE 4.11: GN central logic adapted to RHM.

single copy of each ID except for the local one. The figure does not show the local ports. In this case, the injection port will have as many IDs as possible destinations (N-1) and one single ID at the ejection port (the one associated with the local node). Also, as the central logic will be shared by all the input ports, the number of AND gates and entries in the signal table will not vary and will be set to n. In the evaluation section we will show overhead costs of the GN implementation with support for RHM.

In the basic version of RHM, the GN is used to collect ACKs exchanged between L2 banks (and the four control signals). However, this can be extended to get more functionality. As an example, with one extra bit for IDs (duplicating the number of IDs) we can build two logical GNs which share the same physical links, one for gathering ACKs destined to L2 banks and one for gathering ACKs destined to L1 caches. This will be used to

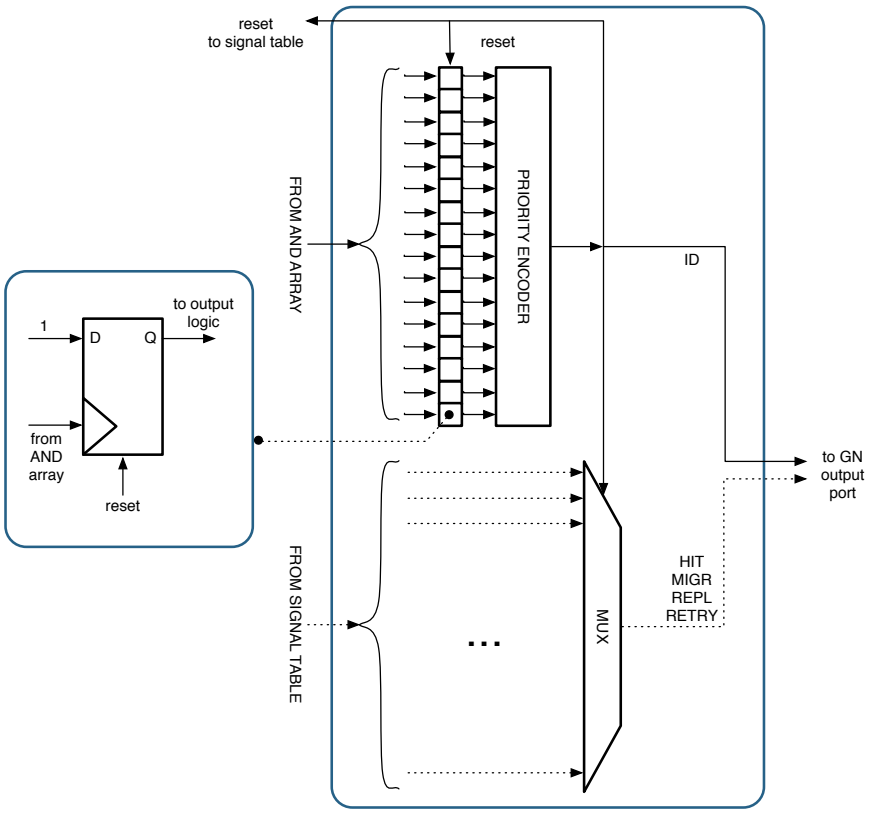


FIGURE 4.12: GN output logic adapted to RHM.

extend RHM to be used with broadcast-based protocols, as explained in Section 4.3.3. Also, out-of-order processors may trigger different request to the network, thus requiring different collecting processes through the GN. This can be supported by extending the number of IDs. Figure 4.14 shows the GN message fields we assume in this thesis. The target system is a CMP system with 16 tiles. Thus, four bits are used to address the target tile. With one extra bit we indicate whether the target component is the L2 bank or the L1 cache. In-order processors are assumed. With the ID we also send the four control signals: *HIT*, *RETRY*,*REPL*, and *MIGR*.

4.2.3 Mapping Algorithm

Each time the MC receives a request, a mapping algorithm chooses the *home* bank for the requested block. The *home* is chosen depending on the requestor's tile and current L2 banks utilization. The MC takes statistics about cache utilization, which are stored in a table (*alloc* table) with $N \times M$ entries, where N is the number of L2 cache banks and M the number of L2 sets. Each entry contains the number of allocations performed

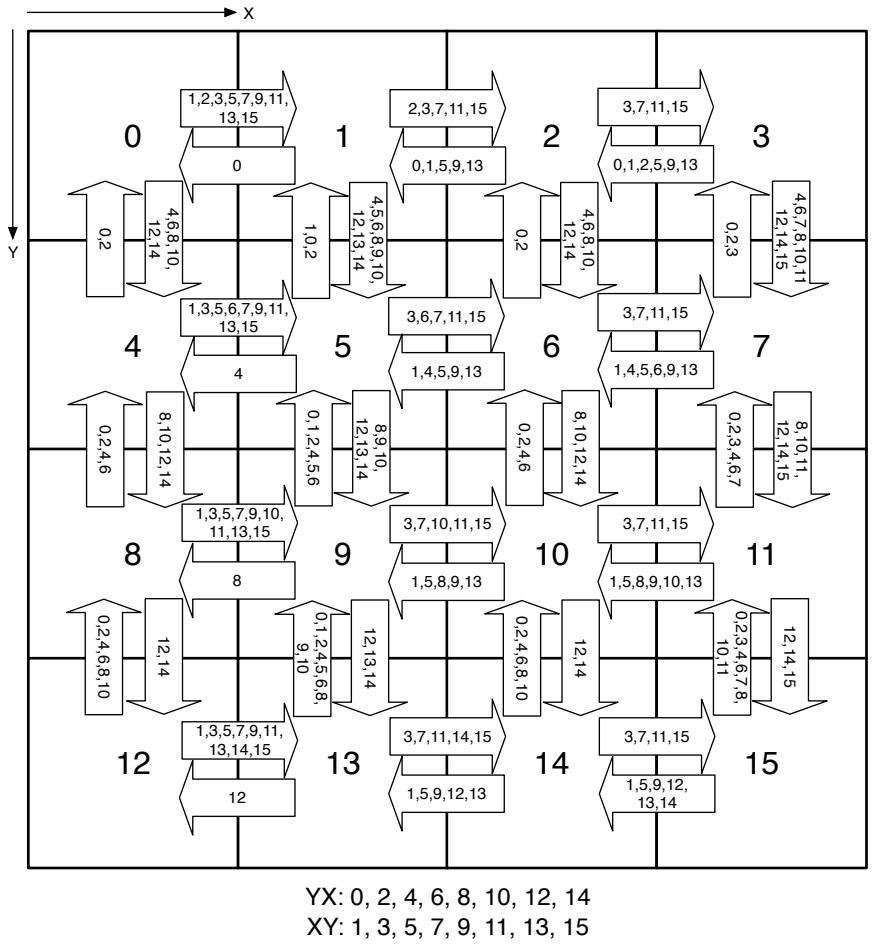


FIGURE 4.13: GCN Mapping of IDs.

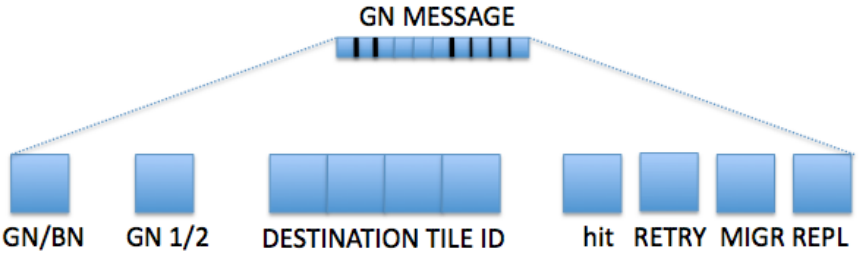


FIGURE 4.14: GN message format.

in set m of L2 bank n. If the associativity of L2 sets is Z, the table has to store at minimum $N \times M \times \log_2 Z$ bits. To balance the utilization of LLC banks when they are full, however, the table will need additional bits at each entry to represent the maximum displacement between the most loaded and the less loaded banks. For a 16×16 tile system with 16-way 256KB bank sets, the minimum memory requirements for this single table is 2KB ($m = 16$, $M = 256$, $Z = 16$). With the increased size, the table will grow to 4KB (to allow 256 allocations per set as the maximum displacement between banks).

```
int function allocate(int r, address a) {
   banklist n; bank b; set s; bank h;

   s = get_set(a);
   if(alloc[r,s]<num_ways) {alloc[r, s]++; return r;}

   for(int h = 1; h <= MaxHops; h++){
     n = BanksReachable(r, h);
     for (int i = 0; i < size(n); i++){
       b = SelectBankClockWise(n, i);
       if (alloc[b,s]<num_ways) {alloc[b,s]++; return b;}
     }
   }

   for(int h = 1; h <= MaxHops; h++){
     n = BanksReachable(r, h);
     for (int i = 0; i < size(n); i++){
       b = SelectBankClockWise(n, i);
       if (alloc[r,s] - alloc[b,s] > UtilThr) {alloc[b,s]++; return b;}
     }
   }

   alloc[r, s]++; return r;
}
```

FIGURE 4.15: Mapping algorithm performed by the MC.

The pseudocode shown in Figure 4.15 describes the simple algorithm which has been implemented.

If there is room in the set in the local L2 bank (r tile), then the *home* is the local tile of the requestor. Otherwise, the algorithm scans the neighbor banks in distance order (first *for* loop). This search is performed until the threshold $MaxHops$ is reached, which can be equal to the physical threshold forced by the system size (number of hops from the requestor to the furthest tile) or lower.

If all the L2 banks are full (*alloc* higher than *num_ways*), the algorithm tries to balance the number of allocations (thus, replacements) in all banks (second *for* loop). A threshold (*UtilThr*) is used. If the difference between the number of allocations in the local tile's bank and a neighbor bank is higher than the threshold, then the neighbor bank is selected as the *home* bank. If all the banks are balanced, then the block is mapped to the requestor's tile.

Notice that this does not imply that RHM defaults to private L2 caches. With private caches all the data accessed by a core must be present in the L2 bank of the same tile, while in RHM this does not apply. For instance, a shared block will be replicated in all L2 caches if they are private, while in RHM it will be present only in the *home* tile.

The proposed policy defaults to private caches only if all L2 banks are full, each core is requesting private blocks and all banks are uniformly used. In this case, very unlikely in a parallel application, all blocks are allocated in the requestor tile, which indeed is the best choice since it minimizes the data access latency.

4.2.4 Replacements in L1 Cache

L1 caches have to communicate with the home L2 when replacing a block, to keep data coherent (write-back of a modified block) and/or to keep the directory updated. The coherence messages involved in this process are commonly referred to as PUTS, which is a short message only used to update the directory, and PUTX, which is a long message used to both update the directory and the data block. Even though in some cases the replacement of a block can be completed without sending any message to the home L2 bank,[2] once the L1 cache is completely full, every cache miss requires communication with the home L2 to manage both the cache miss and the replacement of the LRU block.

If RHM is used, sending a PUTS/X to the home bank may require a search phase like block requests, thus doubling the NoC traffic in case both the requested and the replaced blocks are not stored in the local L2 bank. The traffic increment due to the additional search phase can be avoided in two ways: the first one is to include both the requested and the replaced block addresses in the same message, broadcasting the message to all nodes in case one block (or both ones) are not found in the local bank. This limits the traffic in the regular NoC but not in the Gather Network, since acknowledgements must be sent separately for the two blocks. The second way to optimize replacements is to extend the L1 tag entries and save the ID of the home L2 bank when the block is received, to be able to send a unicast PUTS/X in case of replacement. In case the requested block is provided directly by the L2 bank, its ID is stored in the header of the data message (*Sender* field), while in case the requested block is provided by the owner L1, it must copy to the data message the ID stored in its tags (both indicating the home L2 bank as the message sender or using a dedicated field in the header).

[2] A shared block can be replaced without updating the directory; in this case, the L1 which replaced the block receives an unnecessary invalidation message when the home L2 replaces the block itself or receives a write request

4.3 Optimizations to RHM

This section describes additional optimizations to the basic RHM mechanism, two at protocol-level (block migration and block replication), and two at NoC-level (Parallel Tag Access and the Broadcast Network). While the protocol-level optimizations are orthogonal one to each other and to the NoC-level optimizations, each NoC-level optimization can be only implemented if the other one is not. Parallel Tag Access (PTA) indeed is used to crop broadcast branches, while the Broadcast Network is used to optimize RHM when it's used with Hammer coherence protocol, exploiting the broadcast nature both of RHM and Hammer; in this case broadcast messages must be received by all nodes, so PTA cannot be used.

4.3.1 Block Migration

RHM reduces the access latency by mapping blocks closer to requestors. Once the block is on-chip, however, the core which actually uses it may change at runtime. To reduce the access latency in these cases, the initial placement of the block can be adjusted at runtime allowing blocks to migrate to a new L2 bank. Notice that a sort of migration mechanism is implicit in RHM, since each time a block is replaced from an L2 bank and then requested again it may be mapped to another L2 bank by the MC. However, this may not always be effective: if the block is never replaced by the L2, it stays in the bank where it was mapped to by the MC. Blocks replacement may be forced if many requests are received from an external tile, but that would be highly inefficient since a hit in the LLC would be transformed into a miss just to move the block to another bank, and it would require an expensive off-chip access.

If the initial *home* allocation performed by the MC results sub-optimal, block migration can be enabled to further reduce the number of hops between an L1 cache and the L2 bank where the block is mapped to, through a mechanism similar to the one used in D-NUCA but without the constraint of being limited within a bank set: in RHM a block is allowed to migrate to any L2 bank. However, since the migration process introduces an overhead in terms of traffic and energy, it should be performed only if it actually leads to a benefit in terms of miss latency reduction. Furthermore, if two cores located at two opposite directions keep requesting the same block, this may cause the continuous

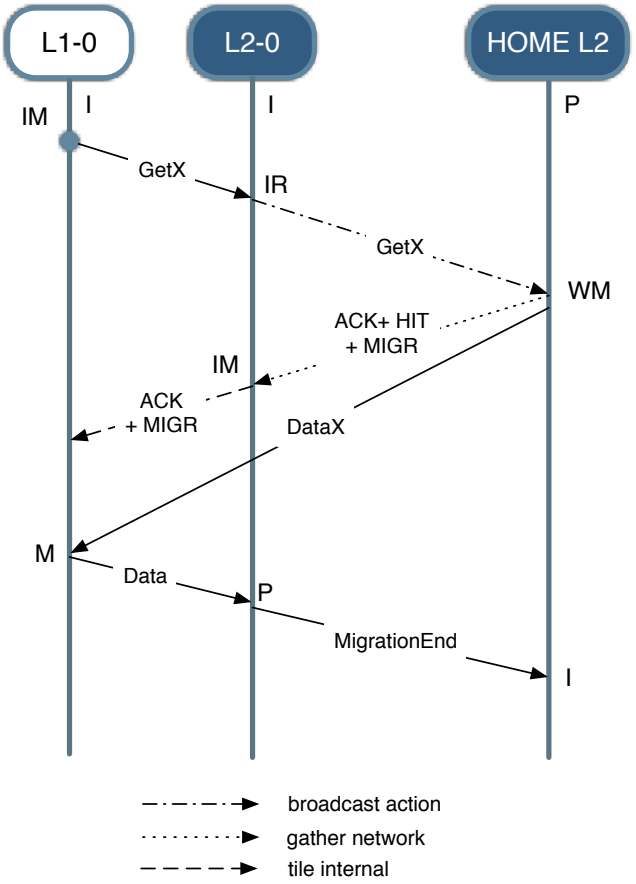


FIGURE 4.16: Block migration process.

migration of the block from a bank to another, leading to a performance degradation due to the overhead of the migration process; this problem is known as *ping-pong*.

Solutions in D-NUCA reduce unnecessary migrations and avoid the ping-pong effect by using a saturating counter for each direction to which a block can move. A counter is updated each time a request comes from a node located in the counter's direction; when the counter saturates, the migration process towards that direction is triggered. In RHM a block may migrate in any direction, so four counters are needed, one per direction. Each time a request is received, the counters are updated adding the distance in hops from the requestor. When a counter is incremented, the one in the opposite direction is decremented. When a counter saturates, it triggers the migration process: the block migrates to the L2 bank located in the same tile of the L1 which sent the request that triggered the migration.

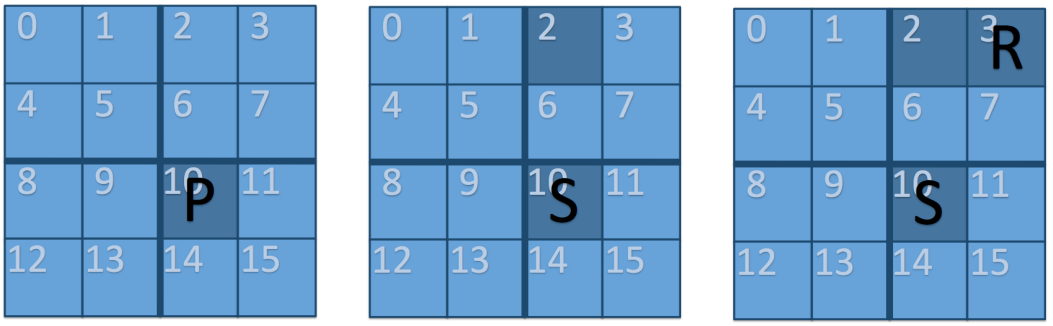
Figure 4.16 shows the block migration process in detail. A write request issued by L1-0 misses in the local tile, and the broadcast request hits in the home L2; the home

acknowledges the request through the GN, setting the *HIT* and the *MIGR* flags, sends the requested block to L1-0 and switches to state *WM*, in which it waits the migration process to finish; L2-0, receiving the global ACK with the *MIGR* flag set, prepares to receive the migrating block by allocating a cache line and sets the *MIGR* flag in the internal ACK sent to L1-0. The *MIGR* flag notifies the L1 that it has to forward the block to the local L2 once *DataX* is received. When the local L2 receives the data, the state of the block switches to *P* and a *MigrationEnd* message is sent to the former home L2, notifying the end of the migration process; at the reception of *MigrationEnd*, the former home L2 deallocates the cache line.

The MigrationEnd message is needed to avoid race conditions in case other requests for the same block are received by L2-0 or Home L2 while the migration process is ongoing. During the migration process, indeed, the reception of other requests for the migrating block may lead to the *false miss* problem: the request misses in both the former home L2 and the new home L2, resulting in an LLC miss which triggers a request to the memory controller. Previous proposals in D-NUCA caches [26] employ a False Miss Avoidance protocol, in which the old home L2 forwards the received requests to the new one, which is in charge to provide the block to L1 requestors; this could lead however to manage the same request twice, first when it is received directly at the new home and then when the forwarded request is received, so the protocol must be further complicated to manage these duplicated requests. To avoid the additional messages and MSHR structures needed to manage duplicated requests, we chose to delay the requests received during the migration process: referring to Figure 4.16, the *RETRY* flag is set in the ACK sent by L2-0 while the block is in state *IR* and in the ACK sent by Home L2 while the block is in state *WM*, so any request received during the migration process will be repeated by the sender. More details about the migration process can be seen in the protocol specification in Appendix A.

4.3.2 Block Replication

Block migration effectively reduces the LLC access latency of private blocks. Each time the accessing core changes, the block potentially migrates to the L2 bank in its tile. For shared blocks, however, this may not be the optimal solution. First of all, the block may be mapped close to one of the sharers but away from the others. Second, the block may



(a) one private copy of the block is stored at L2-10

(b) the block stored at L2-10 is shared by L1-10 and L1-2

(c) the block stored at L2-10 is shared by L1-10, L1-2 and L1-3; it is replicated at L2-3

FIGURE 4.17: Block replication.

ping-pong between two L2 banks when different sharers continuously request it. Since a block cannot be modified while it is shared, it is safe to replicate it in more L2 banks to reduce the access latency of the sharers. To limit the number of replicas, we partition the chip in a reduced number of replication regions as shown in Figure4.17. Four regions are defined, each including four tiles, and at most one copy of the block can be present in the L2 banks of each region.

Figure 4.18 shows the replication process in detail. A read request issued by L1-0 misses in the local L2 and it is broadcasted to all L2 banks; when the broadcast is received at the Home L2, the data is sent to L1-0 and the *HIT* and *REPL* flags of the ACK sent to L2-0 are set, so the internal ACK sent by L2-0 to L1-0 will have the *REPL* flag set to notify L1-0 that it has to provide the block to the local L2 when it receives the data message. Once L2-0 receives the block, a *ReplicationACK* is sent to the Home L2. The reception of *ReplicationACK* completes the migration process at Home L2: the ID of L2-0 is stored and the state of the block switches to *Sr*, indicating a shared block which is replicated. A *ReplicationEnd* message is sent to L2-0, and its reception completes the replication process: the state of the block in L2-0 switches to *Replica*, and starting from this moment it will provide the block when it receives read requests issued by L1 caches located in its region.

To avoid race conditions, requests received for a block during the replication process are delayed as described for the migration process: the *RETRY* flag is set in the ACK sent by L2-0 while the state of the block is *IReplica* and *ReplicaWE*, and in the ACK sent by Home L2 while the state of the block is *WR*.

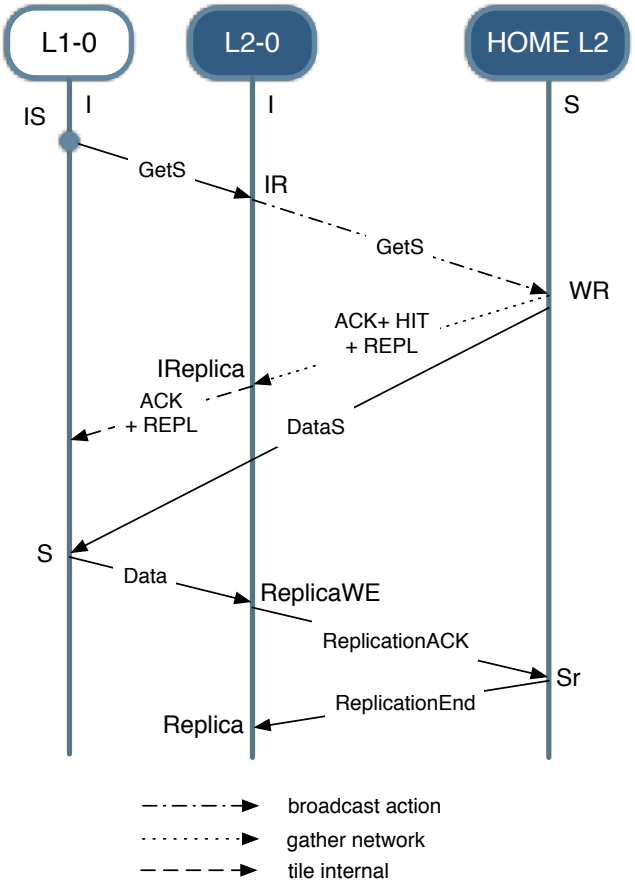


FIGURE 4.18: Block replication process.

The minimal access latency is achieved when each sharer has a replica of the block in its local L2 bank. This case, however, may end up suboptimal as the on-chip cache capacity will be highly reduced due to the high number of replicated blocks. To avoid unnecessary replicas, a saturating counter is associated to each region. Since a replicated block cannot migrate, the bits used to store the migration counters are also used to manage the replication process. As shown in Figure 4.8, we assumed 4 bits per direction and dimension for the migration counters; these become 4 bits per region when the block is shared and can thus be replicated. The first 2 bits are used for the counter: when 4 requests are received consecutively from the same region, the block is replicated.

The remaining 2 bits are used to indicate, when the block is replicated, the ID of the L2 bank which stores the replica within that region, which is used in two cases: first, to correctly manage the invalidation process in case a write request is received for the replicated block. L2 banks containing a replica indeed are only allowed to provide the block when it receives a read request issued by an L1 cache located in its region; in all

other cases the access misses in that L2 bank. Write request thus, even if they are issued in the same region (or in the same tile) of the replica, only hit in the home L2, which is in charge to invalidate the sharers and all the possible replicas. If the block is not replicated in a region, then invalidation messages are sent directly to the sharers; if a replica exists, then a single invalidation message is sent to the replica, which invalidates its copy and sends another invalidation message to each sharer in its region. This way, when the requestor receives all the acknowledgements sent by the sharers, only the home L2 bank holds a valid copy of the block. The ID is also used to allow a replica to become the new home bank if the current home bank must replace its copy. This way a shared block which is replicated in different L2 banks is kept on-chip until all the replicas are replaced.

Figure 4.17 shows how the replication process works. The CMP is divided in four regions, each one including four tiles, marked with a thicker line. Initially, there is only a private copy of the block in the L2 of tile 10 (L2-10), which is the tile the MC mapped that block to (or the destination tile of a previous migration process). In Figure 4.17.(b) L1-2 requests the block with read permission, so the block is shared by L1-10 and L1-2. If L1-2 or any other L1 of the same replication region requests the block several times until the counter for its region saturates, the replication process begins: L2-10 sends the block both to L1-3, which requested it, and to L2-3, where it is saved as a replica (Figure 4.17.(c)). Notice that L2-10 remains the home bank for the block: it keeps updating the directory and managing all requests, except for the read requests issued by L1s located in the same region of L2-3. For those requests, L2-3 is in charge of providing the data to the requestors.

When the home bank has to replace a replicated block, it is not necessary to invalidate the sharers. One of the L2 banks with a replica of the block is chosen to become the new home for the block, so is notified with a message which includes the directory information for that block. On the other hand, if an L2 bank has to replace a replica, it only has to notify the home bank. The home bank will remove its ID from the list of replicas and manage the requests originated in that region.

4.3.3 RHM and Broadcast-based Coherence Protocols

In previous sections we assumed a directory-based coherence protocol, with a data structure (the directory) associated to each L2 cache line to store the list of sharers used when the L2 home bank has to communicate with L1 caches to manage a request; two typical cases are the invalidation of the sharers of a shared block upon a write request and the forwarding of read and write requests to the L1 which owns the modified copy of a private block. The directory introduces an area overhead, since part of the on chip memory has to be used to store the directory entries.

Broadcast-based protocols completely eliminate the list of sharers from the directory information, thus eliminating its extra area and power requirements. The drawback of broadcast-based protocols is the amount of traffic they generate. While in directory-based protocols the L2 cache always knows exactly to which node it has to communicate, thus injecting in the NoC the minimum amount of traffic, in broadcast-based protocols a broadcast must be sent to all L1 caches each time the L2 home bank has to invalidate the sharers of a block or forward a request to the L1 which has a private copy of the block. In addition, each node must answer the broadcast message by sending an acknowledgement message (ACK) to the requestor, and, if the node is also the owner of a private block, the requested data block. Due to these additional communication, broadcast-based protocols have usually worse performance than directory-based.

However, the GN used in the search phase of RHM can be extended and also used to collect the acknowledgments of L1 caches. Notice that RHM uses the GN to transmit ACKs exchanged between L2 banks. Thus, now, in order to extend GN notifications between L1s, the ID of the GN must be extended to differentiate between L2 acknowledgements, generated during the RHM home search phase, and L1 acknowledgements, generated according to Hammer protocol when a request is managed. This allows to save the area and power overhead of the directory maintaining performance comparable to that of the directory protocol.

Furthermore, with some modifications of the GN module, it is possible to reuse the information of the RHM home search phase to save a second broadcast when the home is found and the request requires a broadcast to all L1 caches to be managed. To achieve this goal, the GN is extended to implement a Broadcast Network (BN) to send a fast notification from any node to all other nodes.

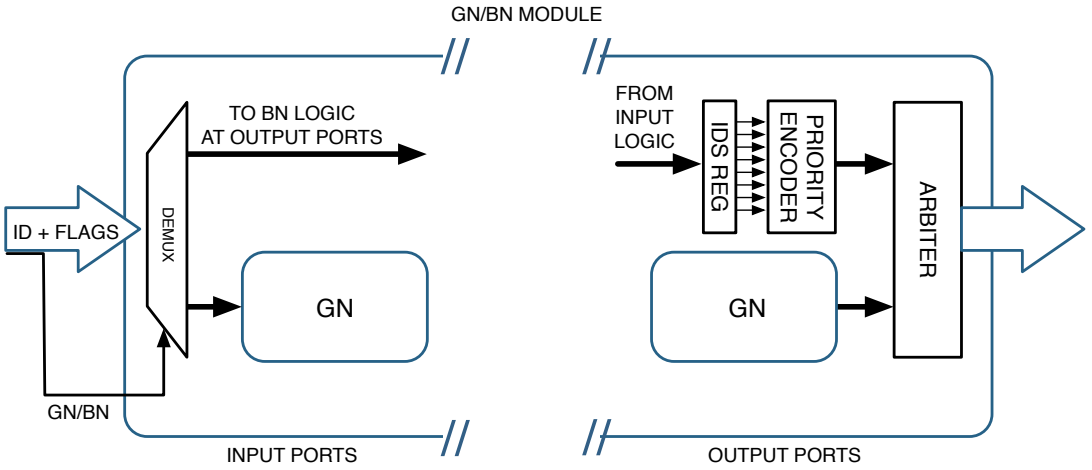


FIGURE 4.19: BN implementation.

4.3.3.1 Broadcast Network

As described in Chapter 3, the GN can be logically described as a set of AND trees, one per tile. Each tree has its root in the destination tile and all other tiles are at the leaves. The sequential implementation we assume in this chapter combines IDs of the destination node, following the structure of the AND tree. By crossing the tree in the opposite direction, it is possible to broadcast the ID of a node to all other nodes.

Figure 4.19 shows the implementation details of the BN. It reuses the links of the GN to transmit to all nodes the ID of the node which initiated the broadcast. An additional bit is used to differentiate these messages from those of the GN. This bit (GN/BN) drives a demultiplexer thus the incoming bit at the input port is forwarded either to the GN input logic or to the BN logic. The BN logic simply disseminates the ID through some output ports, following the XY pattern. At each output port, a register with as many bits as IDs is implemented. When an ID is received at an output port, the associated bit is set. The register is then inputed to a priority encoder and the output link is arbitrated between the GN output logic and the BN output logic.

4.3.4 Merging Hammer Protocol and RHM

The basic coherence actions when Hammer protocol is combined with RHM (RHM-Hammer) are shown in Figures 4.20, 4.21 and 4.22. Each time the L2 home bank needs to communicate with an L1 cache to satisfy a request, it broadcasts the request to all

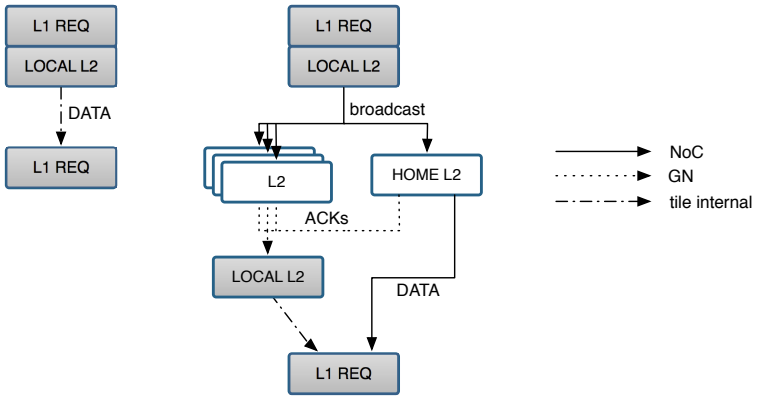


FIGURE 4.20: Read request for a shared block in case of hit (left) or miss (right) in the local L2 bank.

L1 caches, which answer by sending an ACK back to the L1 requestor. The GN we considered so far must then be expanded with an additional GN: one GN (GN-1) will be used to collect the ACKs sent by the L2 banks during the home bank search phase, and a second GN (GN-2) will be used to collect the ACKs sent by the L1 caches back to the L1 requestor once the home bank has been found and it is managing the request. By adding one bit to the ID identifier we easily provide support to the GN-2. Notice that GN-1, GN-2 and BN logic blocks at each router share the same physical links. In Section 4.4 we will provide evaluation results of the number of conflicts inside a GN module.

In case the requested block is not cached on chip, RHM-Hammer behaves like the Directory case (RHM-Directory): the home search phase will miss in all L2 banks, and the L2 of the requestor's tile will send a request to the MC. It fetches the block from main memory and executes the home mapping algorithm. Once the block is received from main memory, a data message is sent to the chosen L2 home, which will provide the block to the requestor.

If the requested block is cached on chip, RHM-Hammer exploits the BN and the GN to speedup the coherence actions. In the case of a read request for a shared block (Figure 4.20) RHM-Hammer behaves like RHM-Directory: if the request hits in the local tile, the local L2 directly sends the data to the requestor. If, however, the request misses in the local tile, a broadcast is sent to all L2 banks and the block is provided by the home bank.

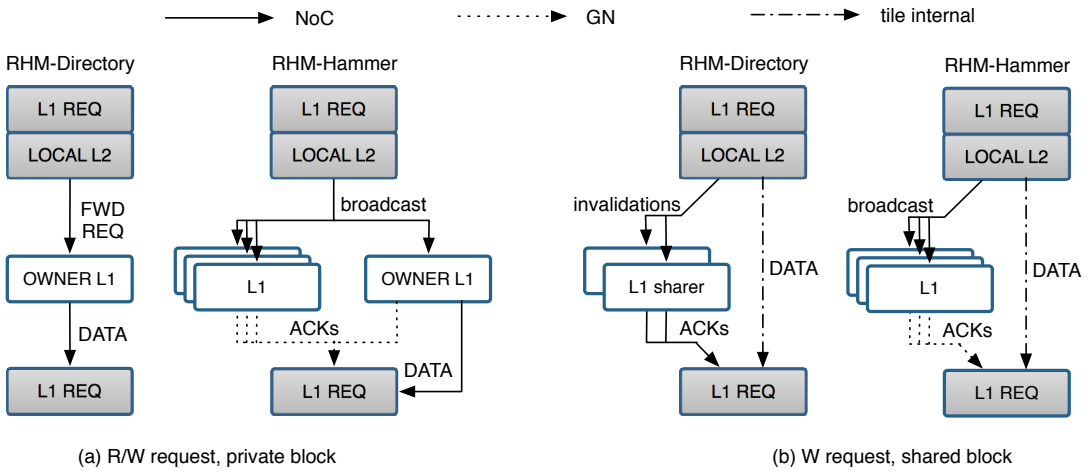


FIGURE 4.21: Request hit in the local L2 bank.

In all other cases RHM-Hammer uses the GN and the BN to manage the request. Figure 4.21 shows the case when a request hits in the local L2 bank. In case of read or write request for a private block (Figure 4.21.a), the local L2 bank sends a broadcast through the NoC to all other L1 caches, which answer acknowledging the broadcast through the GN. The L1 which owns the private copy is in charge of sending the block to the requestor. In case of a write request on a shared block (Figure 4.21.b), the broadcast is used to invalidate the sharers and the data is provided by the local L2 bank. Notice that RHM-Directory can not use the GN to collect the acknowledgements since the invalidation message is only sent to the sharers, so some nodes at leaves of the requestor's AND tree would not send an ACK. The acknowledgement phase is thus faster in RHM-Hammer.

