POLITECNICO DI MILANO
Master in Computer Science and Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria

Exploring Future DNUCA Architectures by Bridging the Application Behaviour and the Coherence Protocol Support

Supervisor: Prof. William FORNACIARI
Assistant Supervisor: Dr. Davide ZONI

Master Thesis of:
Mauro BELLUSCHI
Student n. 819475

Academic Year 2015-2016

Estratto in Lingua Italiana

La rivoluzione portata dai multi-core evidenzia come l'ultimo livello di cache (LLC) sia un componente chiave che influenza l'intero sistema. Una gestione efficace dei dati, deve manterli vicini ai core, riducendo al minimo i costosi accessi in memoria. Inoltre, il numero crescente di core insieme all'introduzione delle NoC come interconnect standard rende le architetture NUCA (Non Uniform Cache Access) una soluzione fondamentale al fine di incrementare le performance e la capacità della cache. L'ultimo livello di cache è solitamente fisicamente distribuito nei vari banchi ed è condiviso tra tutti i core. Perciò, le politiche di mapping e di rimpiazzo dei dati rivestono un ruolo determinante. Tuttavia, lo Static NUCA (SNUCA), soluzione di base che solitamente utilizza un mapping statico per l'ultimo livello di cache, ha dimostrato di avere forti limiti.

Le architetture DNUCA (Dynamic NUCA) consentono un posizionamento adattivo dei blocchi in LLC, offrendo la possibilità di incrementare le prestazioni dell'intero sistema. In tutte le soluzioni DNUCA sono necessari due meccanismi. Il primo è quello di ricerca dei blocchi in LLC, data la possibilità di un blocco di essere posizionato in diversi banchi. Il secondo è il meccanismo che permette di allocare dinamicamente il blocco. Mentre sono state proposte diverse architetture, un'analisi globale degli overhead dei meccanismi non è stata ancora affrontata.

Questa tesi fornisce un'esplorazione e un'analisi completa di tali meccanismi, in relazione anche al comportamento delle applicazioni. I meccanismi sono stati valutati sia dal punto di vista delle performance, sia da quello del traffico aggiuntivo introdotto nella NoC. Inoltre, un'architettura DNUCA provvista di supporto per la migrazione dei blocchi è stata sviluppata e analizzata. In particolare, il protocollo di coerenza MESI è stato esteso con i meccanismi necessari per un sistema DNUCA. Inoltre, viene presentato un nuovo meccanismo di ricerca in LLC. I risultati ottenuti evidenziano come

l'impatto delle applicazioni sulla gerarchia di cache e l'enorme costo dei meccanismi di ricerca non siano trascurabili e mettano in discussione le stesse architetture DNUCA.

Abstract

The multi-core revolution highlights the Last Level Cache (LLC) as a key component which affects the behaviour of the entire system. An efficient LLC data management keeps data closer to the cores, thus minimizing the expensive memory accesses with a net improvement on the overall system performance.

Besides, the ever increasing core count in the chip coupled with the introduction of the on-chip networks as the standard interconnect makes the Non Uniform Cache Access (NUCA) architectures a viable solution to improve the LLC capacity and performance. Considering huge multi-cores, the LLC is usually physically split in banks that are spread across the chip, but identifies a single shared address space between the cores. Therefore, the data placement and replacement policies play a critical function in the overall system by deciding in which LLC bank each cache line has to be mapped. However, the application phases and their parallel execution point out the limitations of the baseline NUCA solution that usually exploits an LLC static data mapping, i.e. Static NUCA (SNUCA).

The Dynamic NUCA (DNUCA) architectures allow a dynamic and adaptive placement of the LLC data to better fit the constantly changing run-time conditions. The possibility to move a cache line among two LLC banks to eventually reduce the distance between the data and the CPU that uses such data can greatly improve the overall system performance while minimizing the communication requirements. To this extent, the additional mechanisms and policies that allow the data migration and dynamic data placement require a careful design stage not to void the benefits of the DNUCA solution due to their overheads. Despite the variety of the existing DNUCA architectures, two mechanisms are common to all of them. The lookup mechanism retrieves a cache line into the LLC, since the block placement is time-dependant, i.e. the same block has not a fixed home bank in the LLC.

Moreover, a dynamic data placement mechanism allows to optimally place a block depending on the considered objective function. The data placement can be placement-dynamic or fully-dynamic. The placement-dynamic scheme freely maps a data in any LLC bank any time it is not present in the LLC. However, no further moving actions are possible within the LLC banks for the data block once placed. Conversely, a fully-dynamic scheme is placement-dynamic and allows to migrate the data block between different LLC banks.