Figure 4.22 shows how the previous cases are managed when the request misses in the local bank. In case of read or write request for a private block (Figure 4.22.a), the local bank starts the home search phase. During this phase, all tiles, upon receiving the request, save in a private table, at the entry of the requestor, the block address and the request type. This table will have as many entries as IDs are supported by the BN.

When the request is received by the home bank, a broadcast is sent through the BN to all the L1 caches: the ID of the requestor is broadcasted through the BN. Once an L1 cache receives the requestor ID, it checks the table to know which address and access type are associated to that ID and performs the actions established by the coherence protocol. If the ID is received at the owner L1, it sends the block to the requestor and invalidates its cache line (in case of write request) or changes the line state from private

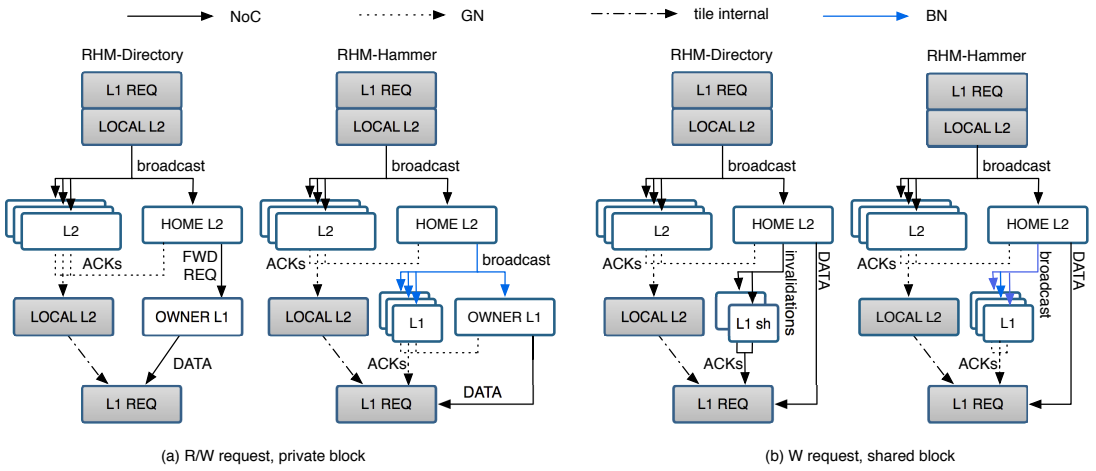


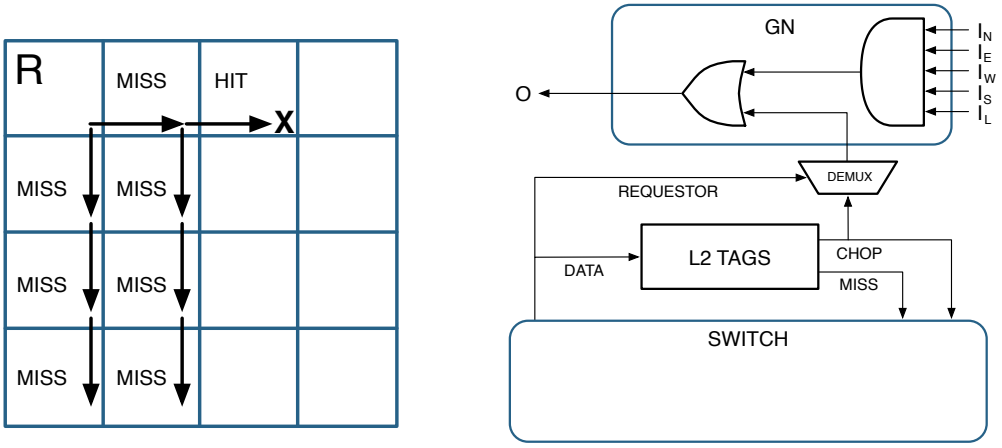
FIGURE 4.22: Request miss in the local L2 bank.

to shared (in case of read request). In case of a write request on a shared block (Figure 4.22.b), the home bank must invalidate the sharers. Again, this is done by broadcasting the request through the BN and all ACKs are collected through the GN. Notice that in RHM-Directory both the invalidation messages and the acknowledgements are sent through the regular NoC.

Although RHM-Hammer seems to generate much more traffic than RHM-Directory, all the additional traffic is actually sent through the GN and the BN, which is faster and has much lower energy requirements than the regular NoC. Furthermore, since in case of miss in the local L2 bank the communication between the home bank and the L1s is done through the BN, it is faster than in the case of RHM-Directory, where the regular NoC is used to forward the request to the owner or to invalidate the sharers of a block.

4.3.5 Parallel Tag Access

Built-in NoC broadcast support and the GN highly reduce the NoC traffic generated during the *home* search phase. However, traffic can still be reduced by eliminating useless broadcast branches. In Figure 4.23.a a request misses in the local L2 bank and is broadcasted to all other banks. Since the data is found in L2-2, there is no need to propagate the broadcast through east and south directions. Parallel Tag Access (PTA) is a mechanism to allow the parallel access to the L2 tags while a broadcast message is crossing the NoC router pipeline. This way, in case of an L2 hit, the broadcast branch can be cropped before the message reaches the crossbar stage of the switch. Figure 4.24.a



(a) Example of broadcast branch cropping.

(b) Modification of the GN to support broadcast branch cropping

FIGURE 4.23: Parallel Tag Access: motivation and implementation.

shows a basic 4-stage router modified to enable PTA. As the flit exits the input buffer to enter the routing stage, it is also sent to the L2 tag array to perform the lookup. In case of a hit, the $CHOP$ signal dismisses the flit (the flit is converted to a bubble). In case of a miss, $MISS$ signal allows the flit to cross and replicate. Those signals are also used to generate a global ACK signal in the GN module of the tile where the broadcast is cropped, as shown in Figure 4.23.b.

Theoretically, the L2 cache must be able to trigger one of the signals within 2 cycles, while the flit is crossing the routing and the VA/SA stages. To limit power consumption and allow a fast cache lookup, a sequential access to the L2 bank must be implemented: the tag array is accessed first and then, in case of a hit, the data array is accessed. If the tag access latency is still higher than two cycles, the L2 bank can be partitioned into sub banks until the size of the tag array allows a 2-cycle lookup. Alternatively, the L2 bank may require more cycles than the flit to cross the router (case of a shorter pipeline design as shown in Figure 4.24.b). In this case, the flit gets blocked at the VA/SA stage until the L2 tag access is performed, then the flit either advances through the crossbar (in case of a miss) or it is dismissed (in case of a hit). Notice that only broadcast request messages, which are single-flit messages, have to wait for the L2 bank to access the tag array. Also, with the obtained high locality of data in the local L2 bank (seen in the evaluation), the effect of this delay is negligible.

The basic 4-stage switch using the 45nm technology Nangate [61] library with Synopsys DC, and the modifications needed to couple the switch and the L2 tags in order to allow

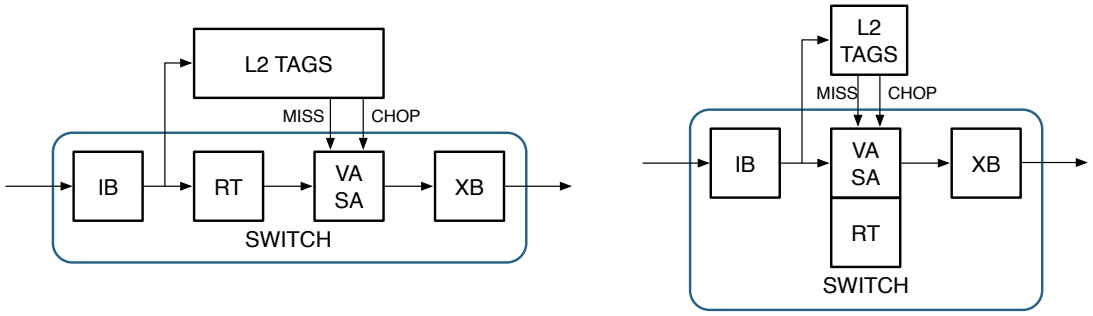


FIGURE 4.24: 4-stages (left) and 3-stages (right) switches modified to allow parallel tag access.

PTA resulted in a negligible area overhead.

Parallel Tag Access can not be used with RHM-Hammer, since home search broadcasts must reach all the tiles to reuse the information when requests are broadcasted through the Broadcast Network. Network. However, it can be used with RHM-Directory.

4.4 Evaluation

In this section RHM is evaluated and compared to other proposed NUCA configurations. In particular, these configurations have been implemented in gMemNoCsim:

- S-NUCA, a baseline configuration where blocks are statically mapped to L2 banks using the less significant bits of the block address
- D-NUCA, where blocks are statically mapped to a bank-set depending on their address; the matrix of L2 banks is divided in bank-sets, one per column of tiles; blocks are inserted in the L2 bank located in the same row of the requestor and then can migrate within the bank-set, one hop each time a migration is triggered
- PRIVATE L2, where L2 banks are private to the core in the tile and simply extend the private L1 cache; a directory is implemented at the memory controller
- FIRST TOUCH, an S-NUCA configuration in which the blocks are mapped to the L2 banks using a First-Touch policy; the first time a block is requested, the memory page containing that block is mapped to the L2 bank in the requestor's tile; we'll assume 4KB as the page size

Routing	XY	Coherence protocol	Directory / Hammer
Flow control	credits	L1 cache size	16 + 16 kB (I + D)
Flit size	8 byte	L1 tag latency	1 cycle
Switch model	4-stage pipelined	L1 data latency	2 cycles
Switching	virtual cut-through	L2 bank size	256 kB
Buffer size:	9 flit deep	L2 tag latency	1 cycle
Virtual channels:	4	L2 data latency	4 cycles
GCN/BCN delay	1 cycle/hop	Cache block size	64 B

TABLE 4.1: Network and cache parameters (RHM evaluation).

These configurations assume a directory-based coherence protocol, and are compared to four configurations of RHM:

- RHM, a basic implementation of RHM with directory protocol
- RHM M, an implementation of RHM with directory protocol where private and shared blocks can migrate from an L2 bank to another
- RHM M+R, an implementation of RHM with directory protocol where private blocks can migrate while shared blocks can be replicated in different L2 banks
- RHM HAMMER, an implementation of the basic RHM with Hammer protocol

The cache coherence protocol for each configuration, the NoC with broadcast support and the GN/BN networks have been implemented and simulated using gMemNoCsim. Each protocol has been tested for deadlocks and race conditions with all the applications used in the simulation phase. Graphite and Sniper's were used to capture the memory access traces of simulated cores executing different applications of the SPLASH-2 and PARSEC benchmark suites and use that in gMemNoCsim for cache hierarchy and NoC timing. Network and cache parameters are shown in Table 4.1. Cache latencies have been obtained using Cacti [58]. The memory controller is placed at the top left corner of the chip. For the sake of fairness, ACKs in D-NUCA are modeled with 2-cycle latencies (as if a combinational GN was used).

4.4.1 Performance

Figure 4.25 shows the average hop distance from the requestor to the *home* tile. For S-NUCA, the block is found on average at a distance of 2.85 hops. This distance is roughly the same for most applications as blocks are uniformly distributed among the

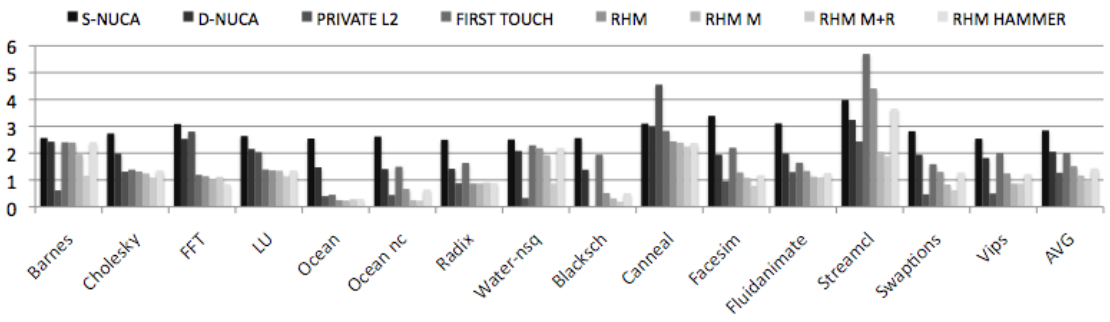


FIGURE 4.25: Avg hop distance between L1 requestors and the tile where the data is found.

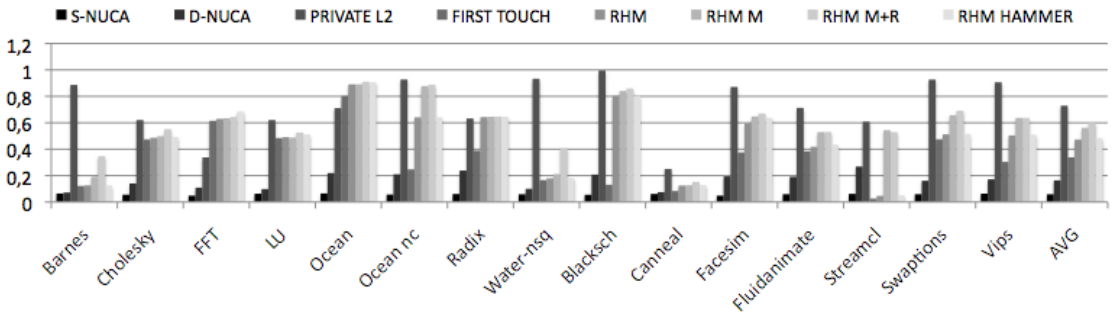


FIGURE 4.26: Percentage of hits in the L2 bank located in the tile's requestor.

L2 banks. With other configurations, however, since blocks are dynamically mapped and/or moved from a bank to another, the distance is quite variable depending on the application. For Barnes, dynamic techniques are not so effective, and the average value is always higher than 2 hops. The exception is for RHM M+R. This is due to the high sharing of blocks between cores. Thus, RHM M+R adapts to this type of sharing. For other applications, e.g. Ocean, those techniques achieve a large reduction in the average number of hops.

On average, RHM locates the data closer to the requestor than the other configurations, and this distance is further reduced if block migration is enabled. Indeed RHM M and RHM M+R achieve a locality close to that of PRIVATE L2.

Figure 4.26 shows the percentage of requests which hit in the L2 bank located in the same tile of the requestor. Again, results when using S-NUCA do not depend on the application due to the uniform mapping of the blocks, and this percentage is quite low (6% on average). This percentage increased to 16% for D-NUCA, but is still much lower when compared to First Touch (33%), RHM (49%), RHM HAMMER (49%), RHM M (56%), RHM M+R (60%) and Private L2 (72%). Thus, the most effective dynamic method is RHM.

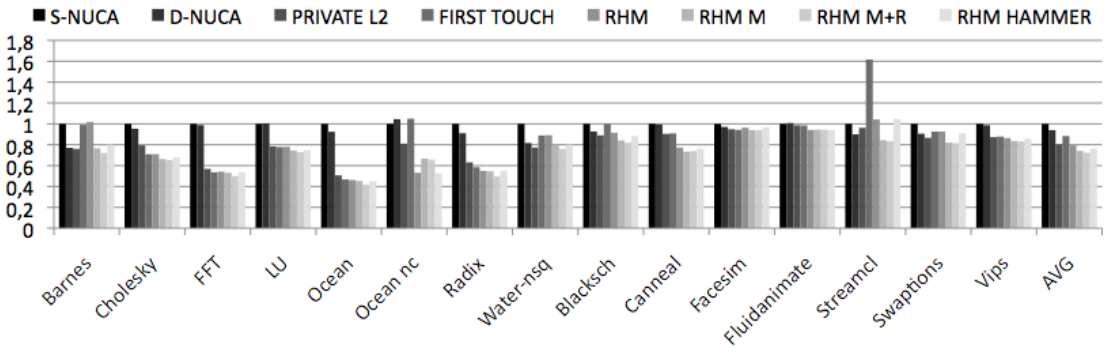


FIGURE 4.27: Execution time normalized to the S-NUCA case.

Figures 4.27 shows the normalized execution time with all the configurations. We can observe how execution time is largely reduced with an average factor of 12% when using FIRST TOUCH and ranging between 20% and 28% when using the various configurations of RHM. RHM achieves lower execution time due to its achieved higher locality in L2. Also, the migration and replication policy helps in further reducing execution time. Contrary to this, D-NUCA is not able to achieve large reductions when compared to S-NUCA. The use of private caches achieves large execution time reductions but its effectiveness depends on the size of the working set of every application. We can also see the on-par execution time benefits of the RHM-HAMMER protocol.

Figure 4.28 shows the average load and store latency, respectively, for the evaluated configurations, normalized to the S-NUCA approach. RHM configurations reduce these latencies by more than 25% on average and up to 75% (FFT store latency). Again, the effectiveness of RHM in reducing the miss latency depends on the memory access pattern of each application. Streamcluster shows a high percentage of blocks which are first accessed by a tile and then by different tiles during different phases of the application. In this case, a first touch policy has the negative effect of overloading the tile where blocks are mapped, and the migration/replication mechanism can effectively move the blocks to the correct tiles. RHM-HAMMER protocol also exhibits low load and store latencies, as it benefits from the fast GCN and BCN networks.

4.4.2 Performance Conclusions

When comparing results of different methods we can deduce some interesting observations. First, The S-NUCA approach has the severe limitation of its static mapping of L2 banks. This leads to the largest distances between L1 requestors and L2 home banks

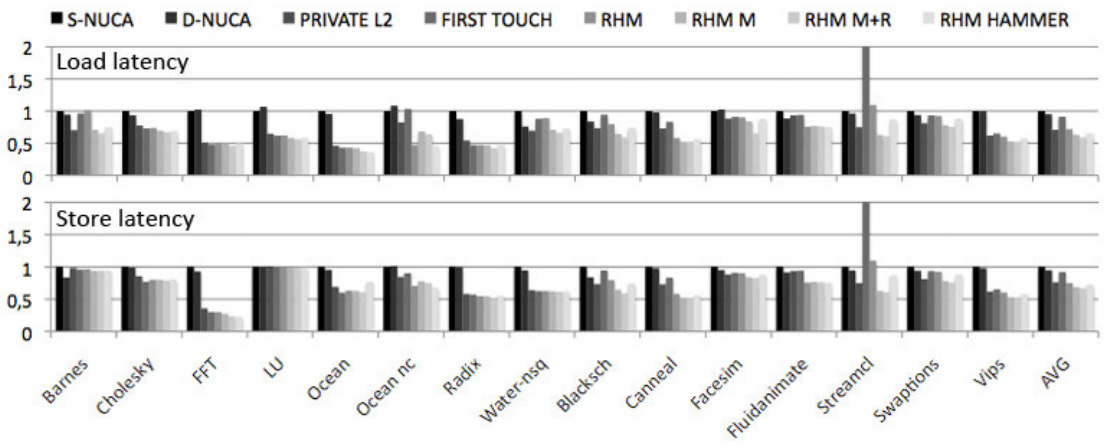


FIGURE 4.28: Average load and store latency, normalized to the S-NUCA case.

(near 3 hops on average) and the lowest rate in hits in local L2 banks (6% of hits). Execution time of applications is the worst when compared to the other policies. The same occurs for the average load latency and the average store latency.

The D-NUCA approach is a first step towards providing dynamism to the placement policy of L2 homes. However, its static partitioning in bank sets still forces poor results in terms of hop distance (2 hops on average) and hit rate in local L2 bank (less than 20%). Execution time is improved, when compared to S-NUCA because of the lower load and store latencies.

Private caches (PRIVATE L2) obviously achieve low hop distances and the largest hit rate in local L2 banks. However, RHM M+R is able to improve further the hop distance but not the hit rate. Because of its cache privacy policy, shared blocks impose an overhead which translates to larger execution time and latencies when compared to RHM. Although PRIVATE L2 reduces execution time of S-NUCA by 20%, an extra of 15% is obtained with RHM with migration and replication support.

For FIRST TOUCH, locality is not correctly promoted (average of 2 hop distance and 35% hit rate in local L2 banks). Execution time and average latencies are similar to the ones achieved by D-NUCA. Although FirstTouch is a simple mechanism not requiring any hardware assistance, it should be noted RHM allows finer-grained assignments (blocks vs pages) and also more effective thread migration as blocks can be effectively migrated along with threads.

When analyzing RHM and RHM HAMMER we can see that they achieve very close results. None of them use migration or partitioning support, thus they only differ on

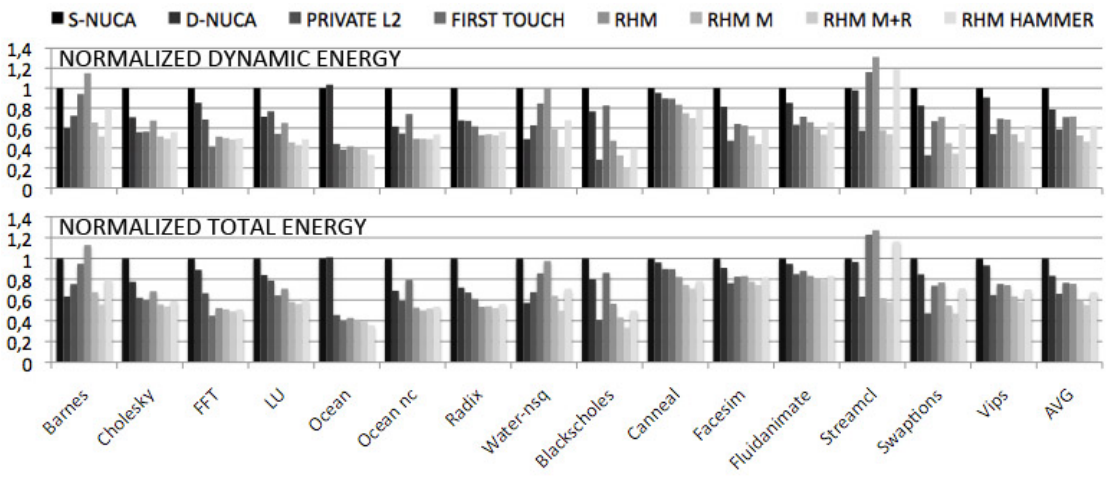


FIGURE 4.29: NoC's energy consumption

the way of locating sharers and owners of blocks. This impacts execution time. RHM HAMMER is able to run faster than RHM (3% faster). This is due to the use of the BN network. Average miss latencies do not get significantly affected. Notice also that the main benefit of RHM HAMMER is its reduced overhead in control structures.

Finally, we can see how RHM M+R is the best RHM option, getting close of the PRIVATE L2 results for hop distances and local hit rate. However, the extra flexibility of moving blocks between L2s makes the solution the most performant. Execution time of S-NUCA is reduced by 35% (execution time of PRIVATE L2 is reduced by 15%).

4.4.3 Energy

Figure 4.29 shows the normalized dynamic and total energy consumed by the NoC with the six configurations. Resource access (input buffer read/write, routing, switch allocation, crossbar traversal and link traversal) have been accounted and fed into Orion 2.0 [59]. If the request misses in the local L2 bank, RHM consumes more energy than the other schemes, due to the broadcasts. However, the high percentage of hits in the local L2 leads to less network activity compared to an S-NUCA. This, combined with the reduced execution time, leads to average energy reductions of 32%. Energy consumption is further reduced by 55% on average when migration is enabled (RHM MIGR).

Figure 4.30 shows the normalized energy consumed by the L2 cache. We used CACTI [58] to obtain the dynamic energy and the leakage per bank. Due to the broadcast access,

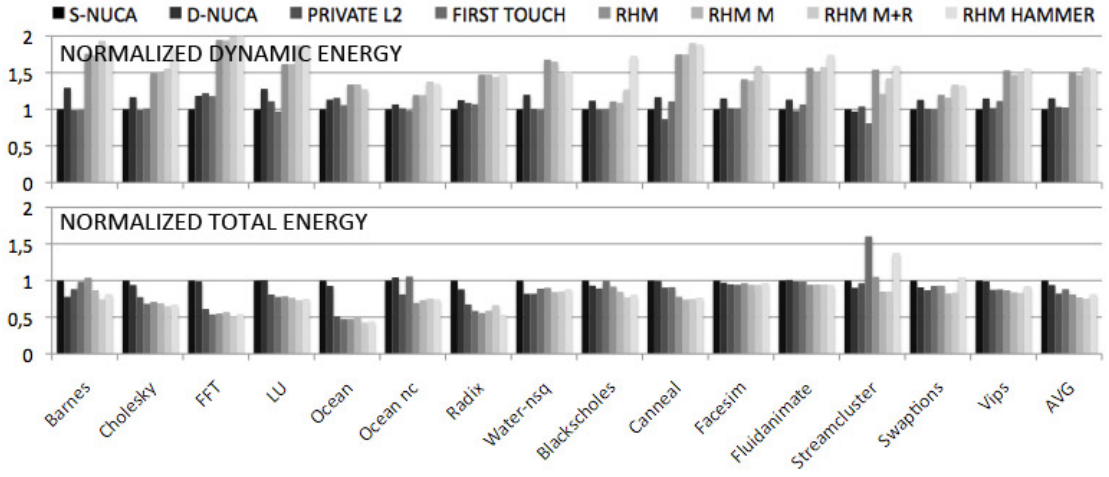


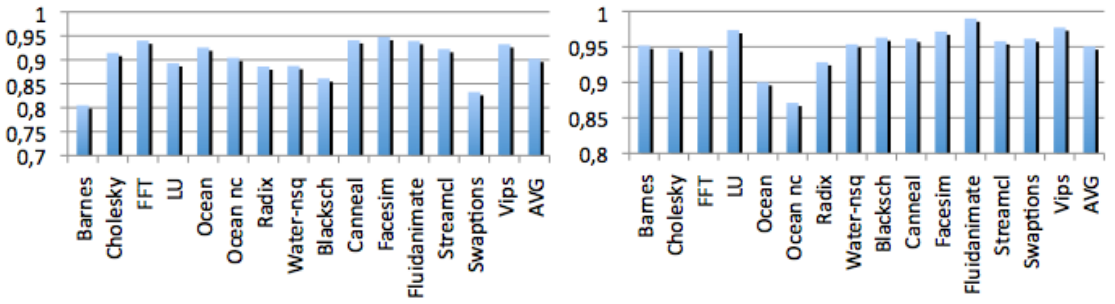
FIGURE 4.30: LLC's energy consumption

RHM consumes more dynamic energy than other proposals (50% more energy on average), but the leakage component, reduced by the lower execution time, dominates over the dynamic energy for the configuration we choose. On average, energy consumption with RHM is reduced by 29% without block migration and 31% when block migration is enabled.

The area overhead and the power consumption of the LLC utilization table at the memory controller are a minimal fraction of the overall chip area and power requirement, due to its very small size compared to the on-chip cache and to the limited number of accesses compared to L1 and L2 accesses (the table is only accessed in case of L2 miss).

4.4.4 Parallel Tag Access

Let's now enable the PTA and see how it impacts performance. Figure 4.31.a shows the normalized reduction in number of broadcast messages received with PTA. On average, PTA helps in reducing the number of received messages by 10%, saving link and router traversals and L2 tag accesses. The average number of messages saved per broadcast is 3.41 (without chopping, the number of messages per broadcast is 15). PTA improves the performance of RHM in two ways: first, as broadcast branches are cropped, the block search phase is faster; second, at the destination tile messages are delivered directly to the L2 bank without having to cross the 4 pipeline stages of the switch. These two effects combined lead to a further average reduction of execution time of 5% (12% for Ocean-nc).



(a) Broadcast messages (b) Execution time

FIGURE 4.31: Normalized reduction in broadcasts and execution time when using PTA.

4.5 Conclusions

In this chapter we presented Runtime Home Mapping, a dynamic policy to map cache blocks to LLC banks in a system which employs a NUCA LLC. Block mapping is performed at runtime by the memory controller, while it's waiting to receive the data from main memory. The mapping algorithm we implemented maps blocks to the same tile of the requestor as long as the local LLC bank is not full. In case it is full, neighboring banks are also checked, to balance the utilization of all banks.

Since in case of L1 miss the home bank for a block is not known, a home search phase is needed. First, the local bank is checked; if the request misses in the local bank, a broadcast is sent to all banks, which must acknowledge its reception. We used LBDR hardware broadcast to speedup the broadcast phase and the GN to speedup the acknowledgement phase.

Thanks to the totally dynamic mapping, blocks can migrate from a bank to another one or can be replicated in different banks, thus adjusting the initial placement performed by the MC and further reducing the LLC access latency.

If the system implements a broadcast-based protocol like Hammer, RHM and the coherence protocol can be tightly integrated, exploiting the broadcast nature of both RHM's search phase and Hammer's requests management at LLCs to optimize the coherence protocol.

One last optimization is the parallel access to the LLC tags and to the switch pipeline, to speedup the LLC access and crop unnecessary broadcast branches; this optimization though can only be used with Directory protocol.

Evaluation results show that RHM is more effective than the other configurations we considered (S-NUCA, D-NUCA, private LLC, First-Touch) in reducing the LLC access latency (and thus the execution time) exploiting at the same time all the on-chip cache capacity, particularly when it is integrated with Hammer protocol and when the migration/replication of blocks is enabled. Despite the expensive home search phase, NoC dynamic energy is reduced thanks to the high number of hits in the local cache. The dynamic energy consumed by LLC banks increases, since every broadcast involves an access to all banks, but since the leakage component dominates the total LLC energy is reduced compared to the other configurations, thanks to the reduced execution time.

A central role in RHM is played by the MC; we believe this component to be the fittest to perform the block mapping efficiently, since it is aware of the traffic between main memory and the LLC banks. OS-based policies like First-Touch are those which in this aspect are closer to that of RHM. We identified several advantages of RHM over OS-based techniques: first, OS-based techniques rely on static mapping at the hardware level, so they do not allow block migration or replication; being based on the paging routine, they can only act at page level, while RHM acts at the finer block-level granularity; last but not least, the paging routine is on the critical path of resolving an LLC miss, so the mapping algorithm must be kept as simple as possible since it latency directly influences the LLC miss latency; the mapping algorithm of RHM, on the other hand, overlaps with the main memory access, so the memory wall can be exploited to mask the execution time of the mapping algorithm.

Further research work on RHM can focus on several directions: a first, important issue is to adapt RHM to work in a system with more than one memory controller. Alternative, more sophisticated policies can be implemented at the MC; the policy we implemented is indeed general, fit for a general-purpose CMP system, but it can be modified to adapt RHM to a specific system running a specific kind of applications. At the NoC-level, alternative broadcast strategies and network topologies can be studied to optimize the search phase and improve the effectiveness of PTA; the mapping algorithm would be adapted to take advantage of the new broadcast strategy and NoC topology.

In the next chapter, RHM is combined with LBDR to allow the independent partitioning of cores, NoC resources and LLC banks in a virtualized CMP system.

Chapter 5

pNC: Partitioned NoC and Cache Hierarchy

In this chapter we integrate LBDR and RHM to create a hardware substrate for the independent partitioning of cores, NoC resources and LLC banks in a virtualized CMP. At the network level, we use LBDR extended with the hardware broadcast support, as in the previous chapters of this thesis; in this chapter, however, we also exploit the connectivity bits to partition the NoC into regions. At the LLC level, we use the basic RHM policy described in the previous chapter, without block migration/replication and without enabling the PTA mechanism. Directory coherence protocol is assumed.

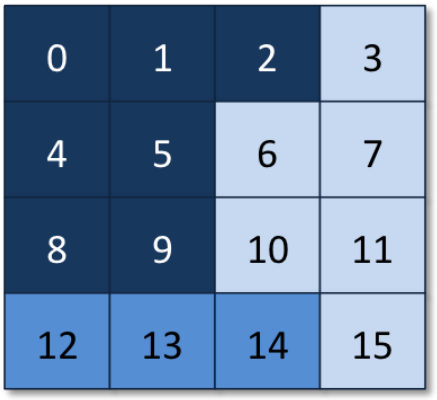


FIGURE 5.1: Partitioned CMP system.

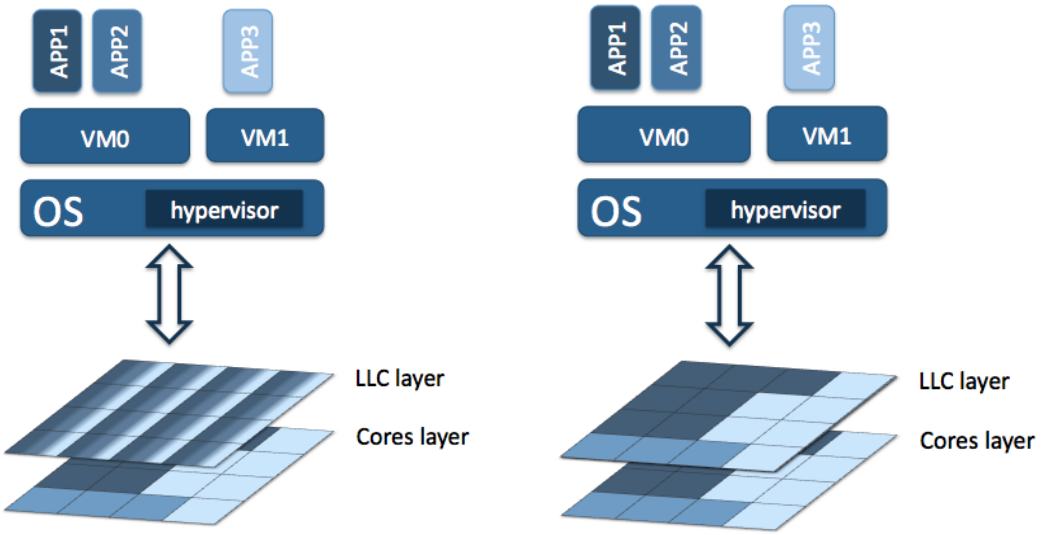
5.1 Introduction

Virtualization provides an effective way to maximize the throughput of a CMP system. An application sees its pool of virtualized resources as an independent dedicated system. In a virtualized system, resources are assigned to different applications providing to each application the view of exclusiveness of its resources. For instance, in Figure 5.4 a 16-tile CMP system is virtualized to execute three applications, each running in a disjoint set of cores. In such situation, it is mandatory to prevent each application from affecting the performance of the other ones. Indeed, a perfect and independent partitioning is needed not only of cores, but also of memory and NoC resources.

Focusing on cache resources, the use of alternative mapping policies of cache blocks to LLC banks such as RHM is particularly appealing in a virtualized/partitioned environment: a static home mapping approach would spread the blocks over all the LLC banks, thus destroying the exclusiveness in the use of cache and NoC resources. A dynamic home mapping approach, instead, can keep all the blocks in LLC banks located within the partition of the application. Figure 5.2.a shows the virtualized system when a static home mapping is used. As can be seen, the LLC layer is not partitioned and every core can access blocks allocated in any LLC bank. However, with a dynamic approach (Figure 5.2.b) both the core and the memory layers are partitioned and thus traffic originated from different applications do not mix.[1]

This chapter illustrates a co-design of the network and memory hierarchy to achieve an effective partitioning of chip resources, enabling an efficient implementation of a

[1]The only exception is when L2 misses require accessing the memory controller, which, can be located in a corner of the chip thus needing to cross other partitions.



a) LLC partitioning following a static LLC bank mapping.

b) LLC partitioning following a dynamic LLC bank mapping.