Several DNUCA architectures have been proposed, while a comprehensive analysis of the overheads due to the additional required mechanisms is still missing. The thesis explores such mechanisms also providing a complete analysis on the relationship between the application behaviour and the DNUCA design stages. The lookup and the migration mechanisms are evaluated considering both the performance and the additional generated traffic viewpoints. Moreover, a complete DNUCA architecture with the migration support has been developed and analysed. In particular, the MESI coherence protocol has been augmented and integrated to work with the DNUCA mechanisms. Moreover, a novel broadcast mechanism is presented as part of the lookup mechanism.

The complete architecture has been implemented into the GEM5 cycle accurate simulator using a representative subset of the SPEC CPU 2006 benchmarks for the analysis. The obtained results highlight the non-negligible impact of the applications to the cache hierarchy as well as the huge overhead introduced by the lookup schemes, thus imposing a careful evaluation of the actual need for a DNUCA architecture.

Acknowledgments

To my parents and my brother, who have always supported and encouraged me during my studies and in life.

To Susanna, for being my reason to smile.

To Luca, Fabio, Andrea and all my university mates; we have been a strong team together in these years.

To Matteo, Alessandro and all my friends, who have always been there for me.

To Davide, for his patient guidance and his priceless teachings.

To Professor Fornaciari and the HIPEAC Research Group, because they show me how exciting can be to do research.

Contents

List of Figures

List of Tables

Acronyms

NoC = Network-on-Chip
LLC = Last Level Cache
MC = Memory Controller
MHSR = Miss Handling Status Register
FSM = Finite State Machine
NUMA = Non Uniform Memory Access
NUCA = Non Uniform Cache Access
SNUCA = Static Non Uniform Cache Access
DNUCA = Dynamic Non Uniform Cache Access
CMP = Chip Multi-Processor
VNET = Virtual Network
HPC = High Performance Computing
RAM = Random Access Memory
SRAM = Static Random Access Memory
DRAM = Dynamic Random Access Memory
SWMR = Single Writer Multiple Reader
IPC = Instructions Per Clock
FIFO = First In First Out

Chapter 1

Introduction

Nowadays, the multi-core is the standard computing architecture to keep the pace with the Moore's Law. This is mainly due to the technology limitations that prevent the increase of single-core performance without an exponential increase of the power consumption. In this scenario, the cache hierarchy is a subsystem of paramount importance, which greatly affects the performance, the power consumption and the footprint of the entire multi-core. In particular, the ability of the cache hierarchy to keep both data and instructions closer to the processing elements that use them and its smaller access time compared to the main memory can significantly boost the overall system performance. On the other hand, the cache footprint can reach 50% of the total chip area in modern multiprocessor [21]. Finally, the cache hierarchy power consumption has been reported to be around 40% of the entire chip [35], thus highlighting the cache subsystem as a critical component in all the multi-core design stages. This thesis investigates current cache architectures for NoC based multi-cores with particular emphasis on the required hardware mechanisms to implement efficient Dynamic Non Uniform Memory Access (DNUCA) architectures. In particular, the broadcast mechanism between Last Level Cache (LLC) banks as well as the data migration support have been evaluated from both the performance and additional NoC traffic viewpoints.

An analysis on a novel cache hierarchy made of three parts is presented. A smart broadcast mechanism is designed to efficiently support DNUCA architectures. An enhanced coherence protocol is provided with LLC to LLC blocks migration support for dynamic line placement. Moreover, a migration policy is designed and evaluated. Last, the application behaviour has been

analysed and exploited to steer the design choices.

The developed and analysed architecture is tailored to general-purpose, tiled-based multi-cores to support the development of the future generation of HPC-like architectures. Moreover, it is totally transparent to the software layer. The proposed analysis can also be combined with enhanced NoC architectures [37, 40] to further improve the overall energy-performance tradeoff.

The rest of this part overviews the computing platforms to which the thesis aims to in Section 1.1. Section 1.2 details the open research aspects and the optimization opportunities for the cache hierarchy of next-generation multi-cores. The goals and contributions of the thesis are provided in Section 1.3. Last, Section 1.4 describes the structure of the whole manuscript.

1.1 Tile-based Multi-Cores and Applications

The multi-core is a commodity due to its versatility, high performance and fine grain control on the chip power envelop. It delivers a flexible architecture where various applications can truly execute in parallel. Thus, the multi-core architectures are exploited in several domains, ranging from embedded to High Performance Computing (HPC) systems. The HPC solutions are designed to support computationally demanding tasks, aiming to capacity computing, i.e., ensuring the best performance while executing the maximum number of applications.

Embedded multi-cores enforce strong low power requirements that influence their design. However, they are still general purpose solutions. On the other side, specific task oriented architectures such as accelerators are emerging to speed up specific tasks in bigger applications. Accelerators have simpler CPUs and non coherent cache hierarchy than HPC and embedded systems architectures.

This thesis targets High Performance Computing, general-purpose, cache coherent multi-cores. These architectures are supposed to execute different kinds of applications, with highly variable requirements and characteristics.

The thesis considers the SPEC CPU2006 [20] benchmark suite for the analysis and the methodology assessment. This set of applications is characterized by integer and floating point single-threaded benchmarks. SPEC can be used to stress the cache hierarchy and the system main memory.

In this work, tiled multi-cores with split Last Level Cache are considered [1]. The chip is designed by replicating the same basic module (the tile) to

reduce the design and validation time. Usually a tile is made of a core, its private caches, an LLC bank and a switch to connect the tile to the rest of the system. In tiled multi-cores, LLC banks are usually shared but physically split among the tiles.

Moreover, an efficient data mapping policy for large, shared LLC can avoid expensive and slow memory accesses, saving power and increasing the overall system performance.

1.2 Problem Overview

The on-chip cache is usually hierarchically organized, with small low-latency caches at the higher level and larger, slower ones at the lower levels. This ensures high on-chip capacity and fast data access.

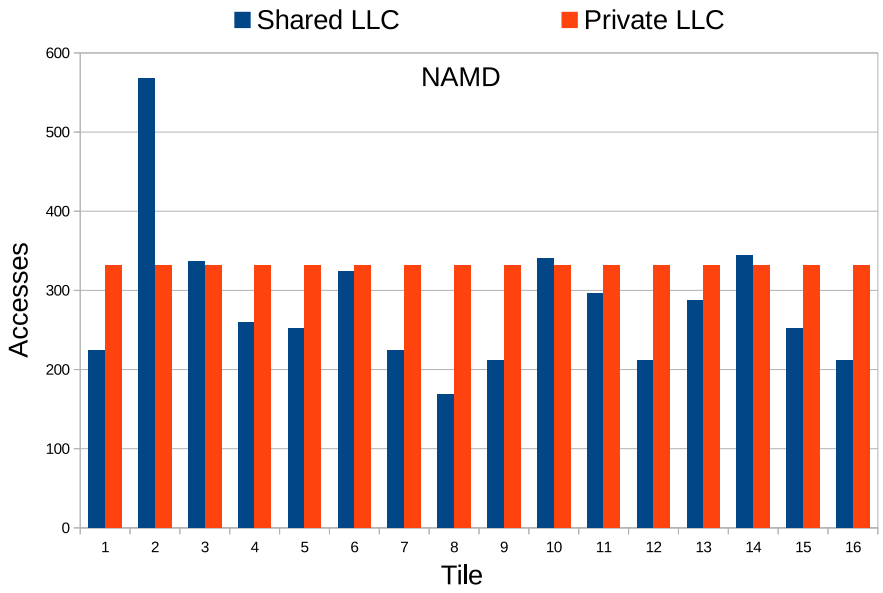
Nowadays, the state of the art proposes Non Uniform Cache Access (NUCA) architectures as promising solutions to access the memory with an optimal cache latency.

Considering a physically split LLC cache hierarchy, the decision to make it shared or private is crucial.

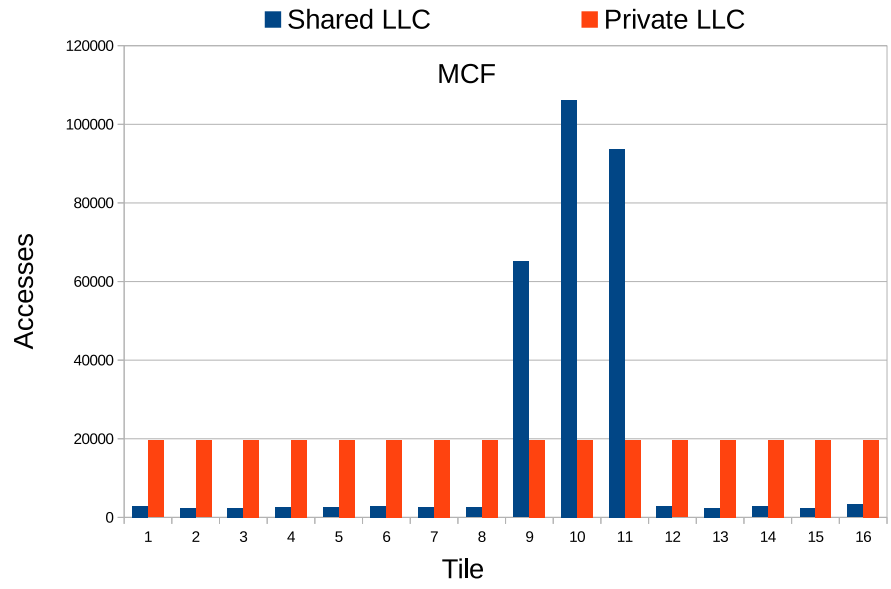
A completely shared last level cache hierarchy ensures a larger space where all the applications can share common data. Traditionally data are statically mapped to the shared LLC by their address. However, it is possible to observe high latencies caused by the distance between the requestor and the accessed LLC bank, since the bank where the data is mapped is not influenced by the position of the requesting core in the topology.