FIGURE 5.2: Virtualized CMP system to three applications. Resources are assigned to different applications.

virtualized system. In order to achieve such partitioning, the following contributions are proposed:

- A management policy where the memory controller is aware of the applications running in the chip and their partitions. The memory controller will be in charge of performing the home LLC bank mapping taking into account the resources assigned to each application.
- A routing framework, combined with the efficient LBDR approach [45]. The framework will allow an efficient partitioned-based search of the home banks when private caches accesses result in a miss.
- An infrastructure to collect control messages, embedded in the NoC, that will be bounded to each partition, improving the latency of private misses. The collecting system will be integrated into the routing framework.
- A method to allow applications with large memory footprints to get more LLC banks. In this case chip resources are not assigned at tile level: an application may use LLC banks which are located in tiles where another application with less memory requirements is running.

- A strategy that allows applications to share some LLC banks if they share code or data.

In addition, our proposal provides support to LLC banks failures. The memory controller will keep into account LLC banks which have been detected as failed. Those banks will be avoided when the mapping is performed.

The whole mechanism is termed pNC, meaning it supports partitioning both at NoC level and at Cache hierarchy level.

This chapter is organized as follows: Section 5.2 describes how to combine LBDR and RHM to allow the efficient partitioning of NoC and LLC layers. Section 5.3 provides evaluation results of a system which implements pNC. In Section 5.4 we draw the conclusions.

5.2 NoC and Cache Hierarchy Substrate

pNC is based on the combination of two previous methods. At the NoC level, LBDR is used to define the partitions. At the cache level, RHM is used to map blocks within a partition. LBDR and RHM are orthogonal techniques, and this chapter describes how they can be integrated to provide a NoC and cache substrate in which both cooperate with the OS's hypervisor to effectively partition chip resources when the CMP system is virtualized.

5.2.1 pNC: LBDR and RHM Support to Virtualization

The basic idea behind pNC is to allow RHM to work at the partition level, where partitions are defined by LBDR. With the connectivity bits, different partitions at the NoC level can be defined, some of them overlapped. This is achieved by using a vector of connectivity bits per output port. Packets injected by applications incorporate in their headers an ID field that identifies the partition. With 3 partition bits, up to 8 overlapped partitions can be defined. Note that two disjoint partitions can use the same ID but routing packets at different regions of the CMP, so the combinations of partitions is actually much larger.

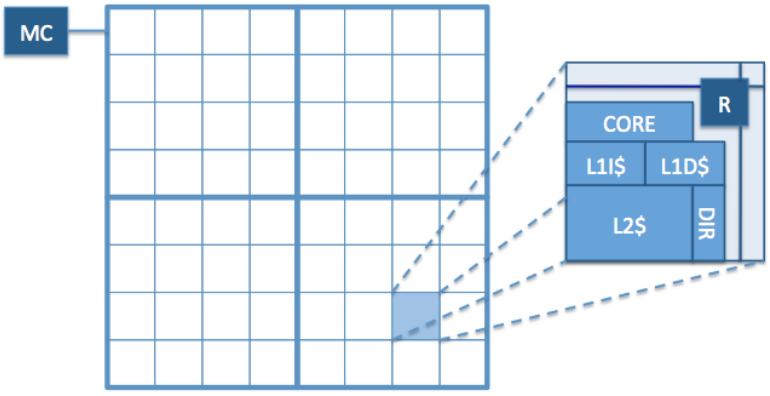


FIGURE 5.3: Baseline system for the pNC approach.

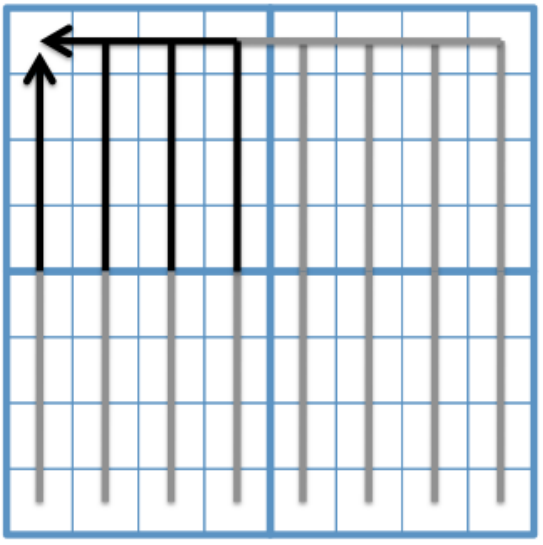


FIGURE 5.4: GN signals in a virtualized environments.

In this chapter we assume the 64-tile CMP shown in Figure 5.3, where tiles are organized in an 8×8 mesh. For the sake of simplification, we assume the default partitioned regions outlined with thicker lines. Thus, the chip is logically divided in four partitions, each including 4×4 tiles. Note that other partitions are possible and they can be reprogrammed on-the-fly in the system by just changing the connectivity bits at the switches.

LBDR connectivity bits prevent traffic from leaving a partition. This is especially useful for broadcast messages, which are constrained within a partition avoiding the unnecessary flooding of other partitions. However, the GN infrastructure, such as it is presented in Chapter 3, cannot work properly if broadcasts are limited within a partition and requires some modifications to allow its use in a virtualized CMP.

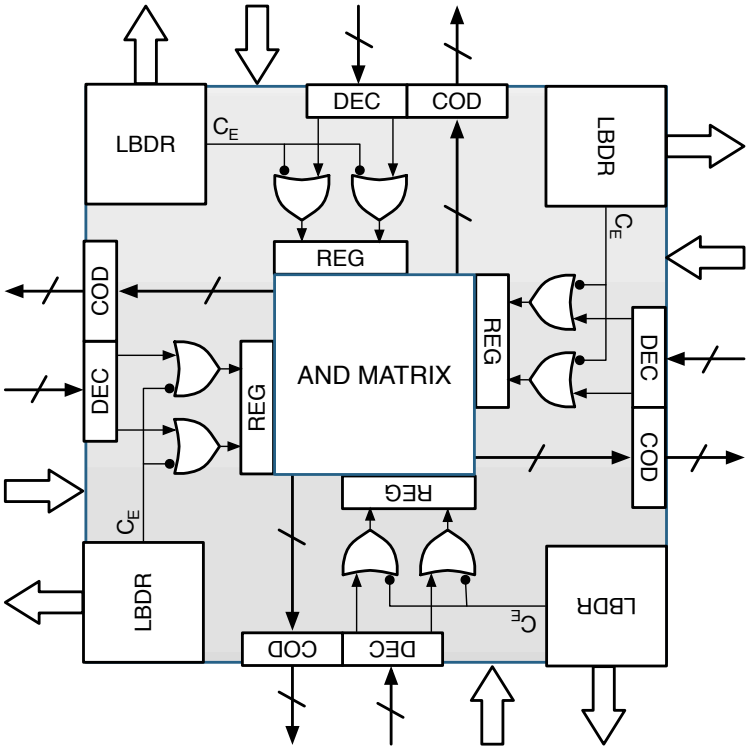


FIGURE 5.5: pNC switch design.

Consider the example where the upper left tile (Tile 0)[2] sends a broadcast message due to an L2 bank miss. Tiles in the same column will collect their ACKs, which in turn will be collected in the first row. Tile 0 is notified when all the tiles trigger their signals. However, if broadcasts are limited within a partition due to LBDR connectivity bits, the tiles located outside the partition will not receive the broadcast and thus will not trigger the GN signal for Tile 0. The GN modules of the tiles at the partition boundary will never receive the signals coming from outside the partition.

LBDR connectivity bits can be used to solve this problem, as they define the boundaries of the partition. We need to link the connectivity bits with the GN infrastructure. Figure 5.5 shows how the GN module can use the connectivity bits to this purpose. The complemented value of the connectivity bit is ORed with the input GN signal at the partition boundary. Thus, that signal will be masked. Figure 5.6 shows a detailed example, where the partition boundary, represented by the C_S and C_E bits set to zero, set all the input signals to one for every GN destination, so no notifications is expected from outside the partition. Notice that this solution adds a delay of a single logic gate.

[2]Tiles are numbered from 0 to 63, starting from the upper left corner and descending row by row to the lower right corner of the CMP.

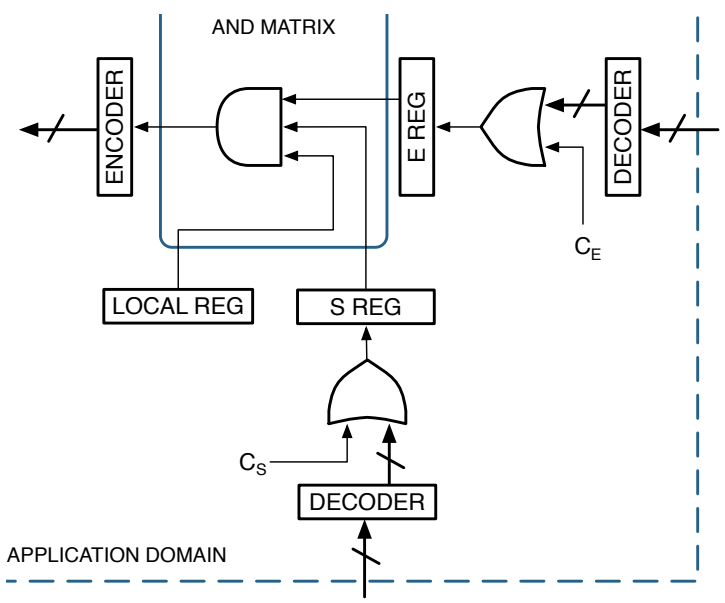


FIGURE 5.6: Example of GCN connected with LBDR bits.

Consider Tile 24, which is located at the lower left corner of the first (upper right) partition. According to Figure 5.4, the GN module of this tile can transmit its ACK to the upper tile (Tile 16) once two ACKs are received: the one sent by the local node, and the one coming from the tile located in the lower row (Tile 32). Since Tile 32 belongs to a different partition, it will never receive the broadcast message, so it will never send an ACK to Tile 0 through the GN. This issue is solved by combining the LBDR connectivity bit of the port that crosses the partition (south port at switch 24) with the GN signals coming from that direction. This way, if the tile connected through a port belongs to the same partition, its GN signal is enabled, otherwise it is always set to 1, thus being a neutral input of the AND gate. In the example shown in Figure 5.6, since Tile 32 belongs to a different partition, the south port connectivity bit of switch 24 is set to 0, so the GN signal coming from Tile 32 is always set to 1 (only the local input signal will be significative).

Generalizing, if LBDR is used to separate regions, the GN can work properly in a virtualized environment with a simple modification: in the AND logic block, each input to the AND gates must be the OR of the signal coming from another tile and the complemented LBDR connectivity bit correspondent to the port which communicates with that tile. This must be done for all inputs except for local signals.

5.2.2 LBDR Regions

Partitions defined by LBDR can be overlapped, thus allowing a tile to belong to different partitions. This is another useful feature with partitioning/virtualization for various reasons. First, two or more applications in different partitions can share several LLC banks (we will not use this possibility in this work). Second, it allows to define a global partition, including all the tiles, so that all of them can reach the memory controller (which is a shared resource).

Third, a partition can steal LLC banks from underutilized partitions (Home Stealing, HS). For instance, an application which is executed in a 16-core partition may have a working set which does not fit in the LLC capacity offered by the partition. The opposite may also happen. The LLC banks may be underutilized by the running application in that partition, which is then using resources that are actually not needed and possibly could be useful to an application which is executed in a neighboring partition.

In our final design, three connectivity bits per port are used. The first one defines the partitions to which applications are mapped to. We refer this to as Processor Partition (PP). We assume four different non-overlapping PPs, each including 16 tiles.[3] The second bit must allow each node to reach a shared memory controller. If a single memory controller is used, then the second bit must define a region as large as the whole chip.[4] A third bit is used to define Home Partitions (HP), thus enabling LLC partitions to differ from the shapes and sizes of PPs, enabling Home Stealing (HS). HPs may increase or reduce the default LLC capacity defined by the partition. Also, HS can be used if two applications share parts of their virtual address space. HPs of the two applications can be configured so that they partially overlap, and shared blocks can be mapped in the banks that belong to both partitions, at the boundaries.

Figure 5.7 shows an example with four partitions with different shapes. PP0 uses HS taking some LLC banks from PP2. In addition, PP2 and PP3 share 4 LLC banks by overlapping their HPs (HP2 and HP3). In summary, PPs set the cores on which an application is running, while HPs define the LLC banks which can be used by those cores. The MC must be aware which tiles belong to the PP and HP of an application to perform the block mapping to LLC banks.

[3] If overlapped PPs were used, then more connectivity bits would be required.

[4] Notice that with several MCs, the partitioning support will need defining disjoint MC partitions. However, we assume just one MC.

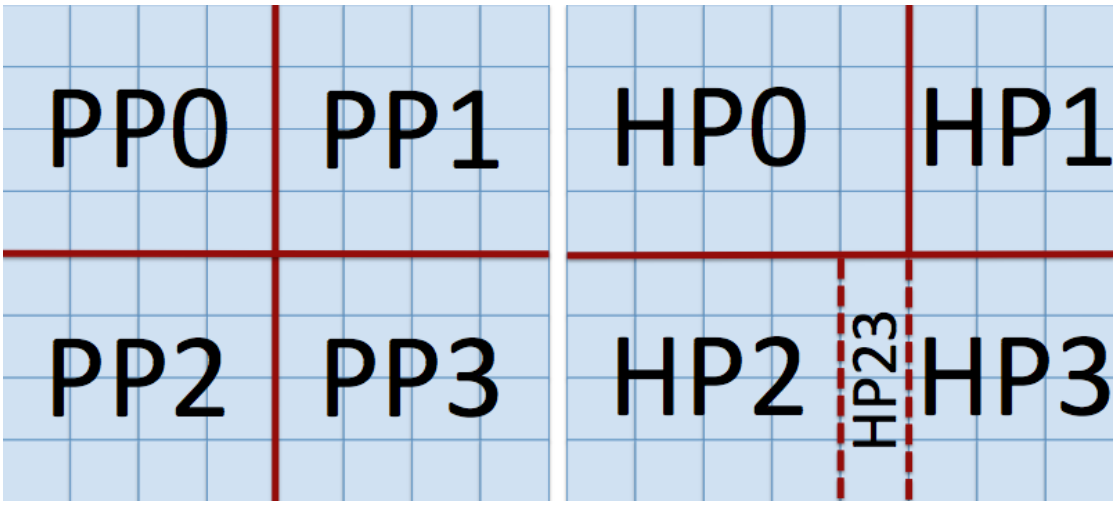


FIGURE 5.7: Processor Partitions and Home Partitions example.

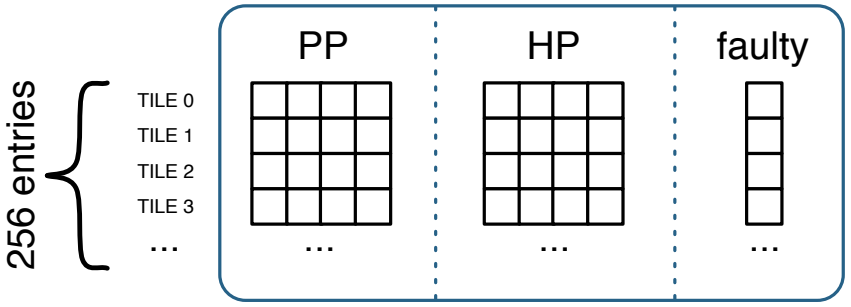


FIGURE 5.8: PP, HP and faulty tables at the MC.

It is worth to note that the configuration of LBDR bits to create partitions, and the definition of PP and HP partitions has to be taken by a higher entity layer, the hypervisor. Based on current application demands, the hypervisor, running in the OS, configures the chip appropriately. These dynamics are not explored in this thesis and are left for future work.

5.2.3 Memory Controller Design

To perform home mapping, the MC must be aware of the partitioning of the system, both for processors and home banks. Two small tables containing this information are initiated by the OS hypervisor, one table defining PPs and another defining HPs (Figure 5.8). Each table has as many entries as the number of tiles in the system, each entry containing one bit per possible partition ID (four in our case). For our 8×8 system, with up to 4 partitions, each table has a size of 256 bits to store the partitioning information. When a request is received at the MC, both tables are checked to compute the home

bank. Regardless the sizes of PP and HP partitions, the home bank is selected among those belonging to the HP. Blocks are mapped to the bank in the requestor's tile until it is full. Then, neighboring banks are checked within the HP set. Optimizations can be derived by using different strategies on shared HP partitions.

We consider only one memory controller. However, in CMP configurations with different MCs, partitions of the chip need to be partitioned between the different MCs. In all cases, a general global partition needs to be defined in order to allow the full utilization of the chip by a single application. In case of multiple MCs, the tiles belonging to a given partition must forward LLC misses to the proper MC. This can be achieved by a small register associated to each LLC bank.

LBDR provides fault-tolerance at NoC level. Orthogonal to LBDR's NoC-level fault-tolerance, pNC provides fault-tolerance at LLC-level, allowing the CMP to still work even when some LLC banks are faulty. This is achieved by an extra bit used in the MC for each tile. When this bit is set then the LLC bank is set as faulty and avoided when mapping blocks. Note that any combination of faults is supported.

Fault-tolerance support can not be achieved if static mapping is used, neither with First-Touch nor with the S-NUCA approach. Both techniques rely on the static mapping of block addresses to LLC banks, thus in the presence of a faulty bank, all the blocks that should be mapped to that bank can not be accessed. The flexibility of pNC allow to avoid mapping blocks to faulty LLC banks, distributing the blocks in neighboring banks.

5.2.4 Mapping Algorithm

The mapping algorithm we implemented at the MC is a slightly modified version of the one saw in the previous chapter, and it is described by the pseudocode of Figure 5.9. A block is mapped to the same tile of the requestor if the local bank is not faulty and if the HP ID of the local bank matches the PP ID. If the local bank is full, neighboring banks are checked. An external bank can only be chosen if it is not faulty and if its HP ID matches the requestor's PP ID. In case there are not faults and there is only one partition including all the tiles, the mapping algorithm defaults to the one seen in Chapter 3.

```
int function allocate(int r, address a) {
   banklist n; bank b; set s; bank h;

   s = get_set(a);
   if((HP[r] == PP[r]) && !Faulty[r] && alloc[r,s]<num_ways)
                                                          {alloc[r, s]++; return r;}

   for(int h = 1; h <= MaxHops; h++){
     n = BanksReachable(r, h);
     for (int i = 0; i < size(n); i++){
       b = SelectBankClockWise(n, i);
       if (HP[h] == PP[r]) && !Faulty[h] && alloc[b,s]<num_ways)
                                                          {alloc[b,s]++; return b;}
     }
   }

   for(int h = 1; h <= MaxHops; h++){
     n = BanksReachable(r, h);
     for (int i = 0; i < size(n); i++){
       b = SelectBankClockWise(n, i);
       if (HP[h] == PP[r]) && !Faulty[h] && alloc[r,s] - alloc[b,s] > UtilThr)
                                                          {alloc[b,s]++; return b;}
     }
   }

   alloc[r, s]++; return r;
}
```

FIGURE 5.9: Mapping algorithm performed by the MC in pNC.

5.3 Evaluation

To evaluate pNC in a virtualized environment, gMemNoCsim was fed with a set of memory access traces obtained running applications from SPLASH-2 and PARSEC benchmark suites in Graphite simulator. Altought gMemNoCsim can be embedded in Graphite, Graphite can not run multiprogrammed workloads, thus the choice to use memory access traces. Synchronization events between threads (barriers) have been included in the traces.

Three mapping policies were implemented:

- S-NUCA, a baseline configuration where blocks are statically mapped to L2 banks using the less significant bits of the block address.
- FT, an S-NUCA configuration in which the blocks are mapped to the L2 banks using a First-Touch policy; the first time a block is requested, the memory page containing that block is mapped to the L2 bank in the requestor's tile; we assumed 4KB as the page size.

Routing	LBDR	L1 cache size	32 kB
Flow control	credits	L1 tag latency	1 cycle
Flit size	8 bytes	L1 data latency	2 cycles
Switch model	4-stage pipelined	L2 bank size	256 kB
Switching	virtual cut-through	L2 tag latency	1 cycle
Buffer size:	9 flit deep	L2 data latency	4 cycles
Virtual channels:	4	Cache block size	64 B
GCN hop delay	1 cycle	Memory controllers	1

TABLE 5.1: Network parameters (pNC evaluation).

- pNC, where blocks are mapped using the basic RHM within the HP partition of the application requesting the block.

We assume an 8×8 CMP system divided in 4×4 regions. Network and cache parameters are shown in Table 5.1. As we use one MC, one region defined by the connectivity bits must cover the whole chip, thus allowing all the tiles to reach the memory controller.

5.3.1 pNC Overhead

Compared to the basic CMP design with static mapping, pNC introduces an area overhead due to the tables at the MC and to the network optimizations. The MC needs three tables: the first two are PP and HP tables, each with 64 four-bits entries (512 bits in total); the third table stores the utilization statistics of each LLC bank and is used by RHM to balance the utilization of LLC banks, as described in Chapter 4. For a 16×16 tile system with 16-way 256KB LLC banks, the minimum memory requirements for this table is 2KB. For fault-tolerance, an additional 64-bit register is used (one bit per tile).

At network-level, switches include the sequential GN module. As we saw in the previous chapters, this logic introduces a 3.1% area overhead at each switch. The GN also requires 8 additional wires per port per direction in our 8×8 system: 6 to encode the ID of the destination node, one to encode the *HIT* flag and one to encode the *RETRY* flag.

5.3.2 Performance in Fault-Free Systems

We analyze first how the three mapping policies behave when the same application is running in the four regions, assuming that the four regions do not share any code or data. Figure 5.10 shows the execution time when the three mapping policies are

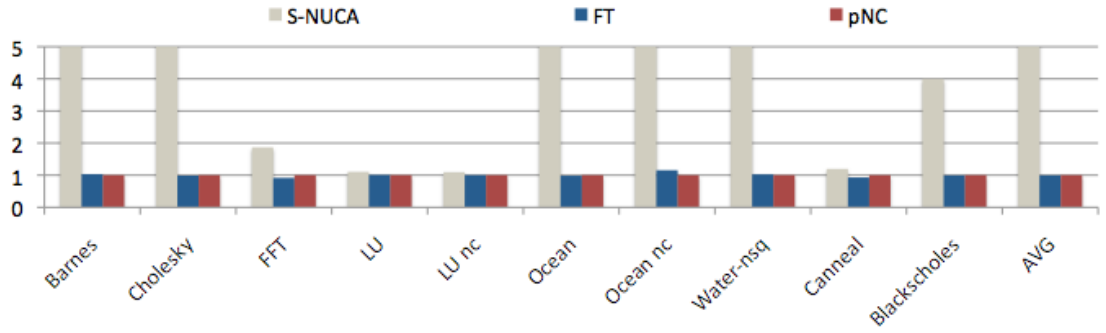


FIGURE 5.10: Normalized execution time.

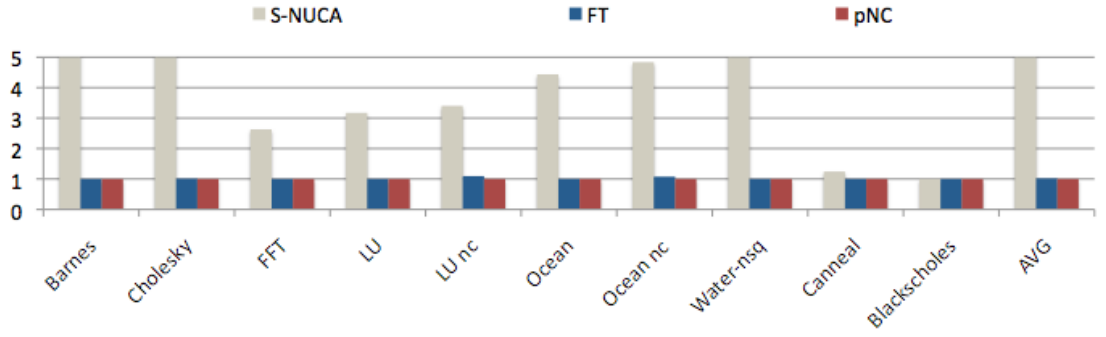


FIGURE 5.11: Normalized L2 misses.

used, normalized to the pNC case. First-Touch and pNC have similar performance results, while static mapping performs much worse. When L2 cache banks are not excessively stressed, the performance of static mapping is just slightly degraded (FFT, LU, LUNC and CANNEAL) compared to pNC and First-Touch. However, as soon as many frequently accessed blocks are mapped to a single bank exceeding its capacity, performance of static mapping quickly degrades and execution time can rise up to 50-60X (BARNES and WATER-NSQ, not shown as the scale is limited to 8).

Figure 5.11 shows the number of LLC misses for the previous applications, normalized to the pNC case. Again, pNC and First-Touch have similar behavior while the number of misses with static mapping is generally higher and can rise up to 700 times (e.g. BARNES), as some LLC banks are highly demanded by the four applications (many frequently used blocks are mapped to the same bank, causing the replacement of blocks which are actually still being used). Also, the average miss latency (not shown) is 81% higher than in the pNC case. LLC banks are further away from requestors.

Let us see now what happens with an heterogeneous workload. Since different applications have different needs in terms of resources, we can now enable the Home Stealing

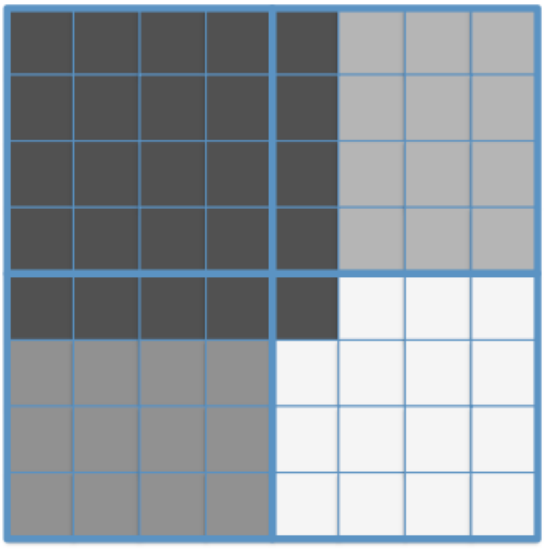


FIGURE 5.12: Home stealing configuration.

property of pNC (pNC-steal). With HS enabled, some cache banks of a region can be used to increase the L2 cache capacity of another partition with higher cache demands.

Figure 5.12 shows the HPs we assume for next evaluation results, where the top-left region is stealing LLC banks from the other regions.

We use the three application scenarios shown in Table 5.2. Regions are numbered from 0 to 3 starting from the upper left and descending to the lower right. The sets are built mixing an application which has high cache capacity demand, which we map in P0, and three applications with lower demands.

Figure 5.13 shows execution time when each set of applications is executed using static mapping, First-Touch, pNC, and pNC-steal. The figure shows the execution time of the application with the higher execution time (Ocean for Set 1, Ocean non-cont. for Set 2, Canneal for Set 3). This time S-NUCA shows much lower performance degradation. The heterogeneity in applications leads to a better balance of L2 bank sets utilization (indeed, with the third set, it achieves even better performance than the rest). FT and pNC have similar average performance, one performing slightly better that the other for different sets. Cache stealing, however, contributes to reduce execution time by 6% in Set 2 and 7% in Set 3, while it has a negligible impact on Set 1.

To take a deeper look to each set, Figure 5.14 shows the execution time for each application of Sets 1, 2 and 3 respectively. Note that the execution time in Figure 5.13 corresponds to the first set of bars in Figure 5.14 (application mapped in partition P0

	P0	P1	P2	P3
Set 1	Ocean	FFT	LU non-cont.	Cholesky
Set 2	Ocean non-cont.	LU	Barnes	Water-nsq
Set 3	Canneal	LU	Barnes	Blackscholes

TABLE 5.2: Sets of applications executed in the CMP.

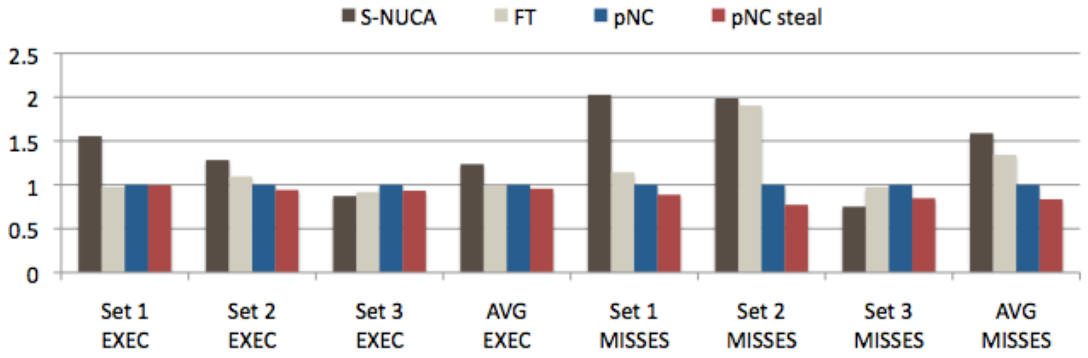
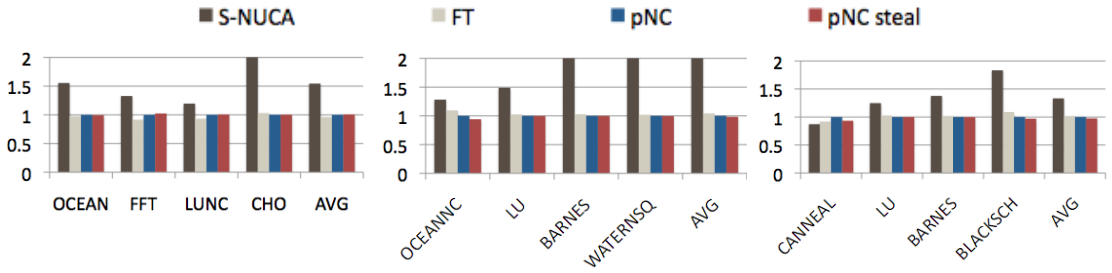


FIGURE 5.13: Normalized execution time and L2 misses (mixed applications).



(a) Set 1 (b) Set 2 (c) Set 3

FIGURE 5.14: Normalized execution time for each application of the three sets.

takes always more time to execute as it is larger). Looking at each single application, we can again notice the performance degradation of S-NUCA, while performance of FT and pNC are similar, with pNC performing slightly better except for CANNEAL, FFT and LUNC. The aim of cache stealing is to expand the LLC capacity of the application running in partition 0 without affecting the performance of the applications running on other partitions, especially on P1 and P2, where LLC capacity is reduced by a quarter. This is achieved both for Set 2 and Set 3, while in Set 1 the FFT mapped into P1 suffers the reduced LLC capacity. In this case, the hypervisor chose a wrong partitioning for the applications and a wrong HS policy.

Figure 5.13 shows also the number of L2 misses for the three sets of applications, normalized to the pNC case. pNC is effective in reducing the number of L2 misses, and Cache Stealing allows a further reduction: on average, the number of misses with static

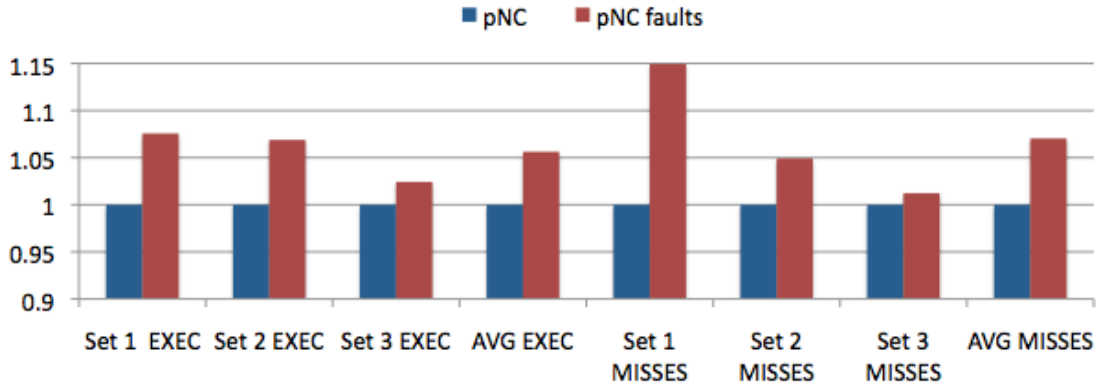


FIGURE 5.15: Normalized execution time and L2 misses (mixed applications) with faulty LLC banks.

mapping is 58% higher than those of pNC, and 34% higher with First-Touch, while pNC-stealing reduces the number of misses by a 17% compared to pNC.

5.3.3 Performance in Faulty Systems

Finally, we evaluate the performance of pNC when some LLC banks are faulty. We run the sets of applications shown in Table 5.2 on a CMP system with 10% randomly-selected faulty LLC banks. Figure 5.15 shows the execution time for the three sets of applications, normalized to the case of a fault-free CMP. The performance degradation ranges from a maximum of 15% (Set 1) to a minimum of 1.5% (Set 3), with an average of 7%. Figure 5.15 shows also the normalized number of misses. When faults are present, the number of misses increases from a minimum of 2.2% (Set 3) to a maximum of 7.8% (Set 1), with an average of 5.8%. pNC performs home mapping even with some faulty banks, and distributes the load between banks minimizing the impact of faults. Note that static mapping and First-Touch can not be used in a system with faulty LLC banks.

5.4 Conclusions

In this chapter we presented pNC, an hardware substrate to independently partition cores, NoC resources and LLC banks in a virtualized CMP system. It is based on the combination of LBDR, which is used at the network level to define the partitions, and RHM, which is used at the LLC level to map blocks within a partition. Through pNC, the partitioning of cores and LLC banks is decoupled: CMP resources can be

independently partitioned to fit the requirements of an application; two LLC partitions can overlap, thus allowing two applications to share code or data. The memory controller is aware of the chip partitioning, and based on it performs the mapping of cache blocks to the LLC banks. Evaluation results with traces obtained from SPLASH-2 and PARSEC applications show that pNC performs slightly better than First-Touch in a virtualized CMP. It must be taken into account that we used the basic RHM mechanism to evaluate pNC, so performance would noticeably improve with block migration/replication enabled and with Hammer protocol.

Hammer protocol would be particularly appealing in a virtualized CMP: the directory, indeed, requires a sharing code with as many bits as tiles in the system. When the system is partitioned, only a portion of these bits in each partition is used; for the 8x8 system considered in this chapter, each LLC entry includes a 64-bit sharing code, but due to the chip partitioning only 16 bits are actually used in each entry (75% of the directory is thus not used). Eliminating the area overhead and performing better than Directory when combined with RHM, Hammer protocol appears then as a better choice for this kind of systems.