An alternative design makes each LLC bank private to the tile, such as lower cache levels are usually designed. A private LLC hierarchy permits to isolate data of different cores and to have the best performance in terms of access latency. Each core accesses to the LLC inside its own tile, without traversing the interconnect, thus ensuring low latency. However, this approach can be very inefficient due to the fact that applications can be more or less data demanding and if all data do not fit in the tile cache, it is possible to observe performance degradation caused by a high replacements rate. Moreover, private LLCs make the main memory the only synchronization point.

These designs exploit the Static NUCA to statically map the data block using its address. However, a shared SNUCA LLC architecture highlights an unbalanced access distribution between the cache banks.



(a) LLC accesses in NAMD benchmark



(b) LLC accesses in MCF benchmark

Figure 1.1: Distribution among the banks of accesses in LLC cache, considering different mappings with SPEC2006 on a 4x4 Mesh NoC.

Figure 1.1(a) and Figure 1.1(b) show the unbalanced accesses distribution between L2 banks of a SNUCA LLC architecture running NAMD and MCF benchmarks, respectively. A 5 milliseconds time window is used to sample data and a 4x4 mesh NoC based architecture is considered.

Figure 1.2(a) and Figure 1.2(b) show the same behaviour considering few representative MIBENCH benchmarks.

A different but related analysis has carried out to study the access distribution between sets inside the single bank. Results show an unbalanced set access pattern within the same L2 bank. Such a behaviour is observed in private LLC design too: accesses are equally distributed among the banks, but considering the single bank they are unbalanced between sets.

Dynamic NUCA mechanisms emerged to face access latencies problem and to efficiently manage the data allocation in the LLC. These promising solutions could be very useful to fully exploit the cache hierarchy and optimize the performance.

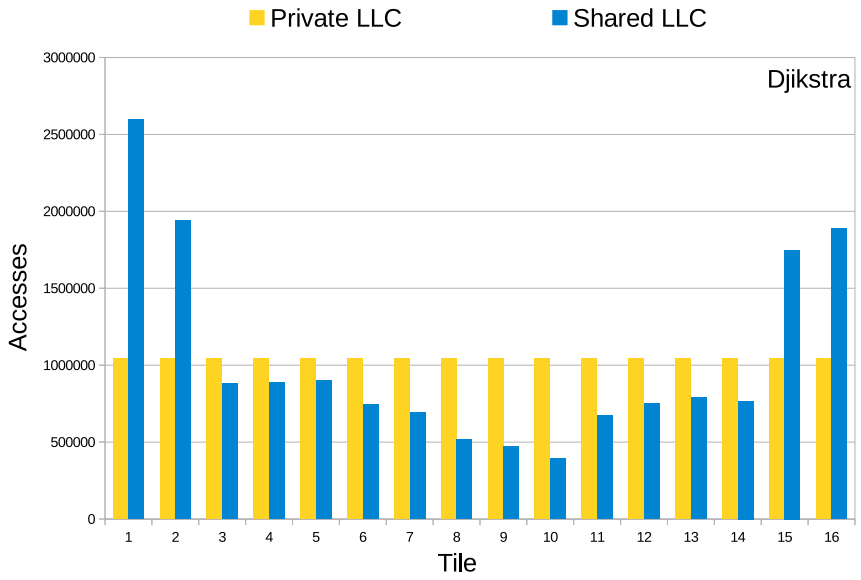
The DNUCA architecture allows to dynamically place blocks in last level cache; the line allocation is time dependent. This can be done in several ways. Some implementations group last-level-cache banks in a certain number of banksets and every block is mapped to a bankset rather than to a single bank. This new kind of approach aims to place data more freely in the last level cache and potentially it is possible to place cache lines everywhere.

A look up mechanism to retrieve the blocks is required due to their dynamic placement in the LLC. Furthermore, some DNUCA solutions implement data migration.

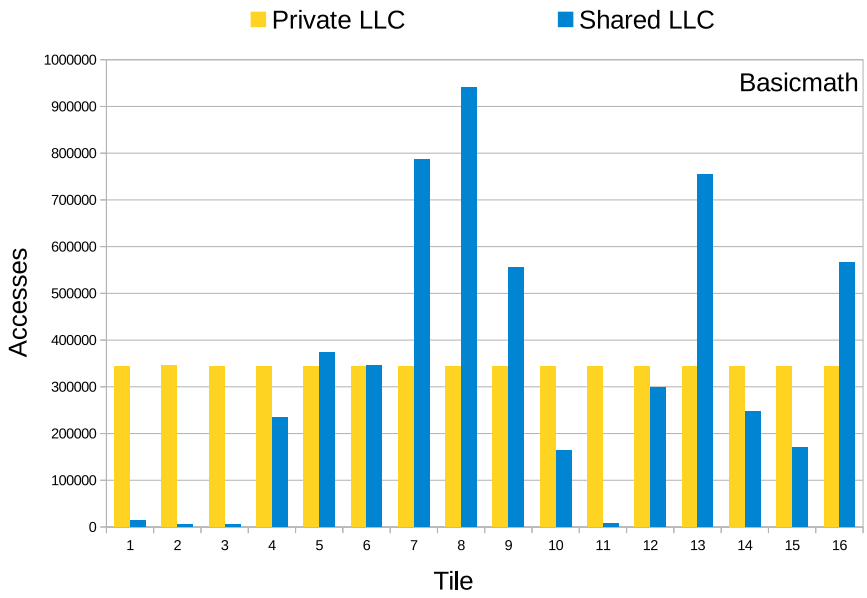
So, before evaluating the possibilities and benefits that can be exploited thanks to new dynamic approaches, these additional mechanisms have to be evaluated. Block searching can be a source of performance degradation. In particular, the block lookup process can be time consuming due to the need to broadcast requests to the whole LLC. Moreover, the degradation grows with the core count and with the system size. Some solutions [25] developed dedicated hardware in order to manage broadcast in LLC.

It is easy to understand that these possible issues and disadvantages that promising NUCA solutions involves can't be ignored. Also irregularity in banks accesses is a source of performance degradation that have to be considered.

Figure 1.3 shows the increase of the traffic volume in the NoC due to the broadcast in an observation interval of 1 millisecond by an architecture which



(a) LLC accesses in Djikstra benchmark



(b) LLC accesses in Basic Math benchmark

Figure 1.2: Distribution among the banks of accesses in LLC cache, considering different mappings with MiBench.

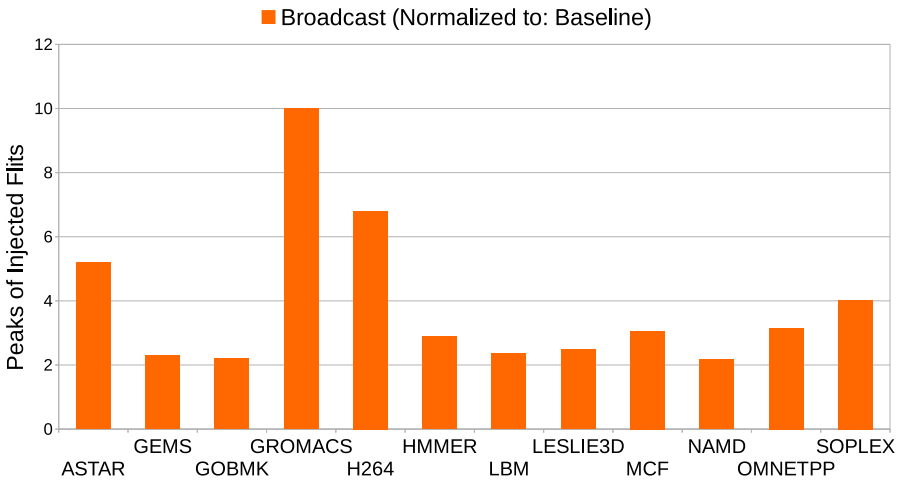


Figure 1.3: Peak of the increment of injected flits in a 4x4 Mesh NoC, due to the broadcast mechanism, in a time interval of 1ms.

uses broadcast and the same without broadcast. Some of the applications of the SPEC CPU2006 suite are analysed and the interval which maximizes the ratio has been chosen on several milliseconds of execution.

A greater number of injected flits has a huge impact on the power consumption of the overall system.

1.3 Goals and Contributions

The thesis provides a comprehensive analysis of the broadcast/multicast and data migration mechanisms in last-level cache to support future DNUCA strategies as also considered in the EU MANGO project [16]. Moreover the application behaviour is also accounted as a key factor in the DNUCA design. Last, a novel DNUCA-capable MESI coherence protocol has been proposed, taking steps from the insights of the carried out analysis.

In a nutshell, the thesis presents the following contributions:

- An analysis of the broadcast/multicast overheads and LLC mapping strategies in DNUCA systems, considering the performance and the additional generated coherence traffic. The application phases and their behaviour are considered as well.