Future work directions involve mainly the role of the MC. In this chapter, we assumed an hypervisor to define the PPs and HPs before an application starts. As far as the PP is concerned, that is a good assumption: the OS indeed can choose the optimal partitioning of cores and the mapping of applications to those cores. The cache requirements however are best monitored on-chip by the MC, as we mentioned in Chapter 4. Rather than relying on a LLC partitioning performed by the hypervisor before the execution begins, an application may start with an HP equal to its PP. The MC would then be in charge of resizing the partitions at runtime, depending on the utilization of each HP. Note that the dynamic resizing of partitions is already supported by LBDR.

Chapter 6

Heterogeneous LLC Design

Last-level caches play an important role in CMP systems since they constitute the last opportunity to avoid expensive off-chip accesses. Current and future CMPs will be equipped with increasingly larger LLCs, occupying a significant fraction of the total chip area and having a large contribution on the global chip leakage energy (as an example, the size of the LLC of the Intel Core i7 can reach up to 15MB as by today [63]). Current LLC implementation can be optimized to reduce its area and power requirements without affecting the overall system performance. In particular, this chapter describes a reorganization of the LLC structure to allow the dynamic reallocation of entries at this cache level depending on the specific necessities of every block. This contribution is orthogonal to the previous ones and can be used with or without the RHM strategy and the GN. In this chapter we focus only on the LLC reorganization, thus not using the previous optimizations.

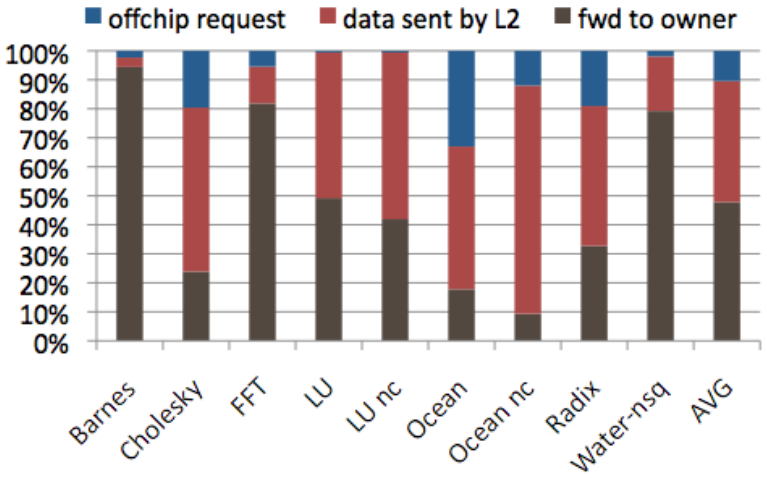


FIGURE 6.1: Breakdown of actions performed by the LLC when an L1 request is received.

6.1 Motivation

The LLC reorganization is motivated by the observation that for private blocks (cache blocks accessed by a single core) the LLC contains stale data (or, more precisely, the data portion of the cache entry is not needed, since block requests are forwarded to the owner L1). While all the cached copies of a shared block have the same value, both in the private and the LLC caches, in the case of a private block (the block is owned by a single private cache) the values of the LLC and the private copy may differ (the block could be in modified state in the private cache). Therefore, the LLC copy of a private block is probably stale, and is not used until the corresponding private cache performs a *writeback* operation on the block, when the only valid copy of the block is moved to the LLC. This way, there are two cases in which the data portion of an LLC entry is needed: a) the block is shared by several cores, and b) a private block has been replaced by the owner private cache. In the rest of situations, keeping the data portion of the LLC entries can be seen as a waste of resources. Figure 6.1 plots the breakdown of actions performed by the LLC of the 16-core CMP assumed in this thesis when different SPLASH-2 applications are simulated using gMemNoCsim connected to Graphite. Almost 50% of requests on average (and more than 80% in some cases) are forwarded to the owner L1 cache, without using the information stored in the data portion of the LLC entry. Therefore, we can see that a large percentage of area and power is wasted in LLC to store private blocks.

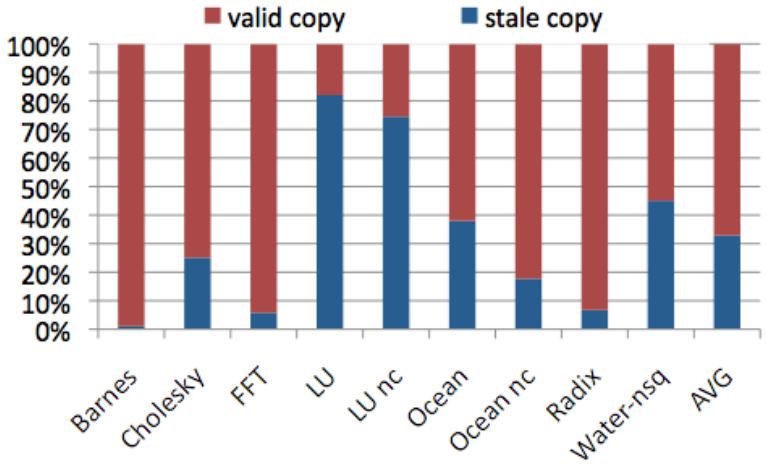


FIGURE 6.2: Percentage of stale and valid blocks replaced at the LLC.

On the other hand, LLC replacements (due to limited capacity and associativity) lead to invalidate data in private caches even if they are still accessed. Figure 6.2 shows the percentage of LLC replacements of private blocks. As derived from the figure, a large percentage of replacements in the LLC (more than 40% in some applications) involve private blocks. Private data is only requested the first time a processor wants to operate on it and then is written and read without sending any request to the LLC, thus becoming older in the LLC set and thus becoming quickly selected by the LRU replacement algorithm (even if it is being actively referenced by a core).

Taking those results as a reference, we can conclude that an effective organization for the LLC should combine two types of entries: entries which include just the tag portion and the directory information for private blocks, and regular entries for the rest of the blocks, including the tag, the directory information and the data blocks. This reorganization of the LLC structure allows the dynamic reallocation of blocks depending on the block being *private* or *shared*. In particular, the tag array and the associated directory can be redesigned with a different (higher) associativity than the L2 data array. The tag and directory array contains information about all the cached blocks, while the data array contains only shared and replaced private blocks. This allows for a smaller LLC with the same performance as private blocks will be kept only at private caches. With this approach, large savings in static power can be achieved without hurting performance.

This proposal can be combined with previous works that aim for reducing static power at L2 caches by dynamically powering down cache lines. This is the case of [40], where LLC entries for private blocks (once the L1 cache writes on the block) are powered down. This

means extra power saving through a line-level power gating mechanism (similar to cache decay for L1 caches [38]). These strategies are orthogonal to the approach described in this chapter, and section 6.3 will provide results for the two mechanisms combined together.

This chapter is organized as follows: in Section 6.2 we describe our proposal and its impact on area and power overhead. In Section 6.3, we perform a detailed analysis of performance and power savings. Then, conclusions are discussed in Section 6.4.

6.2 Dynamic L2 Cache Line Allocation

Figure 6.3 shows the FSM for an L2 block described in Chapter 2 (transient states are omitted for the sake of clarity). At first, the block is not cached on chip (state *I*). Upon a write (GetX) or a read (GetS) request, the block is fetched from main memory and sent to the L1 requestor, which is now the owner of the block and holds a private copy (state *P*). At this point, a write request from another core will be forwarded to the owner, which will send the data to the requestor and invalidate its copy (the requestor will become the new owner). A read request (GetS) will also be forwarded to the owner, but in this case it will not invalidate its copy. Instead, it will provide the data to the requestor but also keep a copy of the block with read-only permission. The block state in the home L2 bank will switch from *P* to *S*.

If the block is in *P* state and the owner replaces the block (PutS or PutX), then the only on-chip copy of the block is held by the home L2 bank, which switches to state *C*. Further requests will be served using the L2 copy of the block.

Usually, there is a 1:1 relationship between the tag/directory and the data portions of the LLC. A different design can be chosen, with fewer data entries than tag/directory entries. If the block is private (state *P* in the directory), then only the tag/directory entry is allocated, as the only valid copy will be held by the owner L1. If, on the contrary, the block is shared or only cached in the L2 bank (state *S* and *C* in the directory), then both the tag/directory entry and the cache line are allocated.

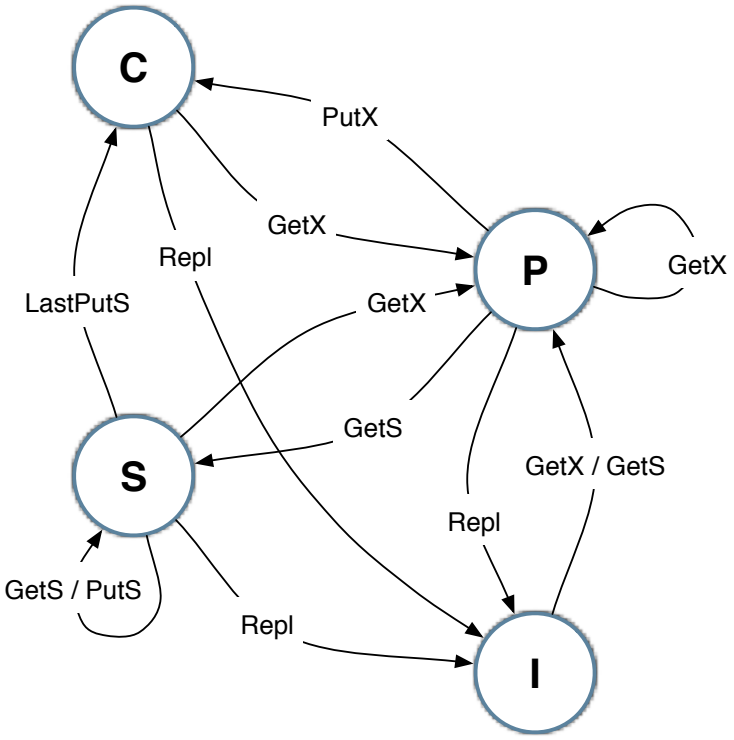


FIGURE 6.3: LLC finite state machine (MESI protocol).

A LLC bank can be reorganized by reducing the associativity of the data entries and keeping the number of tag/directory entries or using the saved area for more tag/directory entries in the directory structure, breaking in both cases the 1:1 relationship. For instance, we can keep the associativity of the tag/directory array to 16 while reducing the L2 data array associativity from 16 down to 8. This means that only the first eight ways of the tag array can store information about a block in state S or C, which data will be saved in the corresponding way of the data array, and all the 16 ways can store information about a private block. Note that this will constrain the area devoted to a shared data in L2 but will not compromise private data, which is tracked by the directory and stored in the L1 caches.

It must be noted that one could think of reducing L2 size by reducing the number of sets, instead of the number of ways (see Figure 6.4). However, this could compromise the cache capacity depending on the data set. Indeed, having less sets would lead to have less entries for shared data, while by reducing the associativity, cache capacity for shared data will not be compromised (as long as shared data does not conflict in the same set). We propose to reduce ways as those are, expectedly, used by private blocks.

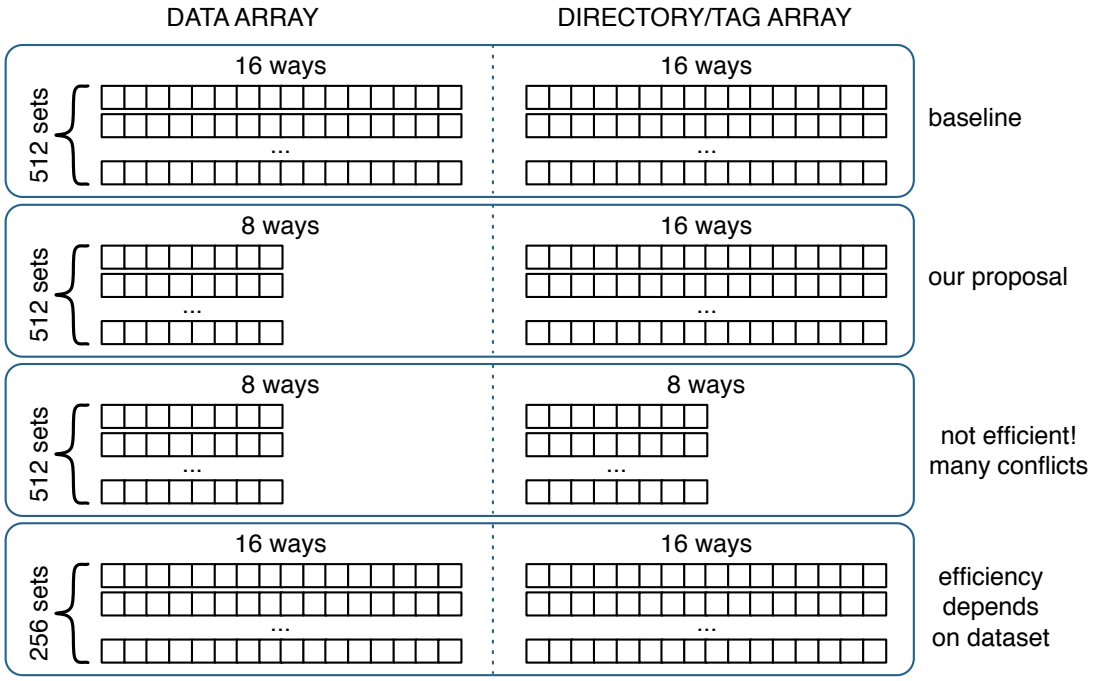


FIGURE 6.4: Different LLC configurations by changing the number of sets and the associativity of the L2 and directory structures.

Another direction one could take is to reduce in the same degree both the associativity in the L2 and in the directory (see Figure 6.4). However, in that case we would incur in high performance penalty (as will be seen later) as the associativity in the directory must be proportional and in line with the associativity in L1 caches. Simply, each directory set will not have enough ways to store information for all the blocks present in L1 caches.

Figure 6.5 shows an example of an L2 cache reorganized according to the scheme proposed in this chapter. Only the first four ways of the tag/directory array may keep information about shared or cached blocks, which will be saved in the same way of the data array. The remaining four sets of the tag/directory array are devoted to private blocks. However, private blocks can also be mapped in the first four ways of the tag/directory array. In this case, the information included in the corresponding set of the data array would not be useful.[1]

Assuming an L2 cache with a 4-way data array and an 8-way tag/directory array, when a block switches from P to S or C, its entry must be moved if it doesn't lie in the first four ways. This may trigger the replacement of another block if all the first four ways

[1]With orthogonal power saving techniques as [40] these entries can be powered down. Its impact is later analyzed.

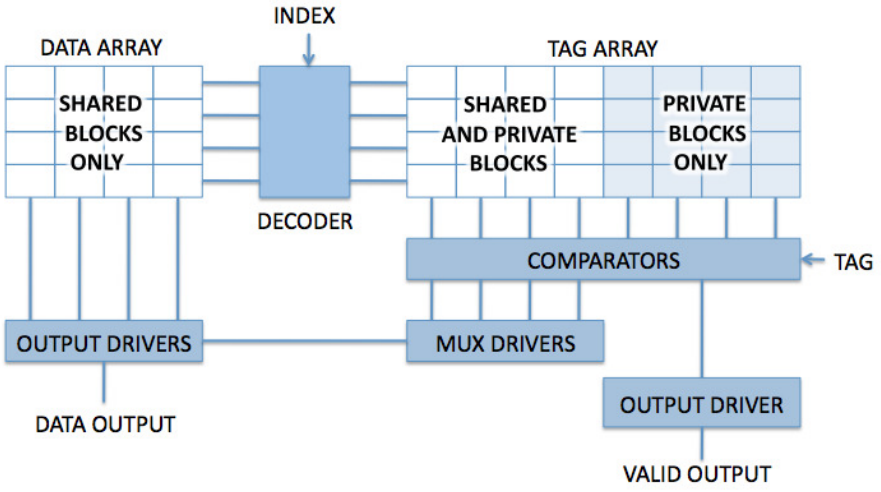


FIGURE 6.5: Example of LLC reorganization.

are already in use, which means that the data array set is full. Thus, the automata to manage the L2 entries needs to be slightly modified.

6.2.1 Replacement Policy

An LRU counter per way is used to implement the replacement policy. The counters are used in the classical way: each time the L2 receives a request for a block, all the counters with a lower value are incremented and the block counter is set to zero. When a new block is requested and all the set entries are already in use, the entry with the highest LRU counter value is selected. With the organization we propose, replacements can be triggered also when a block which is already cached must be saved in the L2 cache. As shown in Figure 6.3, this happens when an owner invalidates its private copy or a read request is received for a private block: the L2 state switches from P to C in the first case or from P to S in the second case. In both cases, a data entry must be allocated for the block in the L2 cache. If the first four ways are already allocated to other blocks, the replacement policy will choose the way with the highest LRU counter only between the first four entries (even though the entry with the highest LRU value could be one of the remaining ones). Note that LRU counters are updated for all the directory entries, so two entries cannot have the same LRU value.

Figure 6.6 shows an example of the policy described above. The rows represent the evolution of an LLC set when different requests are received, with the LRU value specified

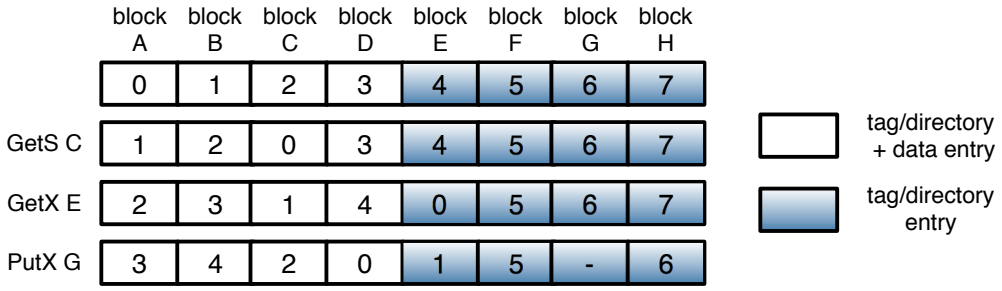


FIGURE 6.6: Replacement policy.

for each entry; starting from the upper row, the LRU values are updated for all entries when a read request is received for a block stored in a tag + data entry (GetS C) and when a write request is received for a block stored in a tag/directory entry (GetX E). When block G, stored in the tag/directory array, is replaced (PutX G), it must be moved to a tag + data entry, so one entry must be replaced; the victim is block D, which has the highest LRU value among the blocks stored in the tag + data array.

6.2.2 Dynamic Power Techniques

With the proposed L2 reorganization, truly shared data is promoted in the L2 data array, and private data is just tracked in the directory. However, it may happen that for a given set more than half of the entries are for private blocks, thus not all the entries in the L2 cache will be used. In such situation, these L2 data entries are wasting energy. To mitigate this effect, the L2 reorganization can be complemented with dynamic power-off strategies as [40], in which L2 entries that store private blocks are powered down. This can be achieved by using "sleep transistors" at each cache line to eliminate the most part of the leakage current, as proposed by Kaxiras et al. [38] for L1 caches. As in [38], we use Powell's gated Vdd design [29] at cache line level, inserting a transistor between the ground and each L2 cache line to reduce the leakage current to a negligible level.

When combined, the different transitions of a block A in the L2 cache can be summarized as follows:

- When an L1 cache requests block A, which is not cached on-chip, the L2 issues an off-chip request, allocates a tag entry to the block (which can be any of the

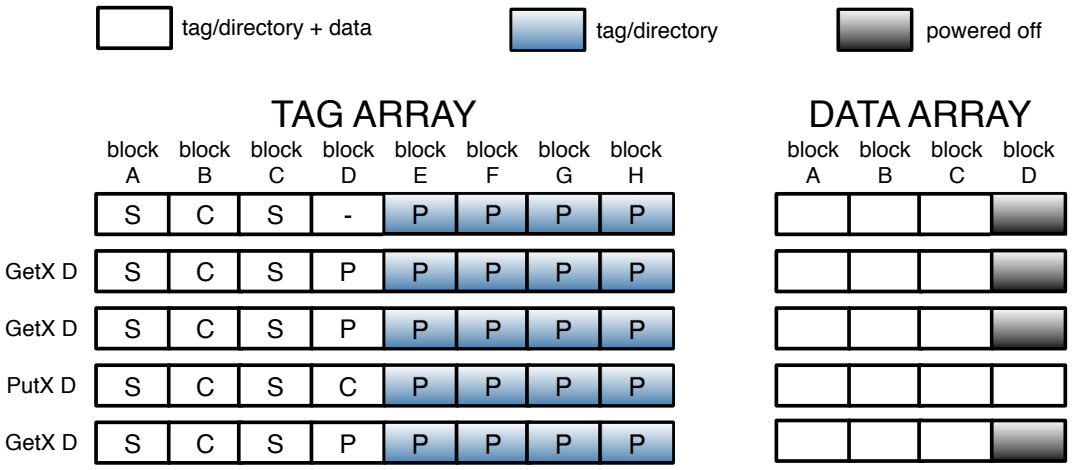


FIGURE 6.7: Evolution of a block when dynamic power-off techniques are used.

entries) and marks the block as private; if the tag entry is one of the first half, its corresponding data entry is powered-off.

- Subsequent write requests will cause a change of the owner but the state of the block in L2 cache will remain P.
- When the owner replaces its copy, it must be saved in the L2 cache. The same happens if the L2 receives a read request for a private block, which will become shared. If A is mapped in the first half of the tag ways, the corresponding data entry must be powered-on. If, on the contrary, A is mapped in the second half, it will trigger a potential internal swap, selecting one entry from the first half (using the LRU algorithm). The selected entry can be in one of the following states:
  - Private: this block and A are swapped, the data entry is powered on to save the write back copy of A.
  - Shared/Cached: this entry will be replaced and allocated to A.
- If a write request (in case A is in state S) or any request (in case A is in state C) is received, the block will be again treated as a private block and the data entry must be powered-off.

Figure 6.7 shows an example of block D evolving as described above.

6.3 Performance Evaluation

The modifications to the basic directory-based protocol needed to implement the new block allocation and replacement policy were implemented in gMemNoCsim. Five different L2 designs have been evaluated and compared using gMemNoCsim connected to Graphite:

- **L2-512-16_D-512-16**. A 512KB L2 bank with 512 sets and 16 ways. The directory also has 512 sets and 16 ways (1:1 ratio). This is the baseline for comparison.
- **L2-512-8_D-512-16**. A 256KB L2 bank with 512 sets and 8 ways. The directory also has 512 sets but keeps 16 ways (1:2 ratio).
- **L2-512-4_D-512-16**. A 128KB L2 bank with 512 sets and 4 ways. The directory keeps the same with 512 sets and 16 ways (1:4 ratio).
- **L2-512-2_D-512-16**. A 64KB L2 bank with 512 sets and 2 ways. The directory keeps the same with 512 sets and 16 ways (1:8 ratio).

In addition, we use a smaller L2 cache of 256KB as the baseline. In this case (**L2-256-16_D-256-16**), 256 sets and an associativity of 16 is used for both the L2 and the directory. Our designs on top of this baseline are **L2-256-8_D-256-16** (1:2 ratio, 128KB), **L2-256-4_D-256-16** (1:4 ratio, 64KB), and **L2-256-2_D-256-16** (1:8 ratio, 32KB).

The system is made of 16 tiles with one processor in each tile and with a private 32KB L1 data cache (with 256 sets and 4 ways). Each tile includes also the L2 bank and the associated directory. All the tiles are connected through a 2D mesh topology using the XY routing algorithm. In a first test every configuration does not incorporate any sleep transistor technology. Later we evaluate the combination of our technique with those. We ran various SPLASH-2 applications with these cache organizations.

Figure 6.8 shows the execution time for the different applications, normalized to the case of the first baseline L2-512-16_D-512-16. As can be seen, some applications have no impact on execution time when L2 banks are reduced. Indeed, in BARNES, CHOLESKY, LU, LUNC, and WATERNSQ, the L2 could be reduced by a factor of 8 (L2-512-2_D-512-16) with practically no impact on performance. On the other hand, other applications

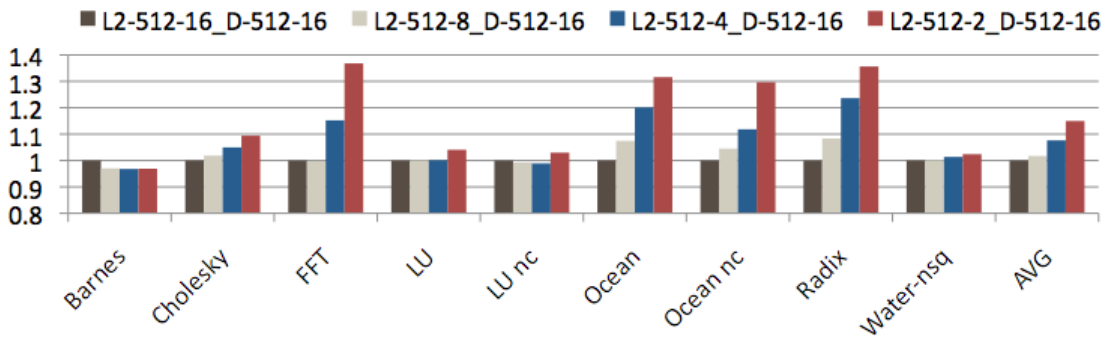


FIGURE 6.8: Normalized execution time. MESI protocol with L2 banks with 512 sets.

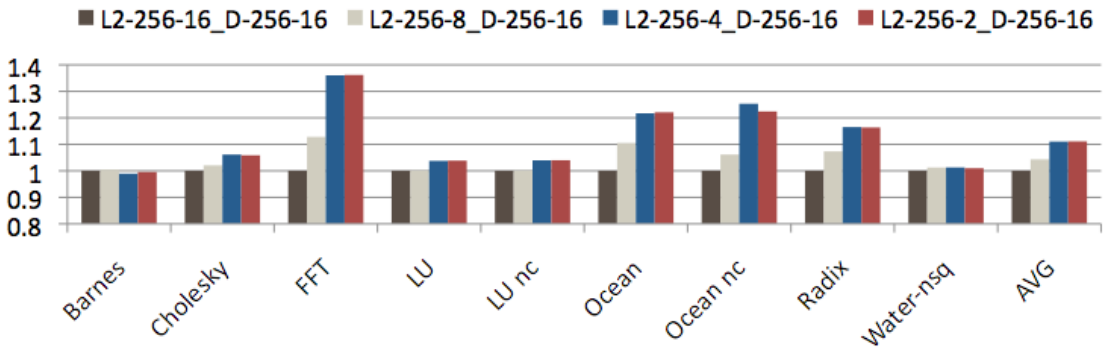


FIGURE 6.9: Normalized execution time. MESI protocol with L2 banks with 256 sets.

can be sensitive to L2 cache capacity to shared blocks. Anyway, by averaging, we can see that a good tradeoff is reducing L2 cache by half, which on average leads to only 1.7% performance decrease. Further reductions in area will tend to 7.5% performance degradation for an L2 reduction factor of 4 and to 15% for a reduction factor of 8.

For the case of smaller L2 banks (those with 256 sets), Figure 6.9 shows the execution time of applications. We can see similar trends with large savings (up to a factor of 8x) in area with no performance degradation, and others with some impact (up to 35%). On average, a reduction of 2x in L2 size have no large impact.

We use Cacti 5.3 [58] to compute area and leakage for the different L2-directory configurations. In Figure 6.10 we can see how area needs compare to the different evaluated designs. Each component is normalized to the baseline design (L2-512-16_D-512-16). Tag array's area is the same for the first four designs, while in L2-256-16_D-256-16 tags take roughly half the area as the overall associativity is reduced. As far as data array is concerned, the area needs decrease with the associativity of each design. Even though L2-512-8_D-512-16 and L2-256-16_D-256-16 have the same data array size, the area of

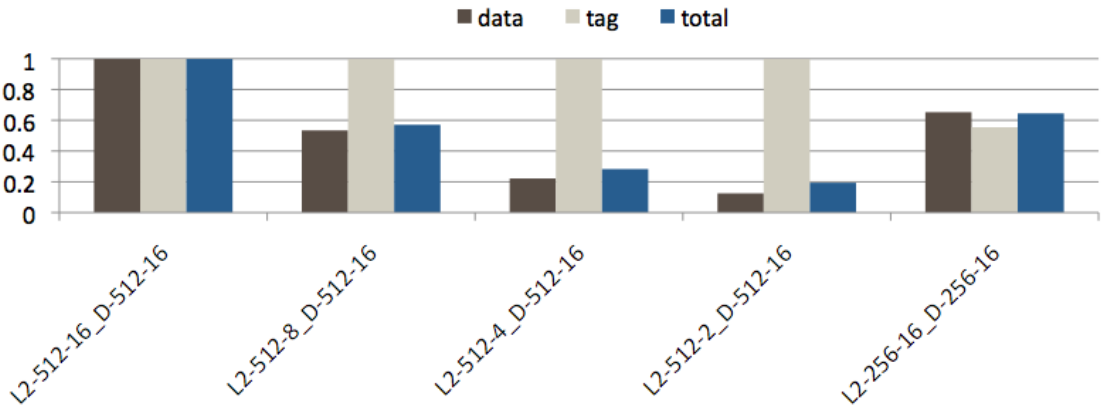


FIGURE 6.10: Normalized LLC area occupancy.

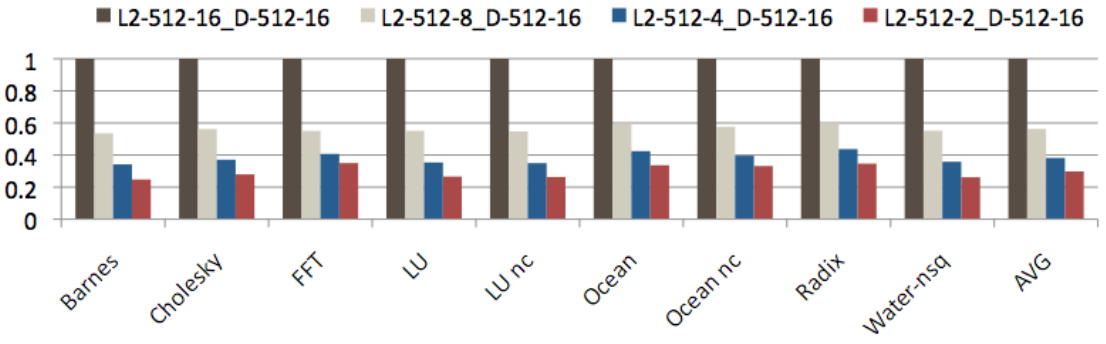


FIGURE 6.11: Normalized L2 leakage. MESI protocol.

the former is lower than the area of the later, due to its lower associativity (lower number of comparators).

Figure 6.11 shows the leakage energy consumed by the L2 banks taking into account the entire execution of each application, and normalized to the baseline (L2-512-16_D-512-16). Leakage is reduced up to 80% due to the cache reorganization. This saving is proportional to the reduction ratio performed.

Figure 6.12 shows leakage savings when our proposal is combined with [29]. We compare the previous four cases and a baseline 256KB cache (L2-256-16_D-256-16). In both baselines, caches have all data entries powered on during the whole execution time, while the proposals power-on the L2 data entries only when needed, as perviously described. The average leakage energy is reduced on average by 75% (for 1:2 ratio), 83% (for 1:4 ratio) and 89% (for 1:8 ratio). Depending on the application, we achieve up to 98% in leakage energy savings (BARNES).

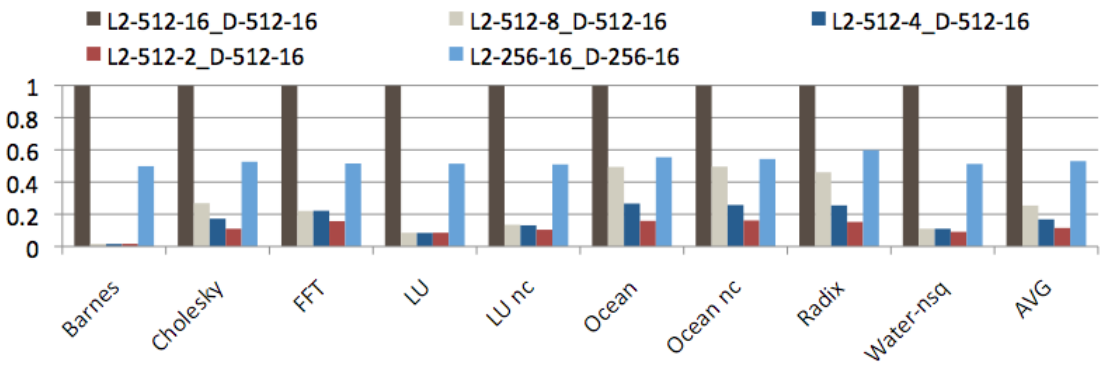


FIGURE 6.12: Normalized L2 leakage. MESI with sleep transistors.

Summarizing, if we consider L2-512-8_D-512-16, on average our proposal reduces the LLC size by more than 40% and reduces the leakage by more than 40% if considered alone, and by 75% if combined with dynamic power saving techniques, keeping almost unaltered the system performance in terms of execution time. Further reduction of the data array lead to higher savings in area (up to 80% on average) and leakage (up to 89% on average) with more noticeable performance degradation (execution time increases by up to 15% on average).

6.3.1 Benefits when Using MOESI Protocol

Another appealing protocol which can be implemented in L1 caches is MOESI. It behaves like the MESI protocol but when an owner (which has its block in state M or E) receives a forwarded GETS its state becomes O (and not S like in MESI protocol). This means it remains the owner of the block with read permission, and when the L2 receives a request, it will still be forwarded to this L1. This is inefficient in a typical memory hierarchy since it takes one more step to provide the data to the requestor, but can largely benefit from our approach. Indeed, shared blocks in state O do not need to be allocated in L2 banks, thus, being all the requests forwarded to the owner. The L2 state diagram when a MOESI protocol is used in L1 caches is shown in Figure 6.13. A block keeps switching between states P and O until the owner invalidates its copy, and only then an L2 cache line must be allocated. In Figure 6.14 we compare the execution time for the different cache configurations with L1s that use a MOESI protocol (the figure shows the cases for 512 sets). However, for the sake of fairness, we use the MESI protocol for the

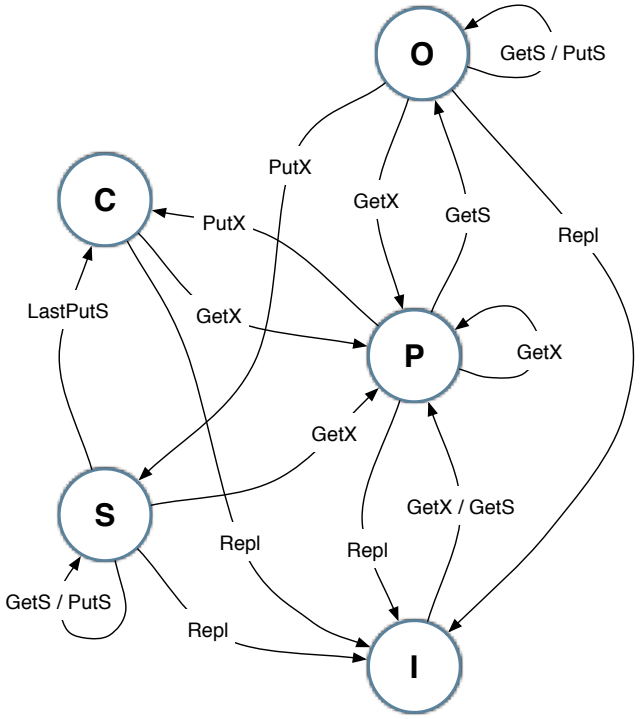


FIGURE 6.13: Simplified FSM for the L2 cache (MOESI protocol).

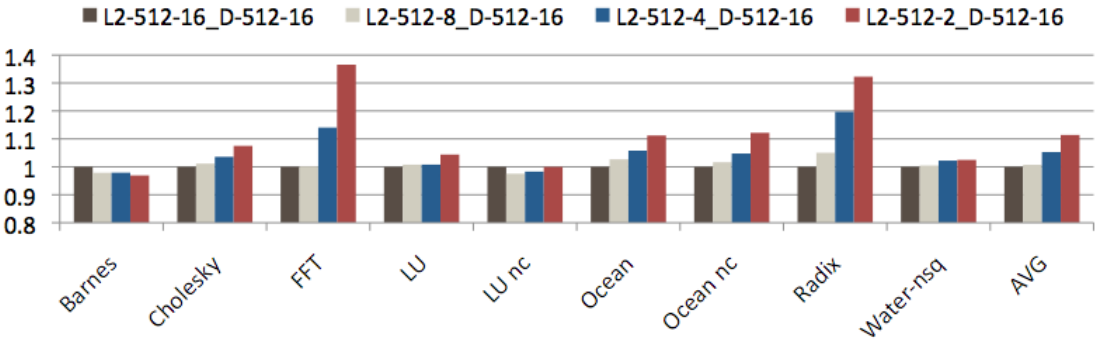


FIGURE 6.14: Normalized execution time. MOESI (for 1:x ratio proposals) and MESI (for baseline).

baseline configuration (which works better than when using MOESI, due to the extra indirection). For our proposals, however, we use the MOESI protocol.

As shown, our organization takes advantage of the O state since less blocks in state S and C must be invalidated. The average penalty when using a data array with lower associativity is now 0.7% (with a 1:2 ratio), 5.2% (with a 1:4 ratio) and 11.4 % (with a 1:8 ratio), while saving 43%, 72% and 81% of area and 75%, 82% and 83% of static energy, respectively (Figure 6.15).

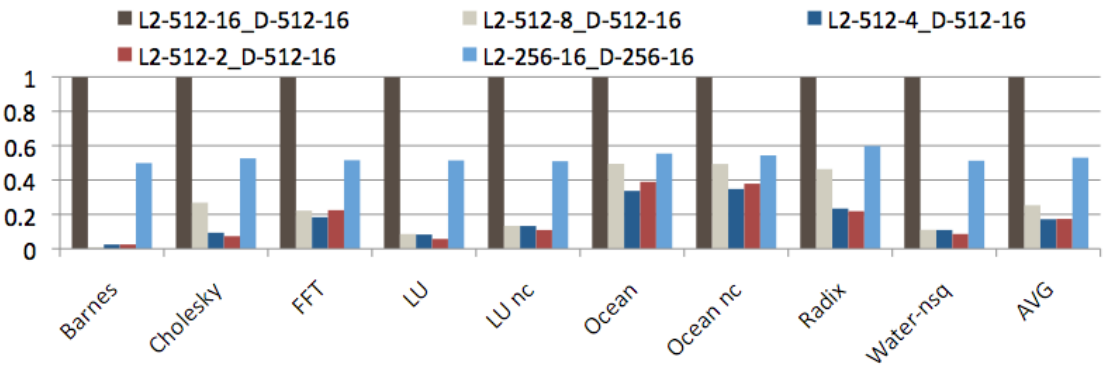


FIGURE 6.15: Normalized L2 leakage. MOESI with sleep transistors (for 1:x ratio proposals) and MESI (for baseline).

6.4 Conclusions

In this chapter we presented an heterogeneous LLC design with two different types of entries: the first type is present both in the data and in the tag/directory array, and is used to store shared blocks and blocks which are present in the LLC but not in private caches; the second type is only present in the tag/data array and is used to store the information about private block, while the actual data is only stored in the owner's private cache. A cache block is dynamically allocated to one or the other type of entries depending on its state. The proposed technique, aimed to reduce area and leakage of the LLC, can be complemented with previously proposed techniques that dynamically power on and off the cache lines; this lead to further reduce the leakage. Evaluation results with SPLASH-2 applications show that the area and leakage of the LLC can be almost halved without hurting the system performance; further reductions in area and leakage are possible at the expense of reduced performance.

Chapter 7

Conclusions

In this thesis we present several techniques at the NoC- and cache-level to improve the performance of a tiled CMP system. The cache hierarchy and the NoC are two tightly coupled components of a CMP: most of NoC traffic is due to coherence messages, so the NoC must be tailored to efficiently manage the traffic pattern generated by the caches. Different classes of coherence messages have different requirements in terms of latency and bandwidth; acknowledgement messages, in particular, have very low bandwidth requirements but should be transmitted with a minimum latency to speed-up coherence actions. Furthermore, many-to-one acknowledgements should be gathered to deliver a single global acknowledgement at the destination node, rather than multiple acknowledgements which probably serialize at the destination's input buffer. Thus, a dedicated network with low bandwidth and latency capable of gathering many-to-one messages can improve system performance, especially if the system includes a high number of cores. Relieving the NoC from the percentage of traffic due to the acknowledgements, with a dedicated network capable of efficiently handling this traffic, enables the implementation of broadcast-based techniques in many-core systems.

In systems which employ a NUCA LLC, the NoC is also used to access the LLC in case of L1 cache miss. The NoC thus has a direct impact on the LLC access latency, and the mapping of cache blocks to the LLC banks determines the number of hops between the tile where the requestor L1 cache is located and the LLC bank where the requested block is mapped to. An ideal mapping policy would map cache blocks as close as possible to the requestor, possibly in the same tile, to reduce the hops number,

exploiting at the same time the whole on-chip cache capacity to avoid the drawbacks of private LLCs. This can be achieved by dynamically mapping cache blocks to the LLC banks at runtime. The on-chip component which is the fittest to perform this mapping is the memory controller: all traffic between on-chip caches and the main memory is indeed managed by the MC, which can thus collect statistics to perform the optimal block mapping.

Extending the role of the MC to give him the task of managing on-chip cache resources is also important when the chip is virtualized. In a virtualized CMP, resources are partitioned and each partition is allocated to an application, which has an exclusive view of its pool of resources. To efficiently exploit the CMP, it should be possible to independently partition cores, the NoC and the LLC layer.

The contributions of this thesis address the issues listed above. First, a lightweight dedicated network has been added to the regular NoC; this network, called the Gather Network (GN), can be used to transmit simple control messages in a few clock cycles. In Chapter 3, we use the GN to transmit invalidation ACKs in a directory-based protocol and coherence ACKs in a broadcast-based protocol. Thanks to the GN, the traffic overhead of a broadcast-based protocol can be highly reduced, thus allowing broadcast-based protocols to reach performance comparable to those of directory-based protocols, eliminating the area overhead due to the directory. This work appears in the following publications:

- Mario Lodde, José Flich, and Manuel E. Acacio. Heterogeneous NoC Design for Efficient Broadcast-based Coherence Protocol Support. In *Proceedings of the 2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip (NOCS '12)*. IEEE Computer Society, Washington, DC, USA, 59-66. 2012.

- Mario Lodde, Toni Roca, and José Flich. Built-in fast gather control network for efficient support of coherence protocols, *Computers & Digital Techniques*, IET, vol.7, no.2. 2013.

- Mario Lodde, Toni Roca, and José Flich. Heterogeneous network design for effective support of invalidation-based coherency protocols. In *Proceedings of the 2012 Interconnection Network Architecture: On-Chip, Multi-Chip Workshop (INA-OCMC '12)*. ACM, New York, NY, USA, 1-4. 2012.

- Mario Lodde and José Flich. A lightweight network of ids to quickly deliver simple control messages. In *Proc. of the 2nd Intl, Workshop on On-chip memory hierarchies and interconnects: organization, management and implementation (OMHI '13)*. 2013.

In Chapter 4 Runtime Home Mapping (RHM), a dynamic mapping policy of cache blocks to LLC banks, is proposed; the GN is used to reduce the additional traffic due to the home bank search phase, thus providing an NoC substrate to support and speed-up the novel mapping policy. In RHM the block mapping is performed by the memory controller, which maps cache blocks as close as possible to the requestor, balancing at the same time the load of LLC banks; block migration and replication are used to correct the initial block placement and further reduce the LLC access latency. Evaluation results show that RHM achieves lower LLC access latencies than other dynamic techniques such as D-NUCA or First-Touch, and it achieves the same locality of private LLCs with a better utilization of the on-chip cache capacity; this reflects in a reduced execution time. Performance can further be improved when RHM is combined with a broadcast-based coherence protocol, thanks to the broadcast nature of both techniques. This work appears in the following publication:

- Mario Lodde, José Flich, and Manuel E. Acacio. Towards Efficient Dynamic LLC Home Bank Mapping with NoC-Level Support. In *Proceedings of the 19th international conference on Parallel Processing (Euro-Par'12)*. Springer-Verlag, Berlin, Heidelberg, 178-190. 2013.

An extended version is under review in:

- Mario Lodde and José Flich. Runtime Home Mapping for Effective Memory Resource Usage. Submitted to *Microprocessors and Microsystems*.

In Chapter 5, we integrate RHM with Logic-Based Distributed Routing (LBDR) to obtain pNC, a hardware substrate which allows the independent and efficient partitioning of cores, NoC resources and LLC banks in a virtualized CMP system. Existent techniques only allow the partitioning of CMP resources at tile-level, while a finer grain is actually needed to better exploit the CMP. pNC allows to decouple the cores and the

LLC partitioning to better fit the requirements of an application. This work appears in the following publication:

- Mario Lodde and José Flich. An NoC and cache hierarchy substrate to address effective virtualization and fault-tolerance. In *Proceedings of the 2013 IEEE/ACM Seventh International Symposium on Networks-on-Chip (NOCS '12)*. IEEE Computer Society, Washington, DC, USA, 21-24. 2012.

Finally, in Chapter 6 we propose a reorganization of LLC banks to reduce LLC area and leakage. The proposal is based on keeping the same associativity of the tag array while reducing the associativity of the data array, thus creating two types of entries: tag-only entries, which are used to store private blocks, and entries including both tag and data, which are used to store shared blocks and blocks which are only present in the LLC. Evaluation results show that with our technique it is possible to halve the LLC area and leakage with a very low performance degradation. This work appears in the following publication:

- Mario Lodde, José Flich, and Manuel E. Acacio. Dynamic last-level cache allocation to reduce area and power overhead in directory coherence protocols. In *Proceedings of the 18th international conference on Parallel Processing (Euro-Par'12)*. Springer-Verlag, Berlin, Heidelberg, 206-218. 2012.

Currently these techniques are being integrated and implemented in an FPGA board; more details are provided in Appendix B.

As a future work, we see the following main directions:

- the GN can be used to transmit other types of simple messages; a promising direction is to extend the GN to create an hardware infrastructure to thread synchronization primitives such a barriers or locks.
- RHM, as presented in this thesis, assume a system with one memory controller; it should be adapted to solve the issues and take advantage of the opportunities offered by a system with more than one memory controller.

- different, more sophisticated mapping algorithms can be implemented at the MC in RHM.
- the role of the MC in pNC can be extended to dynamically resize the Home Partitions at runtime depending on application's demand.

Appendix A

Coherence Protocols

This appendix contains some of the protocols implemented in gMemNoCsim to evaluate the proposals of this thesis. We chose to include in this appendix the basic Directory and Hammer protocols with static mapping of cache blocks to the LLC banks, a Directory protocol with blocks mapped using RHM (including block migration and replication) and the Hammer protocol with blocks mapped using RHM, where the coherence broadcast are tightly coupled with the broadcasts issued by RHM during the home search phase. The other protocols used in this thesis are extensions or reduced versions of these four protocols.

Protocols are described using the syntax required by gMemNoCsim. First, a device is declared (L1/L2), followed by the list of states of its FSM. Then, for each state all the possible transitions are listed. Each line describing a transition includes the event name, a list of coherence actions and the final state.

A.1 Directory (MESI)

This section describes the basic Directory protocol; MESI states are assumed at the L1 caches, and blocks are statically mapped to the L2 banks. This protocol is used as it is in Chapter 3, Chapter 4, Chapter 5 and Chapter 6. The two modified protocols described in Section 3.4.2 and the MOESI protocol used in Chapter 6 can be easily derived from this protocol. D-NUCA and PRIVATE L2 protocols used in Chapter 4 have also been obtained extending this protocol: for D-NUCA, it is necessary to add the

home bank search within a bank set and the block migration mechanism; for PRIVATE L2, the directory must be moved to the MC. This protocol can also be used to simulate First-Touch, just changing the home mapping policy.

```
DEVICE L1
LIST_STATES I,IS,IM,S,E,MS,M,SI,EI,MI,ISI,IMI

STATE I
EVENT CANT_ALLOCATE,{ENQUEUE_REQUEST_REPL,TRIGGER_REPLACEMENT},I
EVENT LOAD,{ALLOCATE_MSHR,ALLOCATE_L1,SEND_GETS_TO_L2},IS
EVENT FETCH,{ALLOCATE_MSHR,ALLOCATE_L1,SEND_GETS_TO_L2},IS
EVENT STORE,{ALLOCATE_MSHR,ALLOCATE_L1,SEND_GETX_TO_L2},IM

STATE IS
EVENT RECEIVED_DATA_SHARED_AT_REQUESTOR,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,UNBLOCK_PROCESSOR,DEQUEUE_REQUEST},S
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,UNBLOCK_PROCESSOR,DEQUEUE_REQUEST},E
EVENT RECEIVED_FWD_GETS,{ENQUEUE_REQUEST},IS
EVENT_RECEIVED_FWD_GETX,{ENQUEUE_REQUEST},IS
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},ISI
EVENT RECEIVED_INV_TO_OWNER,{NONE},IMI

STATE IM
EVENT RECEIVED_ACK,{DECREMENT_ACK_COUNTER},IM
EVENT RECEIVED_LAST_ACK_AND_DATA,{DEALLOCATE_MSHR,UNBLOCK_PROCESSOR,DEQUEUE_REQUEST},M
EVENT RECEIVED_DATA_EXCLUSIVE_AT_REQUESTOR,{COPY_DATA_TO_CACHE},IM
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,UNBLOCK_PROCESSOR,DEQUEUE_REQUEST},M
EVENT RECEIVED_FWD_GETS,{ENQUEUE_REQUEST},IM
EVENT_RECEIVED_FWD_GETX,{ENQUEUE_REQUEST},IM
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},IM
EVENT RECEIVED_INV_TO_OWNER,{NONE},IMI
EVENT RECEIVED_WBACK_TO_L1,{NONE},IM

STATE S
EVENT WRONG_BANK,{MOVE_CACHE_BLOCK,REPEAT_REQUEST},S
EVENT CANT_ALLOCATE,{ENQUEUE_REQUEST_REPL,TRIGGER_REPLACEMENT},S
EVENT LOAD,{UNBLOCK_PROCESSOR},S
EVENT FETCH,{UNBLOCK_PROCESSOR},S
EVENT STORE,{ALLOCATE_MSHR,SEND_GETX_TO_L2},IM
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR,DEALLOCATE_L1},I
EVENT RECEIVED_INV_REPL,{DEALLOCATE_L1},I
EVENT L1_REPLACEMENT,{ALLOCATE_MSHR,SEND_PUTS_TO_L2,DEQUEUE_REQUEST_REPL_MSHR,DEALLOCATE_L1},SI

STATE E
EVENT WRONG_BANK,{MOVE_CACHE_BLOCK,REPEAT_REQUEST},E
EVENT CANT_ALLOCATE,{ENQUEUE_REQUEST_REPL,TRIGGER_REPLACEMENT},E
EVENT LOAD,{UNBLOCK_PROCESSOR},E
EVENT FETCH,{UNBLOCK_PROCESSOR},E
EVENT STORE,{UNBLOCK_PROCESSOR},M
EVENT RECEIVED_FWD_GETS,{SEND_DATA_SHARED_FROM_OWNER_TO_REQUESTOR,SEND_ACCEPTS_TO_L2},MS
EVENT RECEIVED_FWD_GETX,{SEND_DATA_EXCLUSIVE_FROM_OWNER_TO_REQUESTOR,DEALLOCATE_L1},I
EVENT L1_REPLACEMENT,{ALLOCATE_MSHR,SEND_PUTX_TO_L2,DEQUEUE_REQUEST_REPL_MSHR,DEALLOCATE_L1},EI
EVENT RECEIVED_INV_TO_OWNER,{ALLOCATE_MSHR,SEND_PUTX_TO_L2,DEALLOCATE_L1},EI

STATE MS
EVENT WRONG_BANK,{MOVE_CACHE_BLOCK,REPEAT_REQUEST},MS
EVENT CANT_ALLOCATE,{ENQUEUE_REQUEST_REPL,TRIGGER_REPLACEMENT},MS
EVENT LOAD,{UNBLOCK_PROCESSOR},MS
EVENT FETCH,{UNBLOCK_PROCESSOR},MS
EVENT STORE,{RECYCLE_REQUEST},MS
EVENT RECEIVED_FWD_GETS,{SEND_DATA_SHARED_FROM_OWNER_TO_REQUESTOR},MS
EVENT RECEIVED_FWD_GETX,{SEND_DATA_EXCLUSIVE_FROM_OWNER_TO_REQUESTOR,DEALLOCATE_L1},I
EVENT RECEIVED_INV_REPL,{DEALLOCATE_L1},I
EVENT RECEIVED_WBACK_TO_L1,{NONE},S
EVENT L1_REPLACEMENT,{RECYCLE_REQUEST},MS

STATE M
EVENT WRONG_BANK,{MOVE_CACHE_BLOCK,REPEAT_REQUEST},M
EVENT CANT_ALLOCATE,{ENQUEUE_REQUEST_REPL,TRIGGER_REPLACEMENT},M
EVENT LOAD,{UNBLOCK_PROCESSOR},M
EVENT STORE,{UNBLOCK_PROCESSOR},M
EVENT FETCH,{UNBLOCK_PROCESSOR},M
EVENT RECEIVED_FWD_GETS,{SEND_DATA_SHARED_FROM_OWNER_TO_REQUESTOR,SEND_ACCEPTS_TO_L2},MS
EVENT RECEIVED_FWD_GETX,{SEND_DATA_EXCLUSIVE_FROM_OWNER_TO_REQUESTOR,DEALLOCATE_L1},I
EVENT L1_REPLACEMENT,{ALLOCATE_MSHR,SEND_PUTX_TO_L2,DEQUEUE_REQUEST_REPL_MSHR,DEALLOCATE_L1},MI
EVENT RECEIVED_INV_TO_OWNER,{ALLOCATE_MSHR,SEND_PUTX_TO_L2,DEALLOCATE_L1},MI
```

```
STATE SI
EVENT RECEIVED_WBACK_TO_L1,{DEALLOCATE_MSHR},I
EVENT L1_REPLACEMENT,{RECYCLE_REQUEST},SI
EVENT RECEIVED_INV_TO_OWNER,{NONE},SI
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},SI
EVENT LOAD,{RECYCLE_REQUEST},SI
EVENT FETCH,{RECYCLE_REQUEST},SI
EVENT STORE,{RECYCLE_REQUEST},SI

STATE MI
EVENT RECEIVED_WBACK_TO_L1,{DEALLOCATE_MSHR},I
EVENT L1_REPLACEMENT,{RECYCLE_QUEUED_REQUEST},MI
EVENT RECEIVED_INV_TO_OWNER,{NONE},MI
EVENT RECEIVED_FWD_GETS,{SEND_DATA_SHARED_FROM_OWNER_TO_REQUESTOR},MI
EVENT RECEIVED_FWD_GETX,{SEND_DATA_EXCLUSIVE_FROM_OWNER_TO_REQUESTOR},MI
EVENT LOAD,{RECYCLE_REQUEST},MI
EVENT FETCH,{RECYCLE_REQUEST},MI
EVENT STORE,{RECYCLE_REQUEST},MI

STATE EI
EVENT RECEIVED_WBACK_TO_L1,{DEALLOCATE_MSHR},I
EVENT L1_REPLACEMENT,{RECYCLE_REQUEST},EI
EVENT RECEIVED_INV_TO_OWNER,{NONE},EI
EVENT RECEIVED_FWD_GETS,{SEND_DATA_SHARED_FROM_OWNER_TO_REQUESTOR},EI
EVENT RECEIVED_FWD_GETX,{SEND_DATA_EXCLUSIVE_FROM_OWNER_TO_REQUESTOR},EI
EVENT LOAD,{RECYCLE_REQUEST},EI
EVENT FETCH,{RECYCLE_REQUEST},EI
EVENT STORE,{RECYCLE_REQUEST},EI

STATE ISI
EVENT RECEIVED_DATA_SHARED_AT_REQUESTOR,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,UNBLOCK_PROCESSOR,DEALLOCATE_L1},I
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,UNBLOCK_PROCESSOR,DEALLOCATE_L1},I

STATE IMI
EVENT RECEIVED_ACK,{DECREMENT_ACK_COUNTER},IMI
EVENT RECEIVED_LAST_ACK_AND_DATA,{UNBLOCK_PROCESSOR,SEND_PUTX_TO_L2,DEQUEUE_REQUEST,DEALLOCATE_L1},MI
EVENT RECEIVED_DATA_EXCLUSIVE_AT_REQUESTOR,{COPY_DATA_TO_CACHE},IMI
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS,{UNBLOCK_PROCESSOR,SEND_PUTX_TO_L2,DEQUEUE_REQUEST,DEALLOCATE_L1},MI

DEVICE L2
LIST_STATES I,IP,P,PS,S,C,PI

STATE I
EVENT CANT_ALLOCATE,{ENQUEUE_REQUEST_REPL,TRIGGER_REPLACEMENT},I
EVENT CANT_ALLOCATE_AND_CANT_REPLACE,{RECYCLE_REQUEST_TO_L2},I
EVENT RECEIVED_GETS,{ALLOCATE_MSHR,ALLOCATE_L2,SAVE_OWNER,SEND_GET_TO_MC},IP
EVENT RECEIVED_GETX,{ALLOCATE_MSHR,ALLOCATE_L2,SAVE_OWNER,SEND_GET_TO_MC},IP

STATE IP
EVENT RECEIVED_GETS,{ENQUEUE_REQUEST},IP
EVENT RECEIVED_GETX,{ENQUEUE_REQUEST},IP
EVENT L2_REPLACEMENT,{RECYCLE_REQUEST},IP
EVENT RECEIVED_DATA,{DEALLOCATE_MSHR,SEND_DATA_EXCLUSIVE_TO_REQUESTOR,DEQUEUE_REQUEST},P

STATE P
EVENT RECEIVED_GETX,{FWD_REQUEST_TO_OWNER,SAVE_OWNER},P
EVENT RECEIVED_GETS,{FWD_REQUEST_TO_OWNER,ADD_SHARER},PS
EVENT RECEIVED_ACCEPTS_FROM_OLD_OWNER,{NONE},P
EVENT RECEIVED_PUTX_TO_L2,{SEND_WBACK_TO_REQUESTOR},C
EVENT RECEIVED_PUTS_FROM_OWNER,{SEND_WBACK_TO_REQUESTOR},C
EVENT RECEIVED_PUTS_TO_L2,{SEND_WBACK_TO_REQUESTOR},P
EVENT RECEIVED_PUTX_FROM_OLD_OWNER,{SEND_WBACK_TO_REQUESTOR},P
EVENT L2_REPLACEMENT,{ALLOCATE_MSHR,SEND_INV_TO_OWNER,DEQUEUE_REQUEST_REPL_MSHR,DEALLOCATE_L2},PI

STATE S
EVENT RECEIVED_GETS,{ADD_SHARER,SEND_DATA_SHARED_TO_REQUESTOR},S
EVENT RECEIVED_GETX,{REMOVE_REQUESTOR_FROM_SHARERS,SAVE_OWNER,SEND_DATA_EXCLUSIVE_TO_REQUESTOR,SEND_INV_TO_SHARERS,CLEAR_SHARERS},P
EVENT RECEIVED_PUTS_TO_L2,{REMOVE_SHARER_FROM_DIRECTORY,SEND_WBACK_TO_REQUESTOR},S
EVENT RECEIVED_LAST_PUTS_TO_L2,{REMOVE_SHARER_FROM_DIRECTORY,SEND_WBACK_TO_REQUESTOR},C
EVENT L2_REPLACEMENT,{INVALIDATE_SHARERS_REPL,CONDITIONAL_SEND_DATA_TO_MC,DEQUEUE_REQUEST_REPL,DEALLOCATE_L2},I

STATE C
EVENT RECEIVED_GETX,{SAVE_OWNER,SEND_DATA_EXCLUSIVE_TO_REQUESTOR},P
EVENT RECEIVED_GETS,{SAVE_OWNER,SEND_DATA_EXCLUSIVE_TO_REQUESTOR},P
EVENT L2_REPLACEMENT,{CONDITIONAL_SEND_DATA_TO_MC,DEQUEUE_REQUEST_REPL,DEALLOCATE_L2},I

STATE PS
EVENT RECEIVED_ACCEPTS_TO_L2,{SEND_WBACK_TO_ACCEPTS_SENDER,ADD_OWNER_TO_SHARERS,DEQUEUE_REQUEST},S
EVENT RECEIVED_ACCEPTS_FROM_OLD_OWNER,{NONE},PS
EVENT L2_REPLACEMENT,{RECYCLE_REQUEST},PS
```

```
EVENT RECEIVED_PUTX_TO_L2,{SEND_WBACK_TO_REQUESTOR,DEQUEUE_REQUEST},S
EVENT RECEIVED_PUTX_FROM_OLD_OWNER,{ENQUEUE_REQUEST},PS
EVENT RECEIVED_GETX,{ENQUEUE_REQUEST},PS
EVENT RECEIVED_GETS,{ENQUEUE_REQUEST},PS
EVENT RECEIVED_PUTS_TO_L2,{REMOVE_SHARER_FROM_DIRECTORY,SEND_WBACK_TO_REQUESTOR},PS

STATE PI
EVENT RECEIVED_PUTS_TO_L2,{SEND_WBACK_TO_REQUESTOR,DEALLOCATE_MSHR},I
EVENT RECEIVED_PUTX_TO_L2,{SEND_WBACK_TO_REQUESTOR,SEND_DATA_TO_MC,DEALLOCATE_MSHR},I
EVENT RECEIVED_GET_MSHR,{RECYCLE_REQUEST_TO_L2},PI
EVENT RECEIVED_GETS,{RECYCLE_REQUEST_TO_L2},PI
EVENT RECEIVED_GETX,{RECYCLE_REQUEST_TO_L2},PI
EVENT L2_REPLACEMENT,{REPEAT_REQUEST_REPL},PI
```

A.2 Hammer

This section describes the basic Hammer protocol; MESI states are assumed at L1 caches, and blocks are statically mapped to the L2 banks. This protocol is used in Chapter 3.

```
DEVICE L1
LIST_STATES I,IS,IM,S,E,M,SI,EI,MI,IMI

STATE I
EVENT CANT_ALLOCATE,{ENQUEUE_REQUEST_REPL,TRIGGER_REPLACEMENT},I
EVENT LOAD,{ALLOCATE_MSHR,ALLOCATE_L1,SEND_GETS_TO_L2},IS
EVENT FETCH,{ALLOCATE_MSHR,ALLOCATE_L1,SEND_GETS_TO_L2},IS
EVENT STORE,{ALLOCATE_MSHR,ALLOCATE_L1,SEND_GETX_TO_L2},IM
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},I
EVENT RECEIVED_INV_REPL,{NONE},I
EVENT RECEIVED_BCAST_REQ,{SEND_ACK_TO_REQUESTOR},I

STATE IS
EVENT RECEIVED_ACK,{DECREMENT_ACK_COUNTER},IS
EVENT RECEIVED_LAST_ACK_AND_DATA,{DEALLOCATE_MSHR,SEND_ACK_TO_HOME,UNBLOCK_PROCESSOR,DEQUEUE_REQUEST},S
EVENT RECEIVED_DATA_SHARED_AT_REQUESTOR_WAIT_ACKS,{COPY_DATA_TO_CACHE},IS
EVENT RECEIVED_DATA_SHARED_AND_ALL_ACKS,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,SEND_ACK_TO_HOME,UNBLOCK_PROCESSOR,DEQUEUE_REQUEST},S
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,SEND_ACK_TO_HOME,UNBLOCK_PROCESSOR,DEQUEUE_REQUEST},E
EVENT RECEIVED_FWD_GETS,{ENQUEUE_REQUEST},IS
EVENT_RECEIVED_FWD_GETX,{ENQUEUE_REQUEST},IS
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},IS
EVENT RECEIVED_INV_TO_OWNER,{NONE},IMI
EVENT RECEIVED_BCAST_REQ,{SEND_ACK_TO_REQUESTOR},IS

STATE IM
EVENT RECEIVED_ACK,{DECREMENT_ACK_COUNTER},IM
EVENT RECEIVED_LAST_ACK_AND_DATA,{DEALLOCATE_MSHR,SEND_ACK_TO_HOME,UNBLOCK_PROCESSOR,DEQUEUE_REQUEST},M
EVENT RECEIVED_DATA_EXCLUSIVE_AT_REQUESTOR,{COPY_DATA_TO_CACHE},IM
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,SEND_ACK_TO_HOME,UNBLOCK_PROCESSOR,DEQUEUE_REQUEST},M
EVENT RECEIVED_FWD_GETS,{ENQUEUE_REQUEST},IM
EVENT_RECEIVED_FWD_GETX,{ENQUEUE_REQUEST},IM
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},IM
EVENT RECEIVED_INV_TO_OWNER,{NONE},IMI
EVENT RECEIVED_WBACK_TO_L1,{NONE},IM
EVENT RECEIVED_BCAST_REQ,{SEND_ACK_TO_REQUESTOR},IM

STATE S
EVENT WRONG_BANK,{MOVE_CACHE_BLOCK,REPEAT_REQUEST},S
EVENT CANT_ALLOCATE,{ENQUEUE_REQUEST_REPL,TRIGGER_REPLACEMENT},S
EVENT LOAD,{UNBLOCK_PROCESSOR},S
EVENT FETCH,{UNBLOCK_PROCESSOR},S
EVENT STORE,{ALLOCATE_MSHR,SEND_GETX_TO_L2},IM
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR,DEALLOCATE_L1},I
EVENT RECEIVED_INV_REPL,{DEALLOCATE_L1},I
EVENT L1_REPLACEMENT,{DEQUEUE_REQUEST_REPL,DEALLOCATE_L1},I
EVENT RECEIVED_BCAST_REQ,{SEND_ACK_TO_REQUESTOR,DEALLOCATE_L1},I

STATE E
EVENT WRONG_BANK,{MOVE_CACHE_BLOCK,REPEAT_REQUEST},E
EVENT CANT_ALLOCATE,{ENQUEUE_REQUEST_REPL,TRIGGER_REPLACEMENT},E
```

```
EVENT LOAD,{UNBLOCK_PROCESSOR},E
EVENT FETCH,{UNBLOCK_PROCESSOR},E
EVENT STORE,{UNBLOCK_PROCESSOR},M
EVENT RECEIVED_FWD_GETS,{SEND_DATA_SHARED_FROM_OWNER_TO_REQUESTOR},S
EVENT RECEIVED_FWD_GETX,{SEND_DATA_EXCLUSIVE_FROM_OWNER_TO_REQUESTOR,DEALLOCATE_L1},I
EVENT L1_REPLACEMENT,{ALLOCATE_MSHR,SEND_PUTX_TO_L2,DEQUEUE_REQUEST_REPL_MSHR,DEALLOCATE_L1},EI
EVENT RECEIVED_BCAST_REQ,{SEND_DATA_EXCLUSIVE_FROM_OWNER_TO_REQUESTOR,DEALLOCATE_L1},I
EVENT RECEIVED_INV_REPL,{ALLOCATE_MSHR,SEND_PUTX_TO_L2,DEALLOCATE_L1},EI

STATE M
EVENT WRONG_BANK,{MOVE_CACHE_BLOCK,REPEAT_REQUEST},M
EVENT CANT_ALLOCATE,{ENQUEUE_REQUEST_REPL,TRIGGER_REPLACEMENT},M
EVENT LOAD,{UNBLOCK_PROCESSOR},M
EVENT STORE,{UNBLOCK_PROCESSOR},M
EVENT FETCH,{UNBLOCK_PROCESSOR},M
EVENT RECEIVED_FWD_GETS,{SEND_DATA_SHARED_FROM_OWNER_TO_REQUESTOR},O
EVENT RECEIVED_FWD_GETX,{SEND_DATA_EXCLUSIVE_FROM_OWNER_TO_REQUESTOR,DEALLOCATE_L1},I
EVENT L1_REPLACEMENT,{ALLOCATE_MSHR,SEND_PUTX_TO_L2,DEQUEUE_REQUEST_REPL_MSHR,DEALLOCATE_L1},MI
EVENT RECEIVED_BCAST_REQ,{SEND_DATA_EXCLUSIVE_FROM_OWNER_TO_REQUESTOR,DEALLOCATE_L1},I
EVENT RECEIVED_INV_REPL,{ALLOCATE_MSHR,SEND_PUTX_TO_L2,DEALLOCATE_L1},MI

STATE SI
EVENT RECEIVED_WBACK_TO_L1,{DEALLOCATE_MSHR},I
EVENT L1_REPLACEMENT,{RECYCLE_REQUEST},SI
EVENT RECEIVED_INV_TO_OWNER,{NONE},SI
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},SI
EVENT LOAD,{RECYCLE_REQUEST},SI
EVENT FETCH,{RECYCLE_REQUEST},SI
EVENT STORE,{RECYCLE_REQUEST},SI

STATE MI
EVENT RECEIVED_WBACK_TO_L1,{DEALLOCATE_MSHR},I
EVENT L1_REPLACEMENT,{RECYCLE_REQUEST},MI
EVENT RECEIVED_FWD_GETS,{SEND_DATA_SHARED_FROM_OWNER_TO_REQUESTOR},MI
EVENT RECEIVED_FWD_GETX,{SEND_DATA_EXCLUSIVE_FROM_OWNER_TO_REQUESTOR},MI
EVENT LOAD,{RECYCLE_REQUEST},MI
EVENT FETCH,{RECYCLE_REQUEST},MI
EVENT STORE,{RECYCLE_REQUEST},MI

STATE EI
EVENT RECEIVED_WBACK_TO_L1,{DEALLOCATE_MSHR},I
EVENT L1_REPLACEMENT,{RECYCLE_REQUEST},EI
EVENT RECEIVED_FWD_GETS,{SEND_DATA_SHARED_FROM_OWNER_TO_REQUESTOR},EI
EVENT RECEIVED_FWD_GETX,{SEND_DATA_EXCLUSIVE_FROM_OWNER_TO_REQUESTOR},EI
EVENT LOAD,{RECYCLE_REQUEST},EI
EVENT FETCH,{RECYCLE_REQUEST},EI
EVENT STORE,{RECYCLE_REQUEST},EI

STATE IMI
EVENT RECEIVED_ACK,{DECREMENT_ACK_COUNTER},IMI
EVENT RECEIVED_LAST_ACK_AND_DATA,{UNBLOCK_PROCESSOR,SEND_PUTX_TO_L2,DEQUEUE_REQUEST,DEALLOCATE_L1},MI
EVENT RECEIVED_DATA_EXCLUSIVE_AT_REQUESTOR,{COPY_DATA_TO_CACHE},IMI
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS,{UNBLOCK_PROCESSOR,SEND_PUTX_TO_L2,DEQUEUE_REQUEST,DEALLOCATE_L1},MI

DEVICE L2
LIST_STATES I,IP,P,S,C,PI,BP,BS

STATE I
EVENT CANT_ALLOCATE,{ENQUEUE_REQUEST_REPL,TRIGGER_REPLACEMENT},I
EVENT CANT_ALLOCATE_AND_CANT_REPLACE,{RECYCLE_REQUEST_TO_L2},I
EVENT RECEIVED_GETS,{ALLOCATE_MSHR,ALLOCATE_L2,SAVE_OWNER,SEND_GET_TO_MC},IP
EVENT RECEIVED_GETX,{ALLOCATE_MSHR,ALLOCATE_L2,SAVE_OWNER,SEND_GET_TO_MC},IP

STATE IP
EVENT RECEIVED_GETS,{ENQUEUE_REQUEST},IP
EVENT RECEIVED_GETX,{ENQUEUE_REQUEST},IP
EVENT L2_REPLACEMENT,{RECYCLE_REQUEST},IP
EVENT RECEIVED_DATA,{DEALLOCATE_MSHR,SEND_DATA_EXCLUSIVE_TO_REQUESTOR,DEQUEUE_REQUEST},BP

STATE P
EVENT RECEIVED_GETX,{BROADCAST_REQUEST},BP
EVENT RECEIVED_GETS,{BROADCAST_REQUEST},BS
EVENT RECEIVED_PUTX_TO_L2_H,{SEND_WBACK_TO_REQUESTOR},C
EVENT L2_REPLACEMENT,{ALLOCATE_MSHR,BROADCAST_INV_REPL,DEQUEUE_REQUEST_REPL_MSHR,DEALLOCATE_L2},PI

STATE S
EVENT RECEIVED_GETS,{SEND_DATA_SHARED_TO_REQUESTOR},BS
EVENT RECEIVED_GETX,{SEND_DATA_EXCLUSIVE_TO_REQUESTOR,BROADCAST_INV},BP
EVENT L2_REPLACEMENT,{BROADCAST_INV_REPL,CONDITIONAL_SEND_DATA_TO_MC,DEQUEUE_REQUEST_REPL,DEALLOCATE_L2},I
```

```
STATE C
EVENT RECEIVED_GETX,{SEND_DATA_EXCLUSIVE_TO_REQUESTOR},BP
EVENT RECEIVED_GETS,{SEND_DATA_EXCLUSIVE_TO_REQUESTOR},BP
EVENT L2_REPLACEMENT,{CONDITIONAL_SEND_DATA_TO_MC,DEQUEUE_REQUEST_REPL,DEALLOCATE_L2},I

STATE PI
EVENT RECEIVED_PUTX_TO_L2_H,{SEND_DATA_TO_MC,SEND_WBACK_TO_REQUESTOR,DEALLOCATE_MSHR},I
EVENT RECEIVED_GET_MSHR,{RECYCLE_REQUEST_TO_L2},PI
EVENT RECEIVED_GETS,{RECYCLE_REQUEST_TO_L2},PI
EVENT RECEIVED_GETX,{RECYCLE_REQUEST_TO_L2},PI
EVENT L2_REPLACEMENT,{REPEAT_REQUEST_REPL},PI

STATE BP
EVENT RECEIVED_ACK_TO_L2,{DEQUEUE_REQUEST},P
EVENT L2_REPLACEMENT,{RECYCLE_REQUEST},BP
EVENT RECEIVED_GETS,{ENQUEUE_REQUEST},BP
EVENT RECEIVED_GETX,{ENQUEUE_REQUEST},BP
EVENT RECEIVED_PUTX_TO_L2_H,{SEND_WBACK_TO_REQUESTOR},BP

STATE BS
EVENT RECEIVED_ACK_TO_L2,{DEQUEUE_REQUEST},S
EVENT L2_REPLACEMENT,{RECYCLE_REQUEST},BS
EVENT RECEIVED_GETS,{ENQUEUE_REQUEST},BS
EVENT RECEIVED_GETX,{ENQUEUE_REQUEST},BS
EVENT RECEIVED_PUTX_TO_L2_H,{SEND_WBACK_TO_REQUESTOR},BS
```