- The baseline MESI protocol is extended with few additional states to

support the data migration mechanism at the LLC level. A simple migration policy has been designed to exploit application characteristics and execution phases.

- A smart data search mechanism is proposed to limit the additional coherence traffic.

A set of multi-core architectures considering 16 cores have been simulated to assess the proposed architecture. Furthermore, synthetic and real applications have been exploited to assess the benefit of the proposed scheme.

1.4 Thesis Structure

The rest of the thesis is organized in 6 chapters. Chapter 2 overviews the background of the work. Chapter 3 describes the state of the art. Chapter 4 provides a detailed description of our architecture and its novel contributions. Chapter 5 details the methodology validation providing results with synthetic traffic and real applications. Chapter 6 points out conclusions and some future works.

Chapter 2

Background

This chapter describes the background of the work, to lease the reading of the following chapters.

2.1 The Network-on-Chip

The NoC is a scalable and reliable interconnect that allows its nodes to exchange data. A node can be both a CPU or a part of the memory subsystem.

The NoC is composed by routers, Network Interface Controllers (NICs) and links. The first ones route data into the network. The second ones allow the communication between a node and a router. Finally, links connect two routers or eventually a router and a NIC.

Traditionally, the NoC splits each message from the memory subsystem in multiple packets. Then each packet is eventually split in multiple flits to better utilize the NoC resources.

The NoC is characterized by a topology, which defines the way routers are interconnected to each others and how memory and CPU blocks are attached to the NoC. The most common topologies used in NoC are mesh [11], concentrated mesh [2], hybrid bus based [12] [33] and high radix [19].

Furthermore, another key aspect of a NoC is its routing algorithm. It defines each source-destination path inside the NoC. Routing algorithms can be deterministic [2] or adaptive [15][13] [14], based on their capacity to alter the path taken for each packet. The most used deterministic routing scheme in 2D-meshes is XY routing; packets first go in the X direction and then in the Y one.

NoCs can implement the VNET mechanism to support coherence protocols and this is done in order to prevent the traffic from a VNET to be routed on a different one, possibly causing dependencies between different kind of coherence messages and generating deadlocks.

2.2 Cache Hierarchy

Historically, the main memory became a bottleneck for computer systems.

The cache hierarchy emerged to fill the gap between the CPU and the main memory performance.

Cache memories exploit temporal and spatial locality. These are characteristics observed statistically in applications, according to which distribution in time and in space of memory accesses is not homogeneous.

Spatial locality states that an application accesses with a greater probability to addresses near the last ones accessed. On the other side temporal locality states that an application accesses more often to addresses accessed recently in time.

The objective of cache memories is to exploit these characteristics in order to keep most recently accessed data and allow CPU fast accesses, avoiding latencies caused by accessing the main memory.

Caches, despite a very low access time, are characterized by high costs in terms of area and power consumption. This is specially true considering costs and access times of the main memory. The causes of these disparities are the different technologies that are used to build the two types of memory. Cache memories are based on Static RAM technology (SRAM). This one uses flip-flops, like a register file, it has non-destructive read-out and it is very fast, but expensive. The main memory is based on Dynamic RAM (DRAM) instead; it uses a single transistor to store each bit, has a simpler structure, allows larger capacity chips but it has destructive read-out, requires regular refresh and is slower.

Thus, the cache hierarchy is usually designed considering these features and aiming to get a trade off between costs and performance. Accordingly, it is organized into several levels. The closer ones to the core have to be smaller and as fast as possible. Farther ones are bigger and have to provide blocks to the higher layers.

The L1 cache, the closest one to the core, is private and is usually split into data cache and instructions one. After that, we find the bigger, a little

bit slower L2 cache; there is no distinction between data and instructions starting from there.

The number of cache levels is going to grow in the future; nowadays is very common to find up to three layers. However, in our description the L2 cache is the Last Level Cache. As explained in Section 1.2, the LLC can be designed private to the core or shared between all the CPUs.

In order to exploit spatial locality, data are retrieved from memory and stored in blocks (also called cache lines), which contain more contiguous words. Traditionally a word can be sized to 32 or 64 bit, a block can be of 64 or 128 bytes, dependently on the processor architecture. When a word is not found in the cache, it must be fetched from the memory and placed in the cache before continuing and so its block is retrieved from memory. Each cache line includes a tag; it identifies which address it corresponds to.

Depending on where a block can be placed, we can distinguish several cache designs. Set associative caches define a set as a group of blocks in the cache itself. A block is mapped onto a set by its address and then the block can be placed anywhere in that set. The set of a line is usually computed as the module of its address. If every set has n blocks, the cache placement is called n-way set associative. The associativity of a set-associative cache is the number of ways in a set and so it's n itself.