A.3 Directory + RHM with Block Migration and Replication

This section describes the Directory protocol with RHM and block migration and replication. MESI states are assumed at L1 caches. This protocol has been used as it is in Chapter 4. The other versions used in Chapter 4 and Chapter 5, one having the block replication mechanism disabled and another one with both migration and replication disabled, are subsets of this protocol.

```
DEVICE L1
LIST_STATES I,IS,IM,S,E,M,EI,MI,ISI,IMI,IMM,ISWA,IMWA,SWA,MWA,EWA,SWAM,EWAM,MWAM,IMMWA,MWAA,SWAI

STATE I
EVENT CANT_ALLOCATE,{ENQUEUE_REQUEST_REPL,TRIGGER_REPLACEMENT},I
EVENT LOAD,{ALLOCATE_MSHR,ALLOCATE_L1,SEND_GETS_TO_L2_DHM},ISWA
EVENT FETCH,{ALLOCATE_MSHR,ALLOCATE_L1,SEND_GETS_TO_L2_DHM},ISWA
EVENT STORE,{ALLOCATE_MSHR,ALLOCATE_L1,SEND_GETX_TO_L2_DHM},IMWA
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},I
EVENT RECEIVED_INV_REPL,{NONE},I

STATE ISWA
EVENT RECEIVED_DATA_SHARED_AT_REQUESTOR_REPL,{COPY_DATA_TO_CACHE,SEND_DATA_TO_L2_BANK},SWAM
EVENT RECEIVED_DATA_SHARED_AT_REQUESTOR_MIGR,{COPY_DATA_TO_CACHE,SEND_DATA_TO_L2_BANK},SWAM
EVENT RECEIVED_DATA_SHARED_AT_REQUESTOR,{COPY_DATA_TO_CACHE},SWA
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS_MIGR,{COPY_DATA_TO_CACHE,SEND_DATA_TO_L2_BANK},EWAM
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS,{COPY_DATA_TO_CACHE},EWA
EVENT RECEIVED_FWD_GETS,{ENQUEUE_REQUEST},ISWA
EVENT_RECEIVED_FWD_GETX,{ENQUEUE_REQUEST},ISWA
EVENT RECEIVED_DHM_ACK,{SET_DHM_ACK_FLAG},IS
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},ISWA
EVENT RECEIVED_INV_REPL,{NONE},ISWA

STATE IMWA
EVENT RECEIVED_DHM_ACK,{SET_DHM_ACK_FLAG},IM
EVENT RECEIVED_ACK,{DECREMENT_ACK_COUNTER},IMWA
EVENT RECEIVED_LAST_ACK_AND_DATA,{NONE},MWA
```

```
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS_MIGR,{COPY_DATA_TO_CACHE,SEND_DATA_TO_L2_BANK},MWAM
EVENT RECEIVED_DATA_EXCLUSIVE_AT_REQUESTOR_MIGR,{COPY_DATA_TO_CACHE,SEND_DATA_TO_L2_BANK},IMMWA
EVENT RECEIVED_DATA_EXCLUSIVE_AT_REQUESTOR,{COPY_DATA_TO_CACHE},MWAA
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS,{COPY_DATA_TO_CACHE},MWA
EVENT RECEIVED_FWD_GETS,{ENQUEUE_REQUEST},IMWA
EVENT_RECEIVED_FWD_GETX,{ENQUEUE_REQUEST},IMWA
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},IMWA
EVENT RECEIVED_INV_REPL,{NONE},IMWA

STATE SWA
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},SWAI
EVENT RECEIVED_DHM_ACK,{DEALLOCATE_MSHR,UNBLOCK_PROCESSOR,DEQUEUE_REQUEST},S

STATE EWA
EVENT RECEIVED_DHM_ACK,{DEALLOCATE_MSHR,UNBLOCK_PROCESSOR,DEQUEUE_REQUEST},E
EVENT RECEIVED_FWD_GETS,{ENQUEUE_REQUEST},EWA
EVENT_RECEIVED_FWD_GETX,{ENQUEUE_REQUEST},EWA

STATE MWA
EVENT RECEIVED_DHM_ACK,{DEALLOCATE_MSHR,UNBLOCK_PROCESSOR,DEQUEUE_REQUEST},M
EVENT RECEIVED_FWD_GETS,{ENQUEUE_REQUEST},MWA
EVENT_RECEIVED_FWD_GETX,{ENQUEUE_REQUEST},MWA

STATE SWAM
EVENT RECEIVED_DHM_ACK,{DEALLOCATE_MSHR,UNBLOCK_PROCESSOR_DHM_MIGR,DEQUEUE_REQUEST},S

STATE EWAM
EVENT RECEIVED_DHM_ACK,{DEALLOCATE_MSHR,UNBLOCK_PROCESSOR_DHM_MIGR,DEQUEUE_REQUEST},E
EVENT RECEIVED_FWD_GETS,{ENQUEUE_REQUEST},EWAM
EVENT_RECEIVED_FWD_GETX,{ENQUEUE_REQUEST},EWAM

STATE MWAM
EVENT RECEIVED_DHM_ACK,{DEALLOCATE_MSHR,UNBLOCK_PROCESSOR_DHM_MIGR,DEQUEUE_REQUEST},M
EVENT RECEIVED_FWD_GETS,{ENQUEUE_REQUEST},MWAM
EVENT_RECEIVED_FWD_GETX,{ENQUEUE_REQUEST},MWAM

STATE MWAA
EVENT RECEIVED_DHM_ACK,{SET_DHM_ACK_FLAG},IM
EVENT RECEIVED_ACK,{DECREMENT_ACK_COUNTER},MWAA
EVENT RECEIVED_LAST_ACK_AND_DATA,{NONE},MWA
EVENT RECEIVED_FWD_GETS,{ENQUEUE_REQUEST},MWAA
EVENT_RECEIVED_FWD_GETX,{ENQUEUE_REQUEST},MWAA
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},MWAA
EVENT RECEIVED_INV_REPL,{NONE},MWAA

STATE IMWA
EVENT RECEIVED_DHM_ACK,{SET_DHM_ACK_FLAG},IS

STATE SWAI
EVENT RECEIVED_DHM_ACK,{DEALLOCATE_MSHR,UNBLOCK_PROCESSOR,DEQUEUE_REQUEST,DEALLOCATE_L1},I

STATE IS
EVENT RECEIVED_DATA_SHARED_AT_REQUESTOR_MIGR,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,SEND_DATA_TO_L2_BANK,UNBLOCK_PROCESSOR_DHM_MIGR,DEQUEUE_REQUEST},S
EVENT RECEIVED_DATA_SHARED_AT_REQUESTOR_REPL,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,SEND_DATA_TO_L2_BANK,UNBLOCK_PROCESSOR_DHM_REPL,DEQUEUE_REQUEST},S
EVENT RECEIVED_DATA_SHARED_AT_REQUESTOR,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,UNBLOCK_PROCESSOR,DEQUEUE_REQUEST},S
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS_MIGR,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,SEND_DATA_TO_L2_BANK,UNBLOCK_PROCESSOR_DHM_MIGR,DEQUEUE_REQUEST},E
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,UNBLOCK_PROCESSOR,DEQUEUE_REQUEST},E
EVENT RECEIVED_FWD_GETS,{ENQUEUE_REQUEST},IS
EVENT_RECEIVED_FWD_GETX,{ENQUEUE_REQUEST},IS
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},ISI
EVENT RECEIVED_INV_REPL,{NONE},ISI
EVENT RECEIVED_INV_TO_OWNER,{NONE},IMI

STATE IM
EVENT RECEIVED_ACK,{DECREMENT_ACK_COUNTER},IM
EVENT RECEIVED_LAST_ACK_AND_DATA,{DEALLOCATE_MSHR,UNBLOCK_PROCESSOR,DEQUEUE_REQUEST},M
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS_MIGR,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,SEND_DATA_TO_L2_BANK,UNBLOCK_PROCESSOR_DHM_MIGR,DEQUEUE_REQUEST},M
EVENT RECEIVED_DATA_EXCLUSIVE_AT_REQUESTOR_MIGR,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,SEND_DATA_TO_L2_BANK},IMM
EVENT RECEIVED_DATA_EXCLUSIVE_AT_REQUESTOR,{COPY_DATA_TO_CACHE},IM
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,UNBLOCK_PROCESSOR,DEQUEUE_REQUEST},M
EVENT RECEIVED_FWD_GETS,{ENQUEUE_REQUEST},IM
EVENT_RECEIVED_FWD_GETX,{ENQUEUE_REQUEST},IM
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},IM
EVENT RECEIVED_INV_REPL,{NONE},IM
EVENT RECEIVED_INV_TO_OWNER,{NONE},IMI
EVENT RECEIVED_WBACK_TO_L1,{NONE},IM


STATE IMM
EVENT RECEIVED_ACK,{DECREMENT_ACK_COUNTER},IMM
```

```
EVENT RECEIVED_LAST_ACK_AND_DATA,{DEALLOCATE_MSHR,UNBLOCK_PROCESSOR_DHM_MIGR,DEQUEUE_REQUEST},M
EVENT RECEIVED_FWD_GETS,{ENQUEUE_REQUEST},IMM
EVENT_RECEIVED_FWD_GETX,{ENQUEUE_REQUEST},IMM
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},IMM
EVENT RECEIVED_INV_REPL,{NONE},IMM
EVENT RECEIVED_WBACK_TO_L1,{NONE},IMM

STATE IMMWA
EVENT RECEIVED_ACK,{DECREMENT_ACK_COUNTER},IMMWA
EVENT RECEIVED_LAST_ACK_AND_DATA,{NONE},MWAM
EVENT RECEIVED_FWD_GETS,{ENQUEUE_REQUEST},IMMWA
EVENT_RECEIVED_FWD_GETX,{ENQUEUE_REQUEST},IMMWA
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},IMMWA
EVENT RECEIVED_INV_REPL,{NONE},IMMWA
EVENT RECEIVED_WBACK_TO_L1,{NONE},IMMWA
EVENT RECEIVED_DHM_ACK,{SET_DHM_ACK_FLAG},IMM

STATE S
EVENT WRONG_BANK,{MOVE_CACHE_BLOCK,REPEAT_REQUEST},S
EVENT CANT_ALLOCATE,{ENQUEUE_REQUEST_REPL,TRIGGER_REPLACEMENT},S
EVENT LOAD,{UNBLOCK_PROCESSOR},S
EVENT FETCH,{UNBLOCK_PROCESSOR},S
EVENT STORE,{ALLOCATE_MSHR,SEND_GETX_TO_L2_DHM},IMWA
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR,DEALLOCATE_L1},I
EVENT RECEIVED_INV_REPL,{DEALLOCATE_L1},I
EVENT L1_REPLACEMENT,{DEQUEUE_REQUEST_REPL,DEALLOCATE_L1},I

STATE E
EVENT WRONG_BANK,{MOVE_CACHE_BLOCK,REPEAT_REQUEST},E
EVENT CANT_ALLOCATE,{ENQUEUE_REQUEST_REPL,TRIGGER_REPLACEMENT},E
EVENT LOAD,{UNBLOCK_PROCESSOR},E
EVENT FETCH,{UNBLOCK_PROCESSOR},E
EVENT STORE,{UNBLOCK_PROCESSOR},M
EVENT RECEIVED_FWD_GETS,{SEND_ACCEPTS_TO_L2_DHM,SEND_DATA_SHARED_FROM_OWNER_TO_REQUESTOR},S
EVENT RECEIVED_FWD_GETX,{SEND_DATA_EXCLUSIVE_FROM_OWNER_TO_REQUESTOR,DEALLOCATE_L1},I
EVENT L1_REPLACEMENT,{SEND_PUTX_TO_L2_DHM,ALLOCATE_MSHR,DEQUEUE_REQUEST_REPL_MSHR,DEALLOCATE_L1},EI
EVENT RECEIVED_INV_TO_OWNER,{SEND_PUTS_TO_L2},EI

STATE M
EVENT WRONG_BANK,{MOVE_CACHE_BLOCK,REPEAT_REQUEST},M
EVENT CANT_ALLOCATE,{ENQUEUE_REQUEST_REPL,TRIGGER_REPLACEMENT},M
EVENT LOAD,{UNBLOCK_PROCESSOR},M
EVENT STORE,{UNBLOCK_PROCESSOR},M
EVENT FETCH,{UNBLOCK_PROCESSOR},M
EVENT RECEIVED_FWD_GETS,{SEND_DATA_SHARED_FROM_OWNER_TO_REQUESTOR,SEND_ACCEPTS_TO_L2_DHM},S
EVENT RECEIVED_FWD_GETX,{SEND_DATA_EXCLUSIVE_FROM_OWNER_TO_REQUESTOR,DEALLOCATE_L1},I
EVENT L1_REPLACEMENT,{SEND_PUTX_TO_L2_DHM,ALLOCATE_MSHR,DEQUEUE_REQUEST_REPL_MSHR,DEALLOCATE_L1},MI
EVENT RECEIVED_INV_TO_OWNER,{SEND_PUTX_TO_L2},MI

STATE MI
EVENT RECEIVED_WBACK_TO_L1,{DEALLOCATE_MSHR},I
EVENT L1_REPLACEMENT,{RECYCLE_QUEUED_REQUEST},MI
EVENT RECEIVED_INV_TO_OWNER,{NONE},MI
EVENT RECEIVED_FWD_GETS,{SEND_DATA_SHARED_FROM_OWNER_TO_REQUESTOR},MI
EVENT RECEIVED_FWD_GETX,{SEND_DATA_EXCLUSIVE_FROM_OWNER_TO_REQUESTOR},MI
EVENT LOAD,{RECYCLE_REQUEST},MI
EVENT FETCH,{RECYCLE_REQUEST},MI
EVENT STORE,{RECYCLE_REQUEST},MI

STATE EI
EVENT RECEIVED_WBACK_TO_L1,{DEALLOCATE_MSHR},I
EVENT L1_REPLACEMENT,{RECYCLE_QUEUED_REQUEST},EI
EVENT RECEIVED_INV_TO_OWNER,{NONE},EI
EVENT RECEIVED_FWD_GETS,{SEND_DATA_SHARED_FROM_OWNER_TO_REQUESTOR},EI
EVENT RECEIVED_FWD_GETX,{SEND_DATA_EXCLUSIVE_FROM_OWNER_TO_REQUESTOR},EI
EVENT LOAD,{RECYCLE_REQUEST},EI
EVENT FETCH,{RECYCLE_REQUEST},EI
EVENT STORE,{RECYCLE_REQUEST},EI

STATE ISI
EVENT RECEIVED_DATA_SHARED_AT_REQUESTOR,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,UNBLOCK_PROCESSOR,DEALLOCATE_L1},I
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,UNBLOCK_PROCESSOR,DEALLOCATE_L1},I

STATE IMI
EVENT RECEIVED_ACK,{DECREMENT_ACK_COUNTER},IMI
EVENT RECEIVED_LAST_ACK_AND_DATA,{UNBLOCK_PROCESSOR,SEND_PUTX_TO_L2_DHM,DEQUEUE_REQUEST},MI
EVENT RECEIVED_DATA_EXCLUSIVE_AT_REQUESTOR,{COPY_DATA_TO_CACHE},IMI
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS,{UNBLOCK_PROCESSOR,SEND_PUTX_TO_L2_DHM,DEQUEUE_REQUEST},MI

DEVICE L2
```

```
LIST_STATES I,IP,P,PS,S,C,PI,IR,IPW,IPR,PW,IM,WM,PWM,SWM,WR,IReplica,Replica,ReplicaWM,ReplicaWE,ReplicaWME,Sr,SrP,SrP2,ReplicaX,ReplicaI,SrI

STATE I
EVENT CANT_ALLOCATE,{SET_HOME,ENQUEUE_REQUEST_REPL,TRIGGER_REPLACEMENT},I
EVENT CANT_ALLOCATE_AND_CANT_REPLACE,{RECYCLE_REQUEST_TO_L2},I
EVENT MULTIPLE_REQUEST,{SEND_RETRY},I
EVENT RECEIVED_GETS,{ALLOCATE_MSHR,SEND_REQUEST_TO_L2_BANKS},IR
EVENT RECEIVED_GETX,{ALLOCATE_MSHR,SEND_REQUEST_TO_L2_BANKS},IR
EVENT RECEIVED_GETS_TO_L2_BANKS,{SEND_ACK_TO_REQ_TILE},I
EVENT RECEIVED_GETX_TO_L2_BANKS,{SEND_ACK_TO_REQ_TILE},I
EVENT RECEIVED_REPLICATION_HOME,{NONE},I
EVENT CHANGE_REPLICATION_HOME,{SEND_REPLICATION_HOME_NACK},I
EVENT CHANGE_REPLICATION_HOME_NO_REPLICAS,{SEND_REPLICATION_HOME_NACK},I

STATE IR
EVENT RECEIVED_ACK,{DECREMENT_DHM_ACK_COUNTER},IR
EVENT RECEIVED_LAST_ACK,{SEND_DHM_ACK_TO_L1_COND,SEND_GET_TO_MC_DHM},I
EVENT HOME_BANK,{ALLOCATE_L2,SEND_DHM_ACK_TO_L1,SAVE_OWNER},IPW
EVENT HOME_TILE,{ALLOCATE_L2,SEND_DHM_ACK_TO_L1,SAVE_OWNER},IP
EVENT RETRY,{SEND_REQUEST_TO_L2_BANKS},IR
EVENT RECEIVED_LAST_ACK_MIGR,{DECREMENT_DHM_ACK_COUNTER,SEND_DHM_ACK_TO_L1,ALLOCATE_L2},IM
EVENT RECEIVED_ACK_MIGR,{DECREMENT_DHM_ACK_COUNTER,ALLOCATE_L2},IM
EVENT RECEIVED_LAST_ACK_REPL,{DECREMENT_DHM_ACK_COUNTER,SEND_DHM_ACK_TO_L1,ALLOCATE_L2},IReplica
EVENT RECEIVED_ACK_REPL,{DECREMENT_DHM_ACK_COUNTER,ALLOCATE_L2},IReplica

STATE IPW
EVENT RECEIVED_GETS,{ENQUEUE_REQUEST},IP
EVENT RECEIVED_GETX,{ENQUEUE_REQUEST},IP
EVENT RECEIVED_ACK,{DECREMENT_DHM_ACK_COUNTER},IPW
EVENT RECEIVED_LAST_ACK,{SEND_DHM_ACK_TO_L1},IPR
EVENT RETRY,{NONE},IPR
EVENT RECEIVED_GETX_TO_L2_BANKS,{ENQUEUE_REQUEST},IPW
EVENT RECEIVED_GETS_TO_L2_BANKS,{ENQUEUE_REQUEST},IPW
EVENT L2_REPLACEMENT,{RECYCLE_REQUEST},IPW
EVENT RECEIVED_DATA,{COPY_DATA_TO_CACHE,SEND_DATA_EXCLUSIVE_TO_REQUESTOR},PW

STATE IPR
EVENT RECEIVED_GETX_TO_L2_BANKS,{ENQUEUE_REQUEST},IPR
EVENT RECEIVED_GETS_TO_L2_BANKS,{ENQUEUE_REQUEST},IPR
EVENT L2_REPLACEMENT,{RECYCLE_REQUEST},IPR
EVENT RECEIVED_DATA,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,SEND_DATA_EXCLUSIVE_TO_REQUESTOR,DEQUEUE_REQUEST},P

STATE IP
EVENT RECEIVED_GETS,{ENQUEUE_REQUEST},IP
EVENT RECEIVED_GETX,{ENQUEUE_REQUEST},IP
EVENT RECEIVED_GETX_TO_L2_BANKS,{ENQUEUE_REQUEST},IP
EVENT RECEIVED_GETS_TO_L2_BANKS,{ENQUEUE_REQUEST},IP
EVENT L2_REPLACEMENT,{RECYCLE_REQUEST},IP
EVENT RECEIVED_DATA,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,SEND_DATA_EXCLUSIVE_TO_REQUESTOR,DEQUEUE_REQUEST},P

STATE PW
EVENT RECEIVED_ACK,{DECREMENT_DHM_ACK_COUNTER},PW
EVENT RECEIVED_LAST_ACK,{DEALLOCATE_MSHR,SEND_DHM_ACK_TO_L1,DEQUEUE_REQUEST},P
EVENT RETRY,{REPEAT_TILE_REQUEST,DEQUEUE_REQUEST},P
EVENT RECEIVED_GETX,{ENQUEUE_REQUEST},PW
EVENT RECEIVED_GETS,{ENQUEUE_REQUEST},PW
EVENT RECEIVED_GETX_TO_L2_BANKS,{ENQUEUE_REQUEST},PW
EVENT RECEIVED_GETS_TO_L2_BANKS,{ENQUEUE_REQUEST},PW
EVENT L2_REPLACEMENT,{RECYCLE_REQUEST},PW

STATE P
EVENT DHM_MIGRATION_X,{SEND_ACK_TO_REQ_TILE_MIGR,FWD_REQUEST_TO_OWNER_DHM_MIGR},WM
EVENT RECEIVED_GETX,{SEND_DHM_ACK_TO_L1,UPDATE_DHM_MIGR_COUNTERS,FWD_REQUEST_TO_OWNER,SAVE_OWNER,SET_HOME},P
EVENT RECEIVED_GETS,{SEND_DHM_ACK_TO_L1,RESET_DHM_MIGR_COUNTERS,RESET_DHM_REPL_COUNTERS,FWD_REQUEST_TO_OWNER,ADD_SHARER,SET_HOME},PS
EVENT RECEIVED_GETX_TO_L2_BANKS,{UPDATE_DHM_MIGR_COUNTERS,FWD_REQUEST_TO_OWNER,SAVE_OWNER,SEND_ACK_TO_REQ_TILE_HIT,SET_HOME},P
EVENT RECEIVED_GETS_TO_L2_BANKS,{UPDATE_DHM_MIGR_COUNTERS,FWD_REQUEST_TO_OWNER,ADD_SHARER,SEND_ACK_TO_REQ_TILE_HIT,SET_HOME},PS
EVENT RECEIVED_ACCEPTS_FROM_OLD_OWNER,{NONE},P
EVENT RECEIVED_PUTX_TO_L2,{COPY_DATA_TO_CACHE,SEND_WBACK_TO_REQUESTOR},C
EVENT RECEIVED_PUTS_TO_L2,{SEND_WBACK_TO_REQUESTOR},P
EVENT RECEIVED_PUTX_FROM_OLD_OWNER,{SEND_WBACK_TO_REQUESTOR},P
EVENT L2_REPLACEMENT,{SEND_INV_TO_OWNER,ALLOCATE_MSHR,DEQUEUE_REQUEST_REPL_MSHR,DEALLOCATE_L2},PI

STATE S
EVENT DHM_REPLICATION_S,{SAVE_REPLICA,ADD_SHARER,SEND_DATA_SHARED_TO_REQUESTOR_REPL,SEND_ACK_TO_REQ_TILE_REPL},WR
EVENT RECEIVED_GETS,{SEND_DHM_ACK_TO_L1,UPDATE_DHM_REPL_COUNTERS,ADD_SHARER,SEND_DATA_SHARED_TO_REQUESTOR},S
EVENT RECEIVED_GETX,{SEND_DHM_ACK_TO_L1,RESET_DHM_REPL_COUNTERS,REMOVE_REQUESTOR_FROM_SHARERS,SAVE_OWNER,SEND_DATA_EXCLUSIVE_TO_REQUESTOR,
          SEND_INV_TO_SHARERS,CLEAR_SHARERS},P
EVENT RECEIVED_GETX_TO_L2_BANKS,{RESET_DHM_REPL_COUNTERS,REMOVE_REQUESTOR_FROM_SHARERS,SAVE_OWNER,SEND_DATA_EXCLUSIVE_TO_REQUESTOR,
          SEND_INV_TO_SHARERS,CLEAR_SHARERS,SEND_ACK_TO_REQ_TILE_HIT},P
EVENT RECEIVED_GETS_TO_L2_BANKS,{UPDATE_DHM_REPL_COUNTERS,ADD_SHARER,SEND_DATA_SHARED_TO_REQUESTOR,SEND_ACK_TO_REQ_TILE_HIT},S
```

```
EVENT RECEIVED_PUTS_TO_L2,{REMOVE_SHARER_FROM_DIRECTORY,SEND_WBACK_TO_REQUESTOR},S
EVENT RECEIVED_LAST_PUTS_TO_L2,{REMOVE_SHARER_FROM_DIRECTORY,SEND_WBACK_TO_REQUESTOR},C
EVENT L2_REPLACEMENT,{INVALIDATE_SHARERS_REPL,CONDITIONAL_SEND_DATA_TO_MC,DEQUEUE_REQUEST_REPL,DEALLOCATE_L2},I

STATE C
EVENT DHM_MIGRATION_S,{SEND_ACK_TO_REQ_TILE_MIGR,SEND_DATA_EXCLUSIVE_TO_REQUESTOR_MIGR},WM
EVENT DHM_MIGRATION_X,{SEND_ACK_TO_REQ_TILE_MIGR,SEND_DATA_EXCLUSIVE_TO_REQUESTOR_MIGR},WM
EVENT RECEIVED_GETX,{SEND_DHM_ACK_TO_L1,UPDATE_DHM_MIGR_COUNTERS,SAVE_OWNER,SEND_DATA_EXCLUSIVE_TO_REQUESTOR},P
EVENT RECEIVED_GETS,{SEND_DHM_ACK_TO_L1,UPDATE_DHM_MIGR_COUNTERS,SAVE_OWNER,SEND_DATA_EXCLUSIVE_TO_REQUESTOR},P
EVENT RECEIVED_GETX_TO_L2_BANKS,{UPDATE_DHM_MIGR_COUNTERS,SAVE_OWNER,SEND_DATA_EXCLUSIVE_TO_REQUESTOR,SEND_ACK_TO_REQ_TILE_HIT},P
EVENT RECEIVED_GETS_TO_L2_BANKS,{UPDATE_DHM_MIGR_COUNTERS,SAVE_OWNER,SEND_DATA_EXCLUSIVE_TO_REQUESTOR,SEND_ACK_TO_REQ_TILE_HIT},P
EVENT L2_REPLACEMENT,{CONDITIONAL_SEND_DATA_TO_MC,DEQUEUE_REQUEST_REPL,DEALLOCATE_L2},I

STATE PS
EVENT RECEIVED_ACCEPTS_TO_L2,{COPY_DATA_TO_CACHE,ADD_OWNER_TO_SHARERS,DEQUEUE_REQUEST},S
EVENT RECEIVED_ACCEPTS_FROM_OLD_OWNER,{NONE},PS
EVENT L2_REPLACEMENT,{RECYCLE_REQUEST},PS
EVENT RECEIVED_PUTX_TO_L2,{COPY_DATA_TO_CACHE,SEND_WBACK_TO_REQUESTOR,DEQUEUE_REQUEST},S
EVENT RECEIVED_PUTX_FROM_OLD_OWNER,{SEND_WBACK_TO_REQUESTOR},PS
EVENT RECEIVED_GETX,{ENQUEUE_REQUEST},PS
EVENT RECEIVED_GETS,{ENQUEUE_REQUEST},PS
EVENT RECEIVED_GETX_TO_L2_BANKS,{ENQUEUE_REQUEST},PS
EVENT RECEIVED_GETS_TO_L2_BANKS,{ENQUEUE_REQUEST},PS
EVENT RECEIVED_PUTS_TO_L2,{REMOVE_SHARER_FROM_DIRECTORY,SEND_WBACK_TO_REQUESTOR},PS

STATE PI
EVENT RECEIVED_PUTS_TO_L2,{SEND_WBACK_TO_REQUESTOR,DEALLOCATE_MSHR},I
EVENT RECEIVED_PUTX_TO_L2,{SEND_WBACK_TO_REQUESTOR,SEND_DATA_TO_MC,DEALLOCATE_MSHR},I
EVENT RECEIVED_PUTX_TO_L2_H,{SEND_WBACK_TO_REQUESTOR,SEND_DATA_TO_MC,DEALLOCATE_MSHR},I
EVENT L2_REPLACEMENT,{REPEAT_REQUEST_REPL},PI
EVENT RECEIVED_GETS,{ENQUEUE_REQUEST},PI
EVENT RECEIVED_GETX,{ENQUEUE_REQUEST},PI
EVENT RECEIVED_GET_MSHR,{RECYCLE_REQUEST_TO_L2},PI
EVENT RECEIVED_GETX_TO_L2_BANKS,{SEND_RETRY},PI
EVENT RECEIVED_GETS_TO_L2_BANKS,{SEND_RETRY},PI

STATE WM
EVENT RECEIVED_GETX_TO_L2_BANKS,{SEND_RETRY},WM
EVENT RECEIVED_GETS_TO_L2_BANKS,{SEND_RETRY},WM
EVENT RECEIVED_GETS,{ENQUEUE_REQUEST},WM
EVENT RECEIVED_GETX,{ENQUEUE_REQUEST},WM
EVENT RECEIVED_DHM_MIGRATION_END,{DEQUEUE_REQUEST,DEALLOCATE_L2},I
EVENT RECEIVED_PUTX_TO_L2,{SEND_WBACK_TO_REQUESTOR},WM

STATE PWM
EVENT RECEIVED_ACK,{DECREMENT_DHM_ACK_COUNTER},PWM
EVENT RECEIVED_LAST_ACK,{DEALLOCATE_MSHR,DECREMENT_DHM_ACK_COUNTER,SEND_DHM_ACK_TO_L1,SEND_DHM_MIGRATION_END,DEQUEUE_REQUEST},P
EVENT RETRY,{SEND_DHM_MIGRATION_END,DEQUEUE_REQUEST},P
EVENT RECEIVED_GETX_TO_L2_BANKS,{ENQUEUE_REQUEST},PWM
EVENT RECEIVED_GETS_TO_L2_BANKS,{ENQUEUE_REQUEST},PWM
EVENT L2_REPLACEMENT,{RECYCLE_REQUEST},PWM
EVENT RECEIVED_PUTX_TO_L2,{ENQUEUE_REQUEST},PWM

STATE IM
EVENT RECEIVED_GETX,{RECYCLE_REQUEST_TO_L2},IM
EVENT RECEIVED_GETX_TO_L2_BANKS,{SEND_RETRY},IM
EVENT RECEIVED_GETS_TO_L2_BANKS,{SEND_RETRY},IM
EVENT RECEIVED_DATA_TO_L2_BANK_MIGR_P,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,SAVE_OWNER,SEND_DHM_MIGRATION_END,DEQUEUE_REQUEST},P
EVENT RECEIVED_DATA_TO_L2_BANK_MIGR_S,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,COPY_SHARERS,SEND_DHM_MIGRATION_END,DEQUEUE_REQUEST},S
EVENT RECEIVED_DATA_MIGR_WAIT_ACKS_P,{COPY_DATA_TO_CACHE,SAVE_OWNER},PWM
EVENT RECEIVED_DATA_MIGR_WAIT_ACKS_S,{COPY_DATA_TO_CACHE,COPY_SHARERS},SWM
EVENT RECEIVED_ACK,{DECREMENT_DHM_ACK_COUNTER},IM
EVENT RECEIVED_LAST_ACK,{SEND_DHM_ACK_TO_L1,DECREMENT_DHM_ACK_COUNTER},IM

STATE IReplica
EVENT RECEIVED_GETX,{RECYCLE_REQUEST_TO_L2},IReplica
EVENT RECEIVED_GETS_AT_REPLICA,{ENQUEUE_REQUEST},IReplica
EVENT RECEIVED_GETX_TO_L2_BANKS,{SEND_ACK_TO_REQ_TILE},IReplica
EVENT RECEIVED_GETS_TO_L2_BANKS,{SEND_ACK_TO_REQ_TILE},IReplica
EVENT RECEIVED_DATA_TO_L2_BANK_REPLICA,{COPY_DATA_TO_CACHE,SEND_REPLICATION_ACK,DEQUEUE_REQUEST},ReplicaWE
EVENT RECEIVED_DATA_REPL_WAIT_ACKS,{COPY_DATA_TO_CACHE,SEND_REPLICATION_ACK},ReplicaWME
EVENT RECEIVED_ACK,{DECREMENT_DHM_ACK_COUNTER},IReplica
EVENT RECEIVED_LAST_ACK,{SEND_DHM_ACK_TO_L1,DECREMENT_DHM_ACK_COUNTER},IReplica

STATE ReplicaWE
EVENT RECEIVED_GETS_AT_REPLICA,{ENQUEUE_REQUEST},ReplicaWE
EVENT RECEIVED_GETX,{ENQUEUE_REQUEST},ReplicaWE
EVENT RECEIVED_GETX_TO_L2_BANKS,{SEND_ACK_TO_REQ_TILE},ReplicaWE
EVENT RECEIVED_GETS_TO_L2_BANKS,{SEND_ACK_TO_REQ_TILE},ReplicaWE
EVENT RECEIVED_REPLICATION_END,{DEALLOCATE_MSHR,DEQUEUE_REQUEST},Replica
```

```
EVENT RECEIVED_INV_TO_REPLICA,{ENQUEUE_REQUEST},ReplicaWE

STATE ReplicaWME
EVENT RECEIVED_GETS_AT_REPLICA,{ENQUEUE_REQUEST},ReplicaWME
EVENT RECEIVED_GETX_TO_L2_BANKS,{SEND_ACK_TO_REQ_TILE},ReplicaWME
EVENT RECEIVED_GETS_TO_L2_BANKS,{SEND_ACK_TO_REQ_TILE},ReplicaWME
EVENT RECEIVED_ACK,{DECREMENT_DHM_ACK_COUNTER},ReplicaWME
EVENT RECEIVED_LAST_ACK,{SEND_DHM_ACK_TO_L1,DECREMENT_DHM_ACK_COUNTER},ReplicaWE
EVENT RECEIVED_REPLICATION_END,{DEQUEUE_REQUEST},ReplicaWM

STATE ReplicaWM
EVENT RECEIVED_GETS_AT_REPLICA,{ENQUEUE_REQUEST},ReplicaWM
EVENT RECEIVED_GETX_TO_L2_BANKS,{SEND_ACK_TO_REQ_TILE},ReplicaWM
EVENT RECEIVED_GETS_TO_L2_BANKS,{SEND_ACK_TO_REQ_TILE},ReplicaWM
EVENT RECEIVED_ACK,{DECREMENT_DHM_ACK_COUNTER},ReplicaWM
EVENT RECEIVED_LAST_ACK,{DEALLOCATE_MSHR,SEND_DHM_ACK_TO_L1,DECREMENT_DHM_ACK_COUNTER,DEQUEUE_REQUEST},Replica

STATE SWM
EVENT RECEIVED_ACK,{DECREMENT_DHM_ACK_COUNTER},SWM
EVENT RECEIVED_LAST_ACK,{DEALLOCATE_MSHR,SEND_DHM_ACK_TO_L1,SEND_DHM_MIGRATION_END,DEQUEUE_REQUEST},S
EVENT RETRY,{SEND_DHM_MIGRATION_END,DEQUEUE_REQUEST},S
EVENT RECEIVED_GETX_TO_L2_BANKS,{ENQUEUE_REQUEST},SWM
EVENT RECEIVED_GETS_TO_L2_BANKS,{ENQUEUE_REQUEST},SWM
EVENT L2_REPLACEMENT,{RECYCLE_REQUEST},SWM

STATE WR
EVENT RECEIVED_GETS_REPLICA,{ADD_SHARER,SEND_RETRY},WR
EVENT RECEIVED_GETX_TO_L2_BANKS,{SEND_RETRY},WR
EVENT RECEIVED_GETS_TO_L2_BANKS,{SEND_RETRY},WR
EVENT RECEIVED_GETS,{ENQUEUE_REQUEST},WR
EVENT RECEIVED_GETX,{ENQUEUE_REQUEST},WR
EVENT RECEIVED_REPLICATION_ACK,{SEND_REPLICATION_END,DEQUEUE_REQUEST},Sr
EVENT RECEIVED_PUTX_TO_L2,{SEND_WBACK_TO_REQUESTOR},WR
EVENT RECEIVED_REPLICA_PUTS,{REMOVE_REPLICA},WR

STATE Sr
EVENT RECEIVED_GETX,{RESET_DHM_REPL_COUNTERS,REMOVE_REQUESTOR_FROM_SHARERS,SAVE_OWNER,SEND_DATA_EXCLUSIVE_TO_REQUESTOR,SEND_INV_TO_SHARERS_REPL,
          SEND_INV_TO_REPLICAS,CLEAR_SHARERS,CLEAR_REPLICAS},SrP
EVENT RECEIVED_GETS_REPLICA,{ADD_SHARER,SEND_RETRY},Sr
EVENT DHM_REPLICATION_S,{SAVE_REPLICA,ADD_SHARER,SEND_DATA_SHARED_TO_REQUESTOR_REPL,SEND_ACK_TO_REQ_TILE_REPL},WR
EVENT RECEIVED_GETS,{SEND_DHM_ACK_TO_L1,UPDATE_DHM_REPL_COUNTERS,ADD_SHARER,SEND_DATA_SHARED_TO_REQUESTOR},Sr
EVENT RECEIVED_GETX_TO_L2_BANKS,{RESET_DHM_REPL_COUNTERS,REMOVE_REQUESTOR_FROM_SHARERS,SAVE_OWNER,SEND_ACK_TO_REQ_TILE_HIT,
          SEND_DATA_EXCLUSIVE_TO_REQUESTOR,SEND_INV_TO_SHARERS_REPL,SEND_INV_TO_REPLICAS,CLEAR_SHARERS,CLEAR_REPLICAS},SrP2
EVENT RECEIVED_GETS_TO_L2_BANKS,{UPDATE_DHM_REPL_COUNTERS,ADD_SHARER,SEND_DATA_SHARED_TO_REQUESTOR,SEND_ACK_TO_REQ_TILE_HIT},Sr
EVENT RECEIVED_LAST_REPLICA_PUTS,{REMOVE_REPLICA},S
EVENT RECEIVED_REPLICA_PUTS,{REMOVE_REPLICA},Sr
EVENT L2_REPLACEMENT,{CHANGE_REPLICATION_HOME,SEND_REPLICATION_HOME_TO_REPLICAS},SrI

STATE SrI
EVENT RECEIVED_REPLICA_PUTS,{NONE},SrI
EVENT RECEIVED_WBACK_TO_L2,{DEQUEUE_REQUEST_REPL,DEALLOCATE_L2},I
EVENT RECEIVED_GETX_TO_L2_BANKS,{SEND_RETRY},SrI
EVENT RECEIVED_GETS_TO_L2_BANKS,{SEND_RETRY},SrI
EVENT RECEIVED_REPL_HOME_NACK_NO_REPLICAS,{INVALIDATE_SHARERS_REPL,CONDITIONAL_SEND_DATA_TO_MC,DEQUEUE_REQUEST_REPL,DEALLOCATE_L2},I
EVENT RECEIVED_REPL_HOME_NACK,{CHANGE_REPLICATION_HOME,SEND_REPLICATION_HOME_TO_REPLICAS},SrI

STATE Replica
EVENT RECEIVED_GETX,{SEND_REQUEST_TO_L2_BANKS},ReplicaX
EVENT RECEIVED_GETX_TO_L2_BANKS,{SEND_ACK_TO_REQ_TILE},Replica
EVENT RECEIVED_GETS_AT_REPLICA,{SEND_ACK_TO_REQ_TILE_HIT,ADD_HIT_TO_STATS,SEND_DATA_SHARED_TO_REQUESTOR},Replica
EVENT RECEIVED_GETS_TO_L2_BANKS,{SEND_ACK_TO_REQ_TILE},Replica
EVENT RECEIVED_GETS,{SEND_DHM_ACK_TO_L1,SEND_DATA_SHARED_TO_REQUESTOR},Replica
EVENT RECEIVED_INV_TO_REPLICA,{SEND_INV_FROM_REPLICA_TO_SHARERS,SEND_REPL_ACK_TO_HOME,SEND_ACK_TO_REQUESTOR_NOC,DEALLOCATE_L2},I
EVENT L2_REPLACEMENT,{SEND_REPLICA_PUTS,DEQUEUE_REQUEST_REPL,DEALLOCATE_L2},I
EVENT CHANGE_REPLICATION_HOME_NO_REPLICAS,{COPY_SHARERS,SEND_REPLICATION_HOME_WBACK},S
EVENT CHANGE_REPLICATION_HOME,{COPY_SHARERS,COPY_REPLICAS,SEND_REPLICATION_HOME_WBACK},Sr
EVENT RECEIVED_REPLICATION_HOME,{SAVE_REPLICATION_HOME},Replica

STATE SrP
EVENT RECEIVED_REPL_ACK,{DECREMENT_REPL_ACK_COUNTER},SrP
EVENT RECEIVED_LAST_REPL_ACK,{SEND_DHM_ACK_TO_L1,DEQUEUE_REQUEST},P
EVENT RECEIVED_GETX,{ENQUEUE_REQUEST},SrP
EVENT RECEIVED_GETS,{ENQUEUE_REQUEST},SrP
EVENT RECEIVED_GETX_TO_L2_BANKS,{SEND_RETRY},SrP
EVENT RECEIVED_GETS_TO_L2_BANKS,{SEND_RETRY},SrP

STATE SrP2
EVENT RECEIVED_REPL_ACK,{DECREMENT_REPL_ACK_COUNTER},SrP2
EVENT RECEIVED_LAST_REPL_ACK,{DEQUEUE_REQUEST},P
EVENT RECEIVED_GETX,{ENQUEUE_REQUEST},SrP2
```

```
EVENT RECEIVED_GETS,{ENQUEUE_REQUEST},SrP2
EVENT RECEIVED_GETX_TO_L2_BANKS,{SEND_RETRY},SrP2
EVENT RECEIVED_GETS_TO_L2_BANKS,{SEND_RETRY},SrP2

STATE ReplicaX
EVENT RECEIVED_INV_TO_REPLICA,{SEND_INV_FROM_REPLICA_TO_SHARERS,SEND_REPL_ACK_TO_HOME,SEND_ACK_TO_REQUESTOR_NOC,DEALLOCATE_L2},I
EVENT RECEIVED_ACK,{DECREMENT_DHM_ACK_COUNTER},ReplicaX
EVENT RECEIVED_LAST_ACK,{SEND_DHM_ACK_TO_L1,DECREMENT_DHM_ACK_COUNTER},ReplicaX
EVENT RECEIVED_GETS_TO_L2_BANKS,{SEND_RETRY},ReplicaX
EVENT RECEIVED_GETX_TO_L2_BANKS,{SEND_RETRY},ReplicaX
```

A.4 Hammer + RHM

This section describes Hammer protocol with RHM. MESI states are assumed at L1 caches. This protocol is used in Chapter 4.

```
DEVICE L1
LIST_STATES I,IS,IM,S,E,M,EI,MI,ISI,IMI,ISWA,IMWA,SWA,MWA,EWA,MWAA,SWAI

STATE I
EVENT CANT_ALLOCATE,{ENQUEUE_REQUEST_REPL,TRIGGER_REPLACEMENT},I
EVENT LOAD,{ALLOCATE_MSHR,ALLOCATE_L1,SEND_GETS_TO_L2_DHM},ISWA
EVENT FETCH,{ALLOCATE_MSHR,ALLOCATE_L1,SEND_GETS_TO_L2_DHM},ISWA
EVENT STORE,{ALLOCATE_MSHR,ALLOCATE_L1,SEND_GETX_TO_L2_DHM},IMWA
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},I
EVENT RECEIVED_INV_REPL,{NONE},I
EVENT RECEIVED_BCAST_REQ,{SEND_ACK_TO_REQUESTOR},I

STATE ISWA
EVENT RECEIVED_LAST_ACK_AND_DATA,{DECREMENT_ACK_COUNTER},SWA
EVENT RECEIVED_ACK,{DECREMENT_ACK_COUNTER},ISWA
EVENT RECEIVED_DATA_SHARED_AT_REQUESTOR_WAIT_ACKS,{COPY_DATA_TO_CACHE},ISWA
EVENT RECEIVED_DATA_SHARED_AT_REQUESTOR,{COPY_DATA_TO_CACHE},SWA
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS,{COPY_DATA_TO_CACHE},EWA
EVENT RECEIVED_DHM_ACK,{SET_DHM_ACK_FLAG},IS
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},ISWA
EVENT RECEIVED_BCAST_REQ,{SEND_ACK_TO_REQUESTOR},ISWA

STATE IMWA
EVENT RECEIVED_DHM_ACK,{SET_DHM_ACK_FLAG},IM
EVENT RECEIVED_ACK,{DECREMENT_ACK_COUNTER},IMWA
EVENT RECEIVED_LAST_ACK_AND_DATA,{NONE},MWA
EVENT RECEIVED_DATA_EXCLUSIVE_AT_REQUESTOR,{COPY_DATA_TO_CACHE},MWAA
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS,{COPY_DATA_TO_CACHE},MWA
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},IMWA
EVENT RECEIVED_INV_REPL,{NONE},IMWA
EVENT RECEIVED_BCAST_REQ,{SEND_ACK_TO_REQUESTOR},IMWA

STATE SWA
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},SWAI
EVENT RECEIVED_DHM_ACK,{DEALLOCATE_MSHR,UNBLOCK_PROCESSOR,SEND_ACK_TO_HOME,DEQUEUE_REQUEST},S

STATE EWA
EVENT RECEIVED_DHM_ACK,{DEALLOCATE_MSHR,UNBLOCK_PROCESSOR,SEND_ACK_TO_HOME,DEQUEUE_REQUEST},E
EVENT RECEIVED_FWD_GETS,{ENQUEUE_REQUEST},EWA
EVENT RECEIVED_FWD_GETX,{ENQUEUE_REQUEST},EWA

STATE MWA
EVENT RECEIVED_DHM_ACK,{DEALLOCATE_MSHR,UNBLOCK_PROCESSOR,SEND_ACK_TO_HOME,DEQUEUE_REQUEST},M
EVENT RECEIVED_FWD_GETS,{ENQUEUE_REQUEST},MWA
EVENT RECEIVED_FWD_GETX,{ENQUEUE_REQUEST},MWA

STATE MWAA
EVENT RECEIVED_DHM_ACK,{SET_DHM_ACK_FLAG},IM
EVENT RECEIVED_ACK,{DECREMENT_ACK_COUNTER},MWAA
EVENT RECEIVED_LAST_ACK_AND_DATA,{NONE},MWA
EVENT RECEIVED_FWD_GETS,{ENQUEUE_REQUEST},MWAA
EVENT RECEIVED_FWD_GETX,{ENQUEUE_REQUEST},MWAA
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},MWAA
EVENT RECEIVED_INV_REPL,{NONE},MWAA

STATE SWAI
```

```
EVENT RECEIVED_DHM_ACK,{DEALLOCATE_MSHR,UNBLOCK_PROCESSOR,SEND_ACK_TO_HOME,DEQUEUE_REQUEST,DEALLOCATE_L1},I

STATE IS
EVENT RECEIVED_ACK,{DECREMENT_ACK_COUNTER},IS
EVENT RECEIVED_LAST_ACK_AND_DATA,{DEALLOCATE_MSHR,SEND_ACK_TO_HOME,UNBLOCK_PROCESSOR,DEQUEUE_REQUEST},S
EVENT RECEIVED_DATA_SHARED_AT_REQUESTOR_WAIT_ACKS,{COPY_DATA_TO_CACHE},IS
EVENT RECEIVED_DATA_SHARED_AT_REQUESTOR,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,SEND_ACK_TO_HOME,UNBLOCK_PROCESSOR,DEQUEUE_REQUEST},S
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS,{COPY_DATA_TO_CACHE,SEND_ACK_TO_HOME,UNBLOCK_PROCESSOR,DEQUEUE_REQUEST},E
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},IS
EVENT RECEIVED_INV_REPL,{NONE},ISI
EVENT RECEIVED_INV_TO_OWNER,{NONE},IMI
EVENT RECEIVED_BCAST_REQ,{SEND_ACK_TO_REQUESTOR},IS

STATE IM
EVENT RECEIVED_ACK,{DECREMENT_ACK_COUNTER},IM
EVENT RECEIVED_LAST_ACK_AND_DATA,{DEALLOCATE_MSHR,SEND_ACK_TO_HOME,UNBLOCK_PROCESSOR,DEQUEUE_REQUEST},M
EVENT RECEIVED_DATA_EXCLUSIVE_AT_REQUESTOR,{COPY_DATA_TO_CACHE},IM
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,SEND_ACK_TO_HOME,UNBLOCK_PROCESSOR,DEQUEUE_REQUEST},M
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR},IM
EVENT RECEIVED_INV_REPL,{NONE},IM
EVENT RECEIVED_INV_TO_OWNER,{NONE},IMI
EVENT RECEIVED_WBACK_TO_L1,{NONE},IM
EVENT RECEIVED_BCAST_REQ,{SEND_ACK_TO_REQUESTOR},IM

STATE S
EVENT WRONG_BANK,{MOVE_CACHE_BLOCK,REPEAT_REQUEST},S
EVENT CANT_ALLOCATE,{ENQUEUE_REQUEST_REPL,TRIGGER_REPLACEMENT},S
EVENT LOAD,{UNBLOCK_PROCESSOR},S
EVENT FETCH,{UNBLOCK_PROCESSOR},S
EVENT STORE,{ALLOCATE_MSHR,SEND_GETX_TO_L2_DHM},IMWA
EVENT RECEIVED_INV_TO_L1,{SEND_ACK_TO_REQUESTOR,DEALLOCATE_L1},I
EVENT RECEIVED_INV_REPL,{DEALLOCATE_L1},I
EVENT L1_REPLACEMENT,{DEQUEUE_REQUEST_REPL,DEALLOCATE_L1},I
EVENT RECEIVED_BCAST_REQ,{SEND_ACK_TO_REQUESTOR,DEALLOCATE_L1},I

STATE E
EVENT WRONG_BANK,{MOVE_CACHE_BLOCK,REPEAT_REQUEST},E
EVENT CANT_ALLOCATE,{ENQUEUE_REQUEST_REPL,TRIGGER_REPLACEMENT},E
EVENT LOAD,{UNBLOCK_PROCESSOR},E
EVENT FETCH,{UNBLOCK_PROCESSOR},E
EVENT STORE,{UNBLOCK_PROCESSOR},M
EVENT RECEIVED_FWD_GETS,{SEND_ACK_TO_REQUESTOR,SEND_DATA_SHARED_FROM_OWNER_TO_REQUESTOR},S
EVENT RECEIVED_FWD_GETX,{SEND_ACK_TO_REQUESTOR,SEND_DATA_EXCLUSIVE_FROM_OWNER_TO_REQUESTOR,DEALLOCATE_L1},I
EVENT L1_REPLACEMENT,{ALLOCATE_MSHR,SEND_PUTX_TO_L2,DEQUEUE_REQUEST_REPL_MSHR,DEALLOCATE_L1},EI
EVENT RECEIVED_BCAST_REQ,{SEND_ACK_TO_REQUESTOR,SEND_DATA_EXCLUSIVE_FROM_OWNER_TO_REQUESTOR,DEALLOCATE_L1},I
EVENT RECEIVED_INV_REPL,{ALLOCATE_MSHR,SEND_PUTS_TO_L2,DEALLOCATE_L1},EI

STATE M
EVENT WRONG_BANK,{MOVE_CACHE_BLOCK,REPEAT_REQUEST},M
EVENT CANT_ALLOCATE,{ENQUEUE_REQUEST_REPL,TRIGGER_REPLACEMENT},M
EVENT LOAD,{UNBLOCK_PROCESSOR},M
EVENT STORE,{UNBLOCK_PROCESSOR},M
EVENT FETCH,{UNBLOCK_PROCESSOR},M
EVENT RECEIVED_FWD_GETS,{SEND_ACK_TO_REQUESTOR,SEND_DATA_SHARED_FROM_OWNER_TO_REQUESTOR},S
EVENT RECEIVED_FWD_GETX,{SEND_ACK_TO_REQUESTOR,SEND_DATA_EXCLUSIVE_FROM_OWNER_TO_REQUESTOR,DEALLOCATE_L1},I
EVENT L1_REPLACEMENT,{SEND_PUTX_TO_L2_DHM,ALLOCATE_MSHR,DEQUEUE_REQUEST_REPL_MSHR,DEALLOCATE_L1},MI
EVENT RECEIVED_BCAST_REQ,{SEND_ACK_TO_REQUESTOR,SEND_DATA_EXCLUSIVE_FROM_OWNER_TO_REQUESTOR,DEALLOCATE_L1},I
EVENT RECEIVED_INV_REPL,{SEND_PUTX_TO_L2,ALLOCATE_MSHR,DEALLOCATE_L1},MI

STATE MI
EVENT RECEIVED_WBACK_TO_L1,{DEALLOCATE_MSHR},I
EVENT L1_REPLACEMENT,{RECYCLE_REQUEST},SI
EVENT RECEIVED_FWD_GETS,{SEND_ACK_TO_REQUESTOR,SEND_DATA_SHARED_FROM_OWNER_TO_REQUESTOR},MI
EVENT RECEIVED_FWD_GETX,{SEND_ACK_TO_REQUESTOR,SEND_DATA_EXCLUSIVE_FROM_OWNER_TO_REQUESTOR},MI
EVENT LOAD,{RECYCLE_REQUEST},MI
EVENT FETCH,{RECYCLE_REQUEST},MI
EVENT STORE,{RECYCLE_REQUEST},MI
EVENT RECEIVED_INV_REPL,{NONE},MI

STATE EI
EVENT RECEIVED_WBACK_TO_L1,{DEALLOCATE_MSHR},I
EVENT L1_REPLACEMENT,{RECYCLE_QUEUED_REQUEST},EI
EVENT RECEIVED_FWD_GETS,{SEND_ACK_TO_REQUESTOR,SEND_DATA_SHARED_FROM_OWNER_TO_REQUESTOR},EI
EVENT RECEIVED_FWD_GETX,{SEND_ACK_TO_REQUESTOR,SEND_DATA_EXCLUSIVE_FROM_OWNER_TO_REQUESTOR},EI
EVENT LOAD,{RECYCLE_REQUEST},EI
EVENT FETCH,{RECYCLE_REQUEST},EI
EVENT STORE,{RECYCLE_REQUEST},EI
EVENT RECEIVED_INV_REPL,{NONE},EI

STATE ISI
```

```
EVENT RECEIVED_DATA_SHARED_AT_REQUESTOR,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,UNBLOCK_PROCESSOR,SEND_ACK_TO_HOME,DEALLOCATE_L1},I
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,UNBLOCK_PROCESSOR,SEND_ACK_TO_HOME,DEALLOCATE_L1},I

STATE IMI
EVENT RECEIVED_ACK,{DECREMENT_ACK_COUNTER},IMI
EVENT RECEIVED_LAST_ACK_AND_DATA,{UNBLOCK_PROCESSOR,SEND_PUTX_TO_L2_DHM,DEQUEUE_REQUEST},MI
EVENT RECEIVED_DATA_EXCLUSIVE_AT_REQUESTOR,{COPY_DATA_TO_CACHE},IMI
EVENT RECEIVED_DATA_EXCLUSIVE_AND_ALL_ACKS,{UNBLOCK_PROCESSOR,SEND_PUTX_TO_L2_DHM,DEQUEUE_REQUEST},MI

DEVICE L2
LIST_STATES I,IP,P,PS,S,C,PI,IR,IPW,IPR,PW,BP,BS

STATE I
EVENT CANT_ALLOCATE,{SET_HOME,ENQUEUE_REQUEST_REPL,TRIGGER_REPLACEMENT},I
EVENT CANT_ALLOCATE_AND_CANT_REPLACE,{RECYCLE_REQUEST_TO_L2},I
EVENT MULTIPLE_REQUEST,{SEND_RETRY},I
EVENT RECEIVED_GETS,{ALLOCATE_MSHR,SEND_REQUEST_TO_L2_BANKS},IR
EVENT RECEIVED_GETX,{ALLOCATE_MSHR,SEND_REQUEST_TO_L2_BANKS},IR
EVENT RECEIVED_GETS_TO_L2_BANKS,{DECREMENT_DHM_ACK_COUNTER,SEND_ACK_TO_REQ_TILE},I
EVENT RECEIVED_GETX_TO_L2_BANKS,{DECREMENT_DHM_ACK_COUNTER,SEND_ACK_TO_REQ_TILE},I

STATE IR
EVENT RECEIVED_ACK,{DECREMENT_DHM_ACK_COUNTER},IR
EVENT RECEIVED_LAST_ACK,{SEND_DHM_ACK_TO_L1_COND,SEND_GET_TO_MC_DHM},I
EVENT HOME_BANK,{ALLOCATE_L2,SEND_DHM_ACK_TO_L1,SAVE_OWNER},IPW
EVENT HOME_TILE,{ALLOCATE_L2,SEND_DHM_ACK_TO_L1,SAVE_OWNER},IP
EVENT RETRY,{SEND_REQUEST_TO_L2_BANKS},IR

STATE IPW
EVENT RECEIVED_GETS,{ENQUEUE_REQUEST},IP
EVENT RECEIVED_GETX,{ENQUEUE_REQUEST},IP
EVENT RECEIVED_ACK,{DECREMENT_DHM_ACK_COUNTER},IPW
EVENT RECEIVED_LAST_ACK,{SEND_DHM_ACK_TO_L1},IPR
EVENT RETRY,{NONE},IPR
EVENT RECEIVED_GETX_TO_L2_BANKS,{ENQUEUE_REQUEST},IPW
EVENT RECEIVED_GETS_TO_L2_BANKS,{ENQUEUE_REQUEST},IPW
EVENT L2_REPLACEMENT,{RECYCLE_REQUEST},IPW
EVENT RECEIVED_DATA,{SET_HOME,UPDATE_L2,SEND_DATA_EXCLUSIVE_TO_REQUESTOR},PW


STATE IPR
EVENT RECEIVED_GETX_TO_L2_BANKS,{ENQUEUE_REQUEST},IPR
EVENT RECEIVED_GETS_TO_L2_BANKS,{ENQUEUE_REQUEST},IPR
EVENT L2_REPLACEMENT,{RECYCLE_REQUEST},IPR
EVENT RECEIVED_DATA,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,SEND_DATA_EXCLUSIVE_TO_REQUESTOR},BP


STATE IP
EVENT RECEIVED_GETS,{ENQUEUE_REQUEST},IP
EVENT RECEIVED_GETX,{ENQUEUE_REQUEST},IP
EVENT RECEIVED_GETX_TO_L2_BANKS,{ENQUEUE_REQUEST},IP
EVENT RECEIVED_GETS_TO_L2_BANKS,{ENQUEUE_REQUEST},IP
EVENT L2_REPLACEMENT,{RECYCLE_REQUEST},IP
EVENT RECEIVED_DATA,{DEALLOCATE_MSHR,COPY_DATA_TO_CACHE,SEND_DATA_EXCLUSIVE_TO_REQUESTOR,DEQUEUE_REQUEST},BP

STATE PW
EVENT RECEIVED_ACK,{DECREMENT_DHM_ACK_COUNTER},PW
EVENT RECEIVED_LAST_ACK,{DEALLOCATE_MSHR,SEND_DHM_ACK_TO_L1,},BP
EVENT RETRY,{REPEAT_TILE_REQUEST},BP
EVENT RECEIVED_GETX_TO_L2_BANKS,{ENQUEUE_REQUEST},PW
EVENT RECEIVED_GETS_TO_L2_BANKS,{ENQUEUE_REQUEST},PW
EVENT L2_REPLACEMENT,{RECYCLE_REQUEST},PW


STATE P
EVENT RECEIVED_GETX,{SEND_DHM_ACK_TO_L1,ADD_MISS_TO_STATS,BROADCAST_REQUEST},BP
EVENT RECEIVED_GETS,{SEND_DHM_ACK_TO_L1,ADD_MISS_TO_STATS,BROADCAST_REQUEST},BS
EVENT RECEIVED_GETX_TO_L2_BANKS,{BROADCAST_REQUEST,DECREMENT_DHM_ACK_COUNTER,SEND_ACK_TO_REQ_TILE_HIT},BP
EVENT RECEIVED_GETS_TO_L2_BANKS,{BROADCAST_REQUEST,DECREMENT_DHM_ACK_COUNTER,SEND_ACK_TO_REQ_TILE_HIT},BS
EVENT RECEIVED_PUTX_TO_L2_H,{COPY_DATA_TO_CACHE,SEND_WBACK_TO_REQUESTOR},C
EVENT L2_REPLACEMENT,{ALLOCATE_MSHR,COPY_DATA_TO_CACHE,BROADCAST_INV_REPL,DEQUEUE_REQUEST_REPL,DEALLOCATE_L2},PI

STATE S
EVENT RECEIVED_GETS,{SEND_DHM_ACK_TO_L1,SEND_DATA_SHARED_TO_REQUESTOR},BS
EVENT RECEIVED_GETX,{SEND_DHM_ACK_TO_L1,SEND_DATA_EXCLUSIVE_TO_REQUESTOR,BROADCAST_INV},BP
EVENT RECEIVED_GETX_TO_L2_BANKS,{SEND_DATA_EXCLUSIVE_TO_REQUESTOR,BROADCAST_INV,DECREMENT_DHM_ACK_COUNTER,SEND_ACK_TO_REQ_TILE_HIT},BP
EVENT RECEIVED_GETS_TO_L2_BANKS,{SEND_DATA_SHARED_TO_REQUESTOR,DECREMENT_DHM_ACK_COUNTER,SEND_ACK_TO_REQ_TILE_HIT},BS
EVENT L2_REPLACEMENT,{BROADCAST_INV_REPL,CONDITIONAL_SEND_DATA_TO_MC,DEQUEUE_REQUEST_REPL,DEALLOCATE_L2},I

STATE C
```

```
EVENT RECEIVED_GETX,{SEND_DHM_ACK_TO_L1,SEND_DATA_EXCLUSIVE_TO_REQUESTOR},BP
EVENT RECEIVED_GETS,{SEND_DHM_ACK_TO_L1,SEND_DATA_EXCLUSIVE_TO_REQUESTOR},BP
EVENT RECEIVED_GETX_TO_L2_BANKS,{SEND_DATA_EXCLUSIVE_TO_REQUESTOR,DECREMENT_DHM_ACK_COUNTER,SEND_ACK_TO_REQ_TILE_HIT},BP
EVENT RECEIVED_GETS_TO_L2_BANKS,{SEND_DATA_EXCLUSIVE_TO_REQUESTOR,DECREMENT_DHM_ACK_COUNTER,SEND_ACK_TO_REQ_TILE_HIT},BP
EVENT L2_REPLACEMENT,{CONDITIONAL_SEND_DATA_TO_MC,DEQUEUE_REQUEST_REPL,DEALLOCATE_L2},I

STATE PI
EVENT RECEIVED_PUTX_TO_L2_H,{DEALLOCATE_MSHR,SEND_DATA_TO_MC,SEND_WBACK_TO_REQUESTOR},I
EVENT L2_REPLACEMENT,{REPEAT_REQUEST_REPL},PI
EVENT RECEIVED_GETS,{ENQUEUE_REQUEST},PI
EVENT RECEIVED_GETX,{ENQUEUE_REQUEST},PI

STATE BP
EVENT RECEIVED_ACK_TO_L2,{DEQUEUE_REQUEST},P
EVENT L2_REPLACEMENT,{RECYCLE_REQUEST},BP
EVENT RECEIVED_GETS,{ENQUEUE_REQUEST},BP
EVENT RECEIVED_GETX,{ENQUEUE_REQUEST},BP
EVENT RECEIVED_PUTX_TO_L2_H,{SEND_WBACK_TO_REQUESTOR},BP
EVENT RECEIVED_GETS_TO_L2_BANKS,{RECYCLE_REQUEST_TO_L2_DHM},BP
EVENT RECEIVED_GETX_TO_L2_BANKS,{RECYCLE_REQUEST_TO_L2_DHM},BP

STATE BS
EVENT RECEIVED_ACK_TO_L2,{DEQUEUE_REQUEST},S
EVENT L2_REPLACEMENT,{RECYCLE_REQUEST},BS
EVENT RECEIVED_GETS,{ENQUEUE_REQUEST},BS
EVENT RECEIVED_GETX,{ENQUEUE_REQUEST},BS
EVENT RECEIVED_PUTX_TO_L2_H,{SEND_WBACK_TO_REQUESTOR},BS
EVENT RECEIVED_GETS_TO_L2_BANKS,{RECYCLE_REQUEST_TO_L2_DHM},BS
EVENT RECEIVED_GETX_TO_L2_BANKS,{RECYCLE_REQUEST_TO_L2_DHM},BS
```

Appendix B

Implementation of the Target CMP in an FPGA Board

A CMP system including the proposals presented in this thesis is being implemented in an FPGA board, to demonstrate the feasibility of the proposals and to evaluate their overhead and performance using a real system in addition to simulation tools.

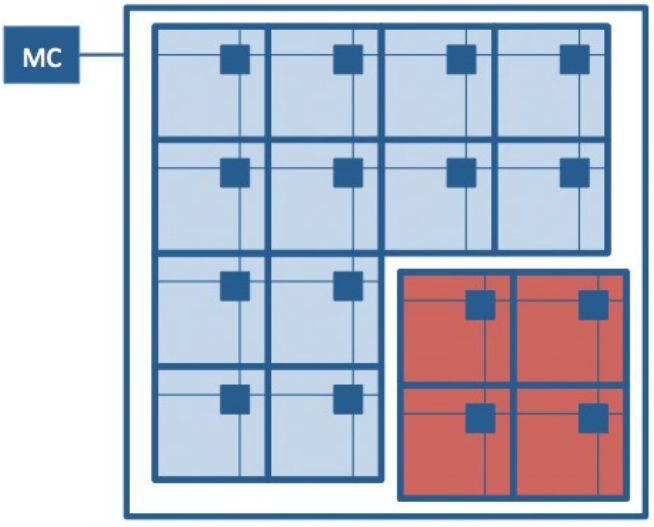


FIGURE B.1: Target system.