Other types of cache designs can be explained starting from the definition of set-associative one. In a direct-mapped cache every block is always mapped to the same location, so it is like it has only one block per set. On the other hand in a fully associative cache a block can be placed anywhere and so the cache has only one huge set.

However, other categorizations are commonly used. Write-through caches update main memory when data are updated in caches. A write-back cache only updates the copy in the cache. When the block is about to be replaced, it is copied back to memory.

One important measure that will be used in the analysis of the next chapters is the miss rate; it is the ratio of the number of accesses which does not find the requested block in cache divided by the total number of accesses.

The various miss types are now described. Compulsory misses are the ones caused by the fact that the first block access can't be successful; lines have to be requested in order to be loaded in cache. Capacity misses are the ones that occur due to the limited capacity of the cache; the block has been previously discarded to free space for another line. Considering not fully

associative caches, conflict misses are caused by the fact that a line may be replaced due to conflicting blocks that map to its set and later retrieved.

Given a block address, it is composed by different parts which have their own meaning and that are used to determine if the line is in cache.

The block offset is usually identified by the least significant bits. It is composed by $n = log2N$ bits, where N is the size of the block. The set index determines in which set the block is and it is composed by the $m = log2M$ higher bits, where M is the number of cache sets. The remaining bits are used for the tag.

The set index is used to determine the cache set. For each block in the set, the associated tag and the one from the memory address are compared. If there is not a match, the line is not in cache. Otherwise, the valid bit is analysed. If it is true, the block is in the cache, otherwise it is not.

If the line at that address is in the cache, then the block offset from that address is used to find the data within the cache block.

If the requested address is not in the cache, then it will be retrieved from memory. As already said, other addresses will be retrieved from memory together with the requested one, in the same block. This is done to exploit spatial locality. The starting address is the one obtained replacing with zeros the block offset part of the address. For the ending address, we replace the block offset with all 1s.

2.3 Basics of Coherence and Coherence Protocols

As previously said, in the tiled multi-cores scenario a shared Last Level Cache is usually used. In this kind of systems, each of the processor cores may read and write to a single shared address space. Before aiming to reach any other key property such as high performance, low power consumption and low cost, for example, we have to provide correctness. Generally this problem can be split in two important sub-issues: consistency and coherence.

Consistency has to define memory correctness; its definitions provide rules about loads and stores and how they act upon memory. It is required that the execution of a thread transforms a given input state into a single well-defined output state and consistency models have to manage multiple threads; they are usually able to reach their scope allowing many correct executions (due

to the fact that the multi-core architecture allows the concurrent execution of multiple threads) and disallowing many (more) incorrect ones. The great number of correct executions makes the job of defining correctness hard.

Cache coherence aims to make the caches of a shared-memory system functionally invisible as is for caches in a single-core system; it is not strictly required, however it helps in providing consistency.

In order to define what coherence is, we can use some invariants. The most used one is the Single-Writer-Multiple-Reader (SWMR) invariant: for any given memory location, at any given moment in time, there is either a single core that may write (and also read) it or a number of cores that may read it. In addition to this invariant, it is required that the value of a memory location is properly and correctly propagated. So we can say that, according the Data-Value Invariant, the value of the memory location at the start of an epoch is the same as the value of the memory location at the end of its last read-write epoch.

There are two ways of managing write requests in cache coherence protocols and so we have two types of protocols: invalidation-based and update-base ones.

In Update-based protocols cores can write on shared blocks. Changes must be propagated to the copies in the L1 caches of the sharers. This can be a promising mechanism because the new block is immediately provided to cores. However the update operation is difficult to implement and maintaining coherence can be difficult too.

Invalidation-based protocols, in order to write on a shared block, invalidate all the shared copies of it in L1 caches. Any successive request for that block will be managed as a new coherence request and the line will be retrieved by lower caches in the hierarchy. Obviously this can lead to a performance degradation caused by the fact that the sharers may still need the block. This means that the core will send a request to the L1, which does not have the line and have to send an additional request to the LLC. Moreover there are even more penalties caused by the invalidation mechanism: an invalidate message is sent to every core and the LLC bank have to wait all the acknowledgements.

Some hybrid solutions exist; however almost all current systems use invalidation-based protocols. This is also assumed in the thesis.

Traditionally, the coherence protocols are implemented as Finite State Machines to ensure the invariants. FSMs represent the evolution of the

block state in base of accesses and coherence actions performed to it. Each state/event pair of the controller triggers a transition, that can lead to another state and can imply some actions.

A coherence protocol is specified by its coherence controllers, i.e. cache controllers.

A request and the successive messages exchanged to satisfy the request are usually defined as a transaction.