Figure B.1 shows the target system: the goal is to implement a 4×4 CMP, with the tiles organized in a 2D mesh. Each tile has the structure assumed throughout this thesis, with one core, one level of private cache (partitioned in instruction and data cache), a bank of shared L2 cache and three switches building three networks (as opposed to one network with virtual channels used in the simulation tool). A single memory controller is used, connected to a corner of the CMP.

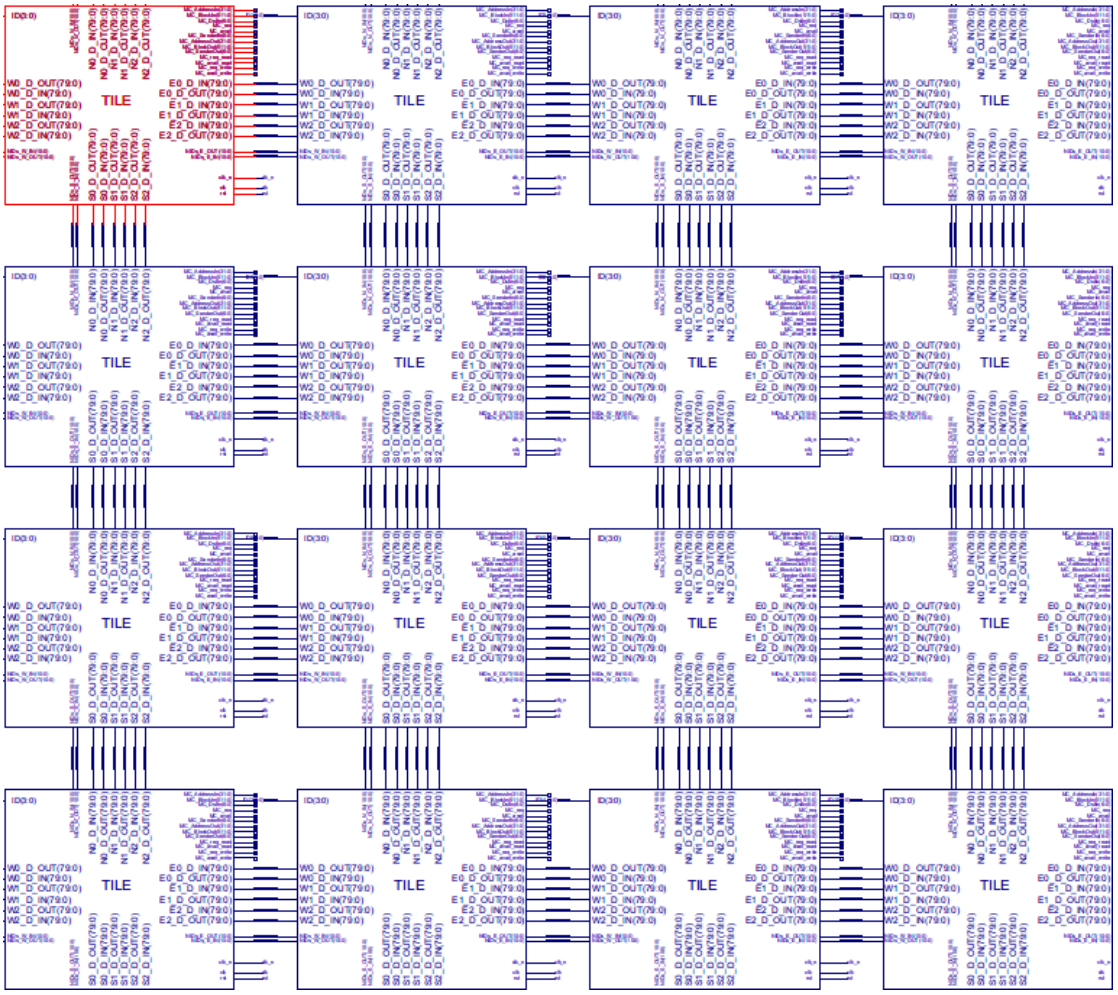


FIGURE B.2: Tiled CMP overview.

A top-down approach was followed during the implementation, defining first the general modules (Figure B.2 shows the global system as specified in the XilinX ISE) and then detailing each module.

Figure B.3 takes a closer look to a single tile. The system has several parameters that can be configured through tile registers located in the TILEREG module; these registers are used, for instance, to store the LBDR configuration bits, to enable/disable the Gather Network and Runtime Home Mapping and to define the system partitioning in case virtualization is used. The MIPSCORE module implements a MIPS-based processor, its private instruction cache and the logic to access the data cache hierarchy through four access types: *load*, *store*, *load linked* and *store conditional*. The latter two are used to implement synchronization primitives.

The private L1 cache and the shared L2 cache bank have a similar structure (Figure B.4 shows the structure of an L1 cache module): incoming requests received from the network interface or, just for L1 caches, from the core, are stored in input buffers; an additional buffer is used to enqueue replacement requests. One of the buffered requests

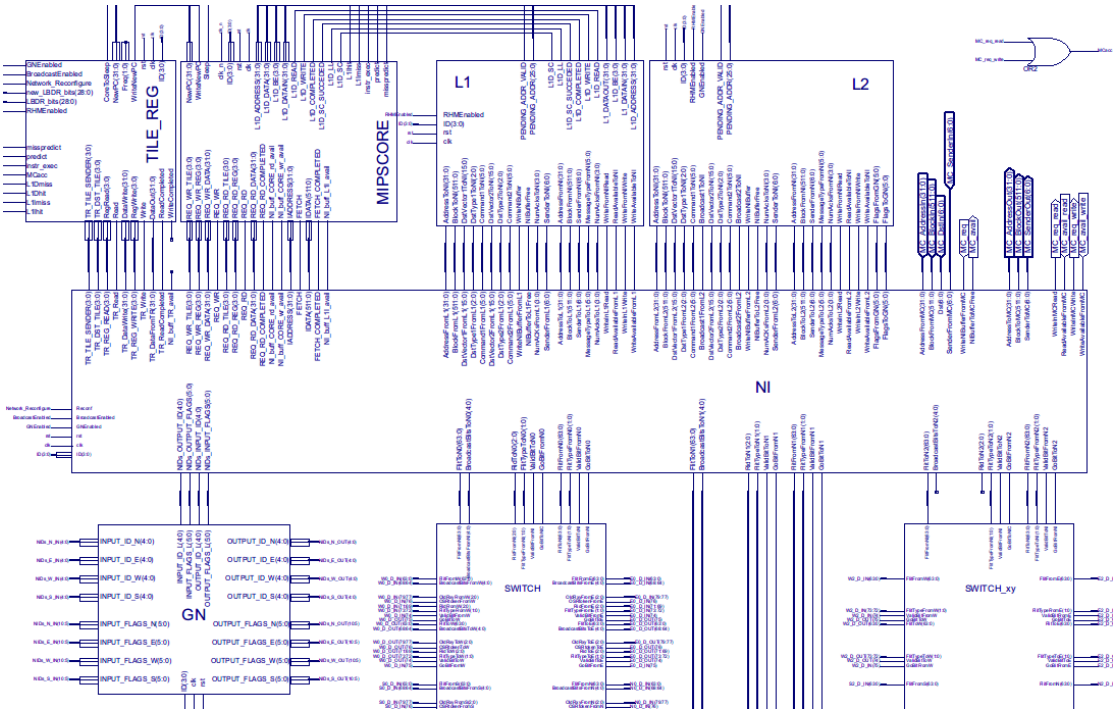


FIGURE B.3: Structure of a tile.

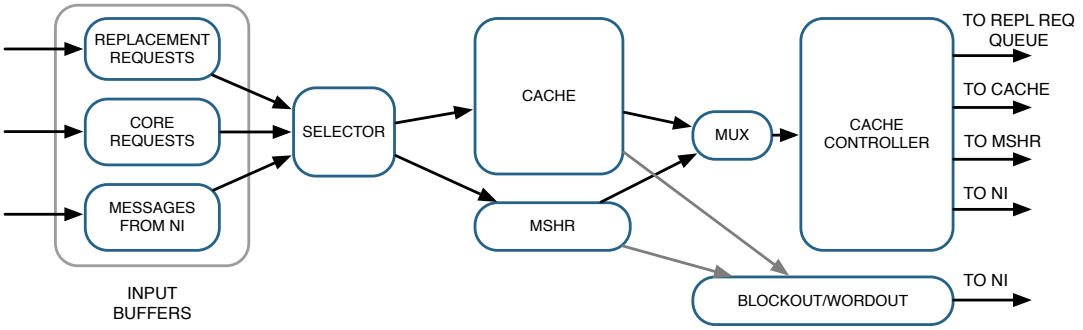


FIGURE B.4: Structure of a cache module.

is selected with a round-robin priority by the SELECTOR logic and propagated to the CACHE and MSHR modules. The CACHE module includes the tag and data arrays and, in the L2 bank, the directory. The MSHR module provides a limited number of entries with the same structure of a cache line, used to store information about the cache line during transient states. The output of these two modules is filtered by a multiplexer (if a valid entry is found in the MSHR, that information must be used rather than the one stored in the CACHE) and sent to the cache controller (CACHE_CC). This module implements the cache coherence protocol: each time the SELECTOR fetches a new request and the state of the requested block is provided by the CACHE and MSHR modules, it is in charge to manage the signaling to perform one or more of the following tasks:

- compose messages to be sent through the NoC. For each message, block address,

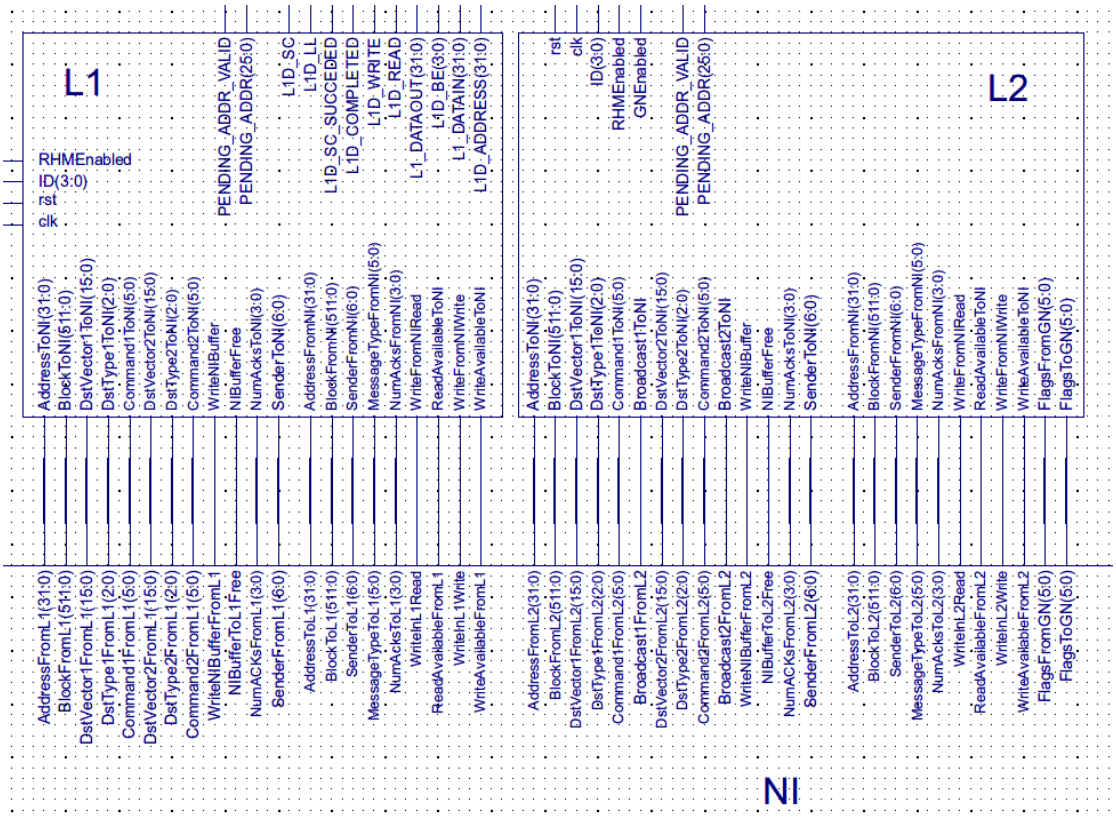


FIGURE B.5: Caches/NI interface.

message size, sender, destination and type are mandatory; additional control fields are used to manage the hardware support to broadcast messages and the Gather Network signaling. To optimize the case of multicast messages (e.g. invalidation messages), destination nodes are specified using a vector of 16 bits, which are set if the message is sent to the corresponding tile, and a destination type (L1/L2/MC). The wiring which connects a cache module and the network interface (Figure B.5)allows the cache controller to send two messages at a time, which is almost always sufficient for the cache coherence protocols which have been implemented. The only exceptions are a few state transitions in RHM, in which three messages must be sent at a time; in this case, two messages are sent first, and then the cache controller freezes until the NI buffer is free; then, the third message is sent and a new request can be fetched by the SELECTOR module.

- allocate or deallocate an entry in the CACHE/MSHR.
- change the block state, set the dirty bit int the CACHE/MSHR tags, update the sharing code in the L2 CACHE tags, update the ACK counter in the MSHR.
- in case a load/store operation ha been completed, notify the core.
- in case a replacement must be triggered, enqueue a replacement request in the input buffer.

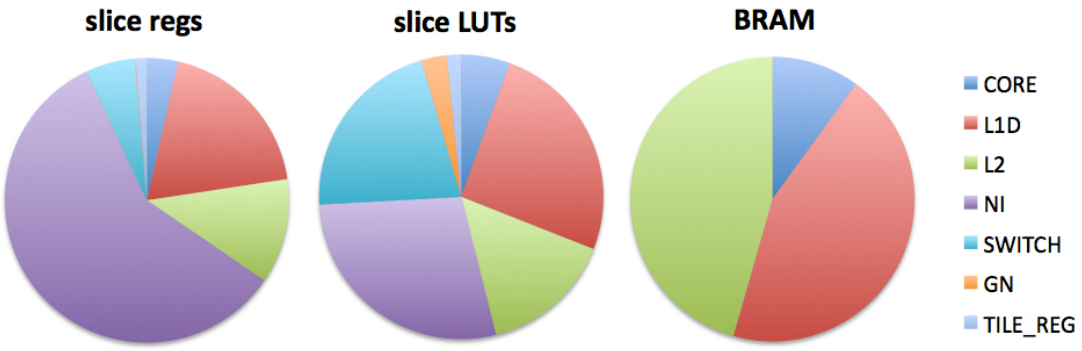


FIGURE B.6: Breakdown of FPGA resources required by a tile.

- notify the SELECTOR that a new request can be fetched from the buffers.

To conclude the description of an L1/L2, the BLOCKOUT/WORDOUT module provides the data block which has to be sent to the NI or the data word which has to be sent to the processor, selecting from three sources (incoming message from the NI, cache line, MSHR line) depending on what's established by the cache controller. A directory-based protocol is used to keep coherence, with MESI states at L1 caches. Blocks are mapped to L2 banks using static mapping or RHM depending on the value of a configuration register.

The remaining modules of Figure B.2 implement the network interface, the switches and the Gather Network. The network interface (NI module) is directly connected to the core, to the tile registers, to the L1 data cache, to the L2 cache bank and, optionally, to the memory controller. The core and the memory controller can send one message at a time (the correspondent buffer at the NI module has one slot), while, as described above, L1 data caches and L2 banks can send two. Buffered messages are transmitted through different networks or through the Gather Network depending on their type. Three switches are included at each tile, one for each network. LBDR configuration bits, stored in the tile registers, encode the routing algorithm (SR [64]).

The Gather Network module uses IDs of four bits and six flags per ID; at the moment, two of this flags are used to encode the HIT and RETRY signals needed by RHM and the remaining are left for future use.

Currently, the tile design is almost complete and an extensive test phase of the system components and the cache coherence protocol is ongoing. Also, each module is being optimized to reduce its resource occupancy. Figure B.6 shows the resource requirements of the top modules used to build a tile. As can be seen, more than 50% of the registers and more than 25% of the LUTs are used to implement the NI. Other two modules which require many resources are the L1 data cache and the L2 bank, which together use more than 25% of the registers, more than 33% of the LUTs and 90% of the BRAM;

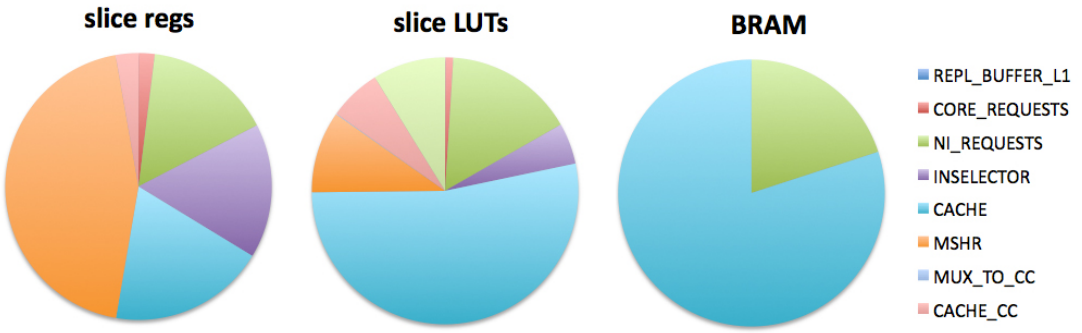


FIGURE B.7: Breakdown of FPGA resources required by L1 data cache.

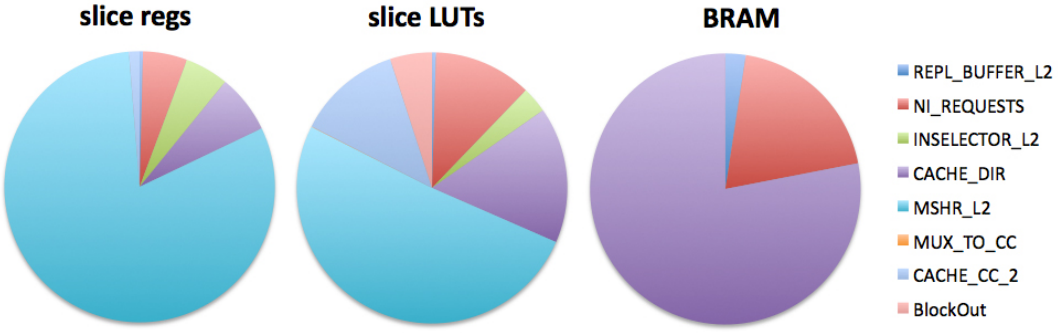


FIGURE B.8: Breakdown of FPGA resources required by an L2 bank.

Device	regs	LUTs	BRAM
CORE	0.13%	0.55%	0.87%
L1D	0.71%	2.51%	3.88%
L2	0.44%	1.49%	3.98%
NI	2.18%	2.77%	0%
SWITCH	0.07%	0.70%	0%
RT	0.05%	0.16%	0%
GN	0.00%	0.29%	0%
TILE	4.37%	9.39%	8.64%

TABLE B.1: FPGA resource occupancy of a single tile.

the remaining 10% of the BRAM is used for the L1 instruction cache within the CORE module.

Figures B.7 and B.8 shows the breakdown of the resources of an L1 and L2 cache respectively. In both cases we can notice that a considerable percentage of registers and LUTs are used to implement the MSHR; this module has not been optimized yet, and its entries are not mapped to the BRAM currently, so a great number of registers is used to implement MSHR entries (notice that the percentage is much higher in the L2 bank, where the MSHR has an higher number of entries).

Implementation results (Table B.1) show that a single tile uses 4.37% of the total FPGA registers, 8.64% of total BRAM and 9.39% of total LUTs (the resource requirements

of the whole tile is not equal to the sum of single components due to optimizations performed by the design tool when synthesizing the design); current design allows thus to implement a CMP system with 10 tiles at most. Current work aims to reduce each tile's requirements and allow 16 tiles to fit in the FPGA board.

References

[1] L. Benini and G. De Micheli. *Networks on chips: technology and tools*. Academic Press, 2006.

[2] J. Flich and D. Bertozzi. *Designing Network On-Chip Architectures in the Nanoscale Era*. Chapman & Hall, 2010.

[3] J. Rattner. Single-chip cloud computer: An experimental many-core processor from intel labs, available at download.intel.com/pressroom/pdf/rockcreek /sccannouncement.

[4] Tile-gx processors family, available at http://www.tilera.com/products/tile-gx.php.

[5] Mppa programmable manycore, available at http://www.kalray.eu/products/mppa-manycore/mppa-256/.

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

[7] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus ipc: the end of the road for conventional microarchitectures. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 248–259, New York, NY, USA, 2000. ACM. ISBN 1-58113-232-8. doi: 10.1145/339647.339691. URL `http://doi.acm.org/10.1145/339647.339691`.

[8] Jose Flich and Jose Duato. Logic-based distributed routing for nocs. *IEEE Comput. Archit. Lett.*, 7(1):13–16, January 2008. ISSN 1556-6056. doi: 10.1109/L-CA.2007.16. URL `http://dx.doi.org/10.1109/L-CA.2007.16`.

[9] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. *SIGARCH Comput. Archit. News*, 30(5):211–222, October 2002. ISSN 0163-5964. doi: 10.1145/635506.605420. URL `http://doi.acm.org/10.1145/635506.605420`.

[10] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. A nuca substrate for flexible cmp cache sharing. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05,

pages 31–40, New York, NY, USA, 2005. ACM. ISBN 1-59593-167-8. doi: 10.1145/1088149.1088154. URL http://doi.acm.org/10.1145/1088149.1088154.

[11] Bradford M. Beckmann and David A. Wood. Managing wire delay in large chip-multiprocessor caches. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 319–330, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2126-6. doi: 10.1109/MICRO.2004.21. URL http://dx.doi.org/10.1109/MICRO.2004.21.

[12] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979. ISSN 0018-9340. doi: 10.1109/TC.1979.1675439. URL http://dx.doi.org/10.1109/TC.1979.1675439.

[13] Håkan Nilsson and Per Stenström. An adaptive update-based cache coherence protocol for reduction of miss rate and traffic. In *Proceedings of the 6th International PARLE Conference on Parallel Architectures and Languages Europe*, PARLE '94, pages 363–374, London, UK, UK, 1994. Springer-Verlag. ISBN 3-540-58184-7. URL http://dl.acm.org/citation.cfm?id=646423.692095.

[14] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. *SIGARCH Comput. Archit. News*, 14 (2):414–423, May 1986. ISSN 0163-5964. doi: 10.1145/17356.17404. URL http://doi.acm.org/10.1145/17356.17404.

[15] Michael Zhang and Krste Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. *SIGARCH Comput. Archit. News*, 33(2):336–345, May 2005. ISSN 0163-5964. doi: 10.1145/1080695.1069998. URL http://doi.acm.org/10.1145/1080695.1069998.

[16] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, July 2012. ISSN 0001-0782. doi: 10.1145/2209249.2209269. URL http://doi.acm.org/10.1145/2209249.2209269.

[17] J. M. Owen, M.D. Hummel, D.R. Meyer, and J.B. Keller. United states patent: 7069361 - system and method of maintaining coherency in a distributed communication system. June 2006.

[18] Pat Conway and Bill Hughes. The amd opteron northbridge architecture. *IEEE Micro*, 27(2):10–21, March 2007. ISSN 0272-1732. doi: 10.1109/MM.2007.43. URL http://dx.doi.org/10.1109/MM.2007.43.

[19] Sangyeun Cho and Lei Jin. Managing distributed, shared l2 caches through os-level page allocation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 455–468, Washington, DC,

USA, 2006. IEEE Computer Society. ISBN 0-7695-2732-9. doi: 10.1109/MICRO.2006.31. URL http://dx.doi.org/10.1109/MICRO.2006.31.

[20] Alberto Ros and Manuel E. Acacio. Evaluation of low-overhead organizations for the directory in future many-core cmps. In *Proceedings of the 2010 conference on Parallel processing*, Euro-Par 2010, pages 87–97, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-21877-4. URL http://dl.acm.org/citation.cfm?id=2031978.2031992.

[21] Abhishek Das, Matt Schuchhardt, Nikos Hardavellas, Gokhan Memik, and Alok Choudhary. Dynamic directories: a mechanism for reducing on-chip interconnect power in multicores. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, pages 479–484, San Jose, CA, USA, 2012. EDA Consortium. ISBN 978-3-9810801-8-6. URL http://dl.acm.org/citation.cfm?id=2492708.2492829.

[22] Yong Li, Ahmed Abousamra, Rami Melhem, and Alex K. Jones. Compiler-assisted data distribution for chip multiprocessors. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 501–512, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0178-7. doi: 10.1145/1854273.1854335. URL http://doi.acm.org/10.1145/1854273.1854335.

[23] Yuanrui Zhang, Wei Ding, Mahmut Kandemir, Jun Liu, and Ohyoung Jang. A data layout optimization framework for nuca-based multicores. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 489–500, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1053-6. doi: 10.1145/2155620.2155677. URL http://doi.acm.org/10.1145/2155620.2155677.

[24] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive nuca: near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 184–195, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-526-0. doi: 10.1145/1555754.1555779. URL http://doi.acm.org/10.1145/1555754.1555779.

[25] Pierfrancesco Foglia, Cosimo Antonio Prete, Marco Solinas, and Giovanna Monni. Re-nuca: Boosting cmp performance through block replication. In *Proceedings of the 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, DSD '10, pages 199–206, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4171-6. doi: 10.1109/DSD.2010.41. URL http://dx.doi.org/10.1109/DSD.2010.41.

[26] P. Foglia, F. Panicucci, C.A. Prete, and M. Solinas. Analysis of performance dependencies in nuca-based cmp systems. In *Computer Architecture and High Performance Computing, 2009. SBAC-PAD '09. 21st International Symposium on*, pages 49–56, 2009. doi: 10.1109/SBAC-PAD.2009.12.

[27] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in cmps. *SIGARCH Comput. Archit. News*, 33(2):357–368, May 2005. ISSN 0163-5964. doi: 10.1145/1080695.1070001. URL `http://doi.acm.org/10.1145/1080695.1070001`.

[28] J. Merino, V. Puente, and J.-A. Gregorio. Esp-nuca: A low-cost adaptive non-uniform cache architecture. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–10, 2010. doi: 10.1109/HPCA.2010.5416641.

[29] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. Gated-vdd: a circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the 2000 international symposium on Low power electronics and design*, ISLPED '00, pages 90–95, New York, NY, USA, 2000. ACM. ISBN 1-58113-190-9. doi: 10.1145/344166.344526. URL `http://doi.acm.org/10.1145/344166.344526`.

[30] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: simple techniques for reducing leakage power. *SIGARCH Comput. Archit. News*, 30(2):148–157, May 2002. ISSN 0163-5964. doi: 10.1145/545214.545232. URL `http://doi.acm.org/10.1145/545214.545232`.

[31] Nam Sung Kim, Krisztián Flautner, David Blaauw, and Trevor Mudge. Single-vdd and single-vt super-drowsy techniques for low-leakage high-performance instruction caches. In *Proceedings of the 2004 international symposium on Low power electronics and design*, ISLPED '04, pages 54–57, New York, NY, USA, 2004. ACM. ISBN 1-58113-929-2. doi: 10.1145/1013235.1013254. URL `http://doi.acm.org/10.1145/1013235.1013254`.

[32] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. L. Scott. Integrating adaptive on-chip storage structures for reduced dynamic power. Technical report, Rochester, NY, USA, 2002.

[33] Chuanjun Zhang, Frank Vahid, and Walid Najjar. A highly configurable cache for low energy embedded systems. *ACM Trans. Embed. Comput. Syst.*, 4(2):363–387, May 2005. ISSN 1539-9087. doi: 10.1145/1067915.1067921. URL `http://doi.acm.org/10.1145/1067915.1067921`.

[34] Andreas Moshovos, Gokhan Memik, Alok Choudhary, and Babak Falsafi. Jetty: Filtering snoops for reduced energy consumption in smp servers. In *Proceedings*

of the 7th International Symposium on High-Performance Computer Architecture, HPCA '01, pages 85–, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1019-1. URL http://dl.acm.org/citation.cfm?id=580550.876432.

[35] V. Salapura, M. Blumrich, and A. Gara. Design and implementation of the blue gene/p snoop filter. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 5–14, 2008. doi: 10.1109/HPCA.2008.4658623.

[36] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *Proceedings of the 1999 international symposium on Low power electronics and design*, ISLPED '99, pages 273–275, New York, NY, USA, 1999. ACM. ISBN 1-58113-133-X. doi: 10.1145/313817.313948. URL http://doi.acm.org/10.1145/313817.313948.

[37] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. The filter cache: an energy efficient memory structure. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, pages 184–193, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-7977-8. URL http://dl.acm.org/citation.cfm?id=266800.266818.

[38] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. *SIGARCH Comput. Archit. News*, 29(2):240–251, May 2001. ISSN 0163-5964. doi: 10.1145/384285.379268. URL http://doi.acm.org/10.1145/384285.379268.

[39] Jaume Abella, Antonio González, Xavier Vera, and Michael F. P. O'Boyle. Iatac: a smart predictor to turn-off l2 cache lines. *ACM Trans. Archit. Code Optim.*, 2(1):55–77, March 2005. ISSN 1544-3566. doi: 10.1145/1061267.1061271. URL http://doi.acm.org/10.1145/1061267.1061271.

[40] L. Li, Ismail Kadayif, Yuh-Fang Tsai, Narayanan Vijaykrishnan, Mahmut T. Kandemir, Mary Jane Irwin, and Anand Sivasubramaniam. Leakage energy management in cache hierarchies. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 131–140, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1620-3. URL http://dl.acm.org/citation.cfm?id=645989.674306.

[41] Jose Flich and Davide Bertozzi. *Designing Network On-Chip Architectures in the Nanoscale Era*. Chapman & Hall/CRC, 2010. ISBN 1439837104, 9781439837108.

[42] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. ISBN 0122007514.

[43] Jose Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks: An Engineering Approach*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1st edition, 1997. ISBN 0818678003.

[44] William J. Dally. Virtual-channel flow control. *SIGARCH Comput. Archit. News*, 18(2):60–68, May 1990. ISSN 0163-5964. doi: 10.1145/325096.325115. URL `http://doi.acm.org/10.1145/325096.325115`.

[45] S. Rodrigo, J. Duato J. Flich, and M. Hummel. Efficient unicast and multicast support for cmps. In *In Proc. of the 41st IEEE/ACM Intl. Symp. on Microarchitecture*, pages 364 – 375, December 2008.

[46] Liqun Cheng, Naveen Muralimanohar, Karthik Ramani, Rajeev Balasubramonian, and John B. Carter. Interconnect-aware coherence protocols for chip multiprocessors. *SIGARCH Comput. Archit. News*, 34(2):339–351, May 2006. ISSN 0163-5964. doi: 10.1145/1150019.1136515. URL `http://doi.acm.org/10.1145/1150019.1136515`.

[47] A. Flores, J.L. Aragon, and M.E. Acacio. Heterogeneous interconnects for energy-efficient message management in cmps. *Computers, IEEE Transactions on*, 59(1): 16–28, 2010. ISSN 0018-9340. doi: 10.1109/TC.2009.129.

[48] Evgeny Bolotin, Zvika Guz, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. The power of priority: Noc based distributed cache coherency. In *Proceedings of the First International Symposium on Networks-on-Chip*, NOCS '07, pages 117–126, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2773-6. doi: 10.1109/NOCS.2007.42. URL `http://dx.doi.org/10.1109/NOCS.2007.42`.

[49] I. Walter, I. Cidon, and A. Kolodny. Benoc: A bus-enhanced network on-chip for a power efficient cmp. *Computer Architecture Letters*, 7(2):61–64, 2008. ISSN 1556-6056. doi: 10.1109/L-CA.2008.11.

[50] Dana Vantrease, Mikko H. Lipasti, and Nathan Binkert. Atomic coherence: Leveraging nanophotonics to build race-free cache coherence protocols. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 132–143, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-4244-9432-3. URL `http://dl.acm.org/citation.cfm?id=2014698.2014902`.

[51] Noel Eisley, Li-Shiuan Peh, and Li Shang. In-network cache coherence. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 321–332, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2732-9. doi: 10.1109/MICRO.2006.27. URL `http://dx.doi.org/10.1109/MICRO.2006.27`.

[52] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. Improving multiprocessor performance with coarse-grain coherence tracking. *SIGARCH Comput. Archit. News*, 33(2):246–257, May 2005. ISSN 0163-5964. doi: 10.1145/1080695.1069991. URL `http://doi.acm.org/10.1145/1080695.1069991`.

[53] Andreas Moshovos. Regionscout: Exploiting coarse grain sharing in snoop-based coherence. *SIGARCH Comput. Archit. News*, 33(2):234–245, May 2005. ISSN 0163-5964. doi: 10.1145/1080695.1069990. URL `http://doi.acm.org/10.1145/1080695.1069990`.

[54] Samuel Rodrigo, Jose Flich, Jose Duato, and Mark Hummel. Efficient unicast and multicast support for cmps. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 364–375, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-1-4244-2836-6. doi: 10.1109/MICRO.2008.4771805. URL `http://dx.doi.org/10.1109/MICRO.2008.4771805`.

[55] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *The 16th IEEE Intl. Symp. on High-Performance Computer Architecture*, 2010.

[56] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, November 2011.

[57] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, 2010. doi: 10.1109/HPCA.2010.5416636.

[58] Cacti 5 technical report, available at http://www.hpl.hp.com/techreports/2008/hpl-2008-20.html.

[59] A.B. Kahng, B. Li, l. Peh, and K. Samadi. Orion 2.0: A power-area simulator for interconnection networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(1):191–196, 2012.

[60] E. S. Shin, V.J.III Mooney, and G.F. Rileyn. Round-robin arbiter design and generations. In *International Symposium on System Synthesis*, pages 243–2485, 2002.

[61] The nangate open cell library,45nm freepdk,available at https://www.si2.org/openeda.si2.org/projects/nangatelib/.

[62] M. Lodde and J. Flich. A lightweight network of ids to quickly deliver simple control messages. In *Proc. of the 2nd Intl, Workshop on On-chip memory hierarchies and interconnects: organization, management and implementation*, August 2013.

[63] "intel core i7 technical specifications", available at http://www.intel.com/products/processor/corei7ee/specifications.htm.

[64] A. Mejia, J. Flich, J. Duato, Sven-Arne Reinemo, and Tor Skeie. Segment-based routing: An efficient fault-tolerant routing algorithm for meshes and tori. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 105–105, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 1-4244-0054-6. URL `http://dl.acm.org/citation.cfm?id=1898953.1899038`.