The difference between coherence protocols is in the differences between their controllers characteristics. These include different sets of block states, transitions, events and transactions.

A block can be in steady states or in transient ones. A transient state identifies a block for whom an event is waited. Stable states are the ones that are not currently in the mid of a coherence transaction. Steady and transient states depends on the protocol implementation.

However, there are four cache block characteristics that should be encoded in the state: *validity*, *dirtiness*, *exclusivity*, and *ownership*. A *valid* block has the most up-to-date data value. A cache block is *dirty* if its value is the most up-to-date value, which differs from the value in the LLC/memory. Thus the cache controller is in charge to answer to all the requests for the block. A cache block is *exclusive* if it is the only privately cached copy of that block in the system. A cache controller is the *owner* of a block if it is responsible for responding to coherence requests for that block. The block can be dirty.

Accordingly, one of the design choices of a coherence protocol is the number of steady states the block can have. In particular choosing L1 cache states is very important. The properties of each cache block are encoded in order to represent its characteristics, which are the ones described above. Typically the states introduced by [31] are used. Considering L1 caches, the five typical states are M,O,E,S and I. The basic ones are M, S and I. The other ones depend on the specific protocol. States O and E are two optimizations which can be used to extend the MSI protocol, thus obtaining MOSI, MESI and MOESI protocols.

- **M(odified)**: The block is valid, exclusive, owned. It can be dirty and the only valid copy of the block with read-write permissions. The cache must respond to requests for the block. The copy of the block at the LLC/memory is potentially stale.

- **S(hared)**: This cache has a read-only valid copy of the block. Other

caches may have valid, read-only copies of the block.

- **I(nvalid)**: The block is invalid. The cache either does not contain the block or it contains a potentially stale copy that can't be read or written. These two situations can be distinguished as not. The first case can be denoted as Not Present state. We are not going to distinguish these two states.

O and E states are optimizations. In a typical MOESI protocol they have the following meaning:

- **O(wned)**: The cache has a read-only copy of the block and it is valid and owned. It may be dirty. It is not exclusive. The cache must respond to requests for the block. The copy of the block in the LLC/memory may be stale. Other caches may have a read-only copy of the block, but they are not owners.

- **E(xclusive)**: The cache has a read-only copy of the block and it is valid, exclusive, and clean. No other caches have a valid copy of the block. The one in the LLC/memory is up-to-date. There are protocols in which the Exclusive state is not an ownership state. Here, when it is in this state, a block is considered owned by the cache.

Concerning a coherence protocol, it is possible to have different type of messages and different types of events that can interact in the system and which can cause state transitions. There are two possible types of messages: coherence requests and coherence responses. The cache can receive a request in order to obtain write permissions (GetX) or read requests (GetS). By sending responses caches can send data, ACKS or coherence responses in order to manage properly received requests.

The protocols FSMs also include transient states which are required to solve the race conditions due to accesses to the same block by different cores; these additional states are further discussed.

After an L1 cache miss, a request is sent out to a specific node or can be sent to all the caches in the system. Depending on this design choice, the coherence protocol can be a directory protocol (first case) or a snoopy protocol (the second case).

Snoopy protocols usually rely on a shared communication medium (typically a bus) which must have a total ordering of messages. Each cache controllers FSM evolves depending on the block state. All the caches evolve to a

correct state and the protocol is designed to maintain the SWMR invariant. The messages in the interconnect must be totally ordered. All the caches must see the same message order. However this kind of shared interconnect can heavily limit the architecture.

Directory protocol requests are sent to a single node and are managed following the order of their reception at the node. Thus the interconnect is not forced to provide the total message order property. Considering tiled systems, the block is usually mapped in the same LLC bank and so to the same tile. Requests for the block will be sent to that bank, which acts as a home for the block. Thus, it is necessary to rely on a structure to keep track of which cores are using specific blocks. This is the role of the directory, a bit vector associated to each cache block; the size of the vector is equal to the number of cores.

Directory protocols were ideated to overcome the Snoopy protocols limits. However, their scalability is limited too. More cores the system has, more area overhead is going to be introduced by the directory and this has consequences on the power consumption too.

In tiled systems the directory structure is distributed in the LLC cache banks. It is made of sharer vector, the steady and the transient states. The vector of sharers, called sharing code, represents the bigger part of the directory.

Several designs have been proposed to efficiently implement the sharing code. It 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. We refer to this protocols as directory-based protocols.

A solution for the limited scalability can be 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. However, traffic increases, specially not useful traffic (messages sent to nodes which are not sharers).

2.4 LLC Mapping and Non Uniform Cache Access (NUCA) Architectures

Multi-core cache structures were usually designed to have uniform cache access time regardless of the block being accessed. For those architectures, the access time represented a significant bottleneck as the cache became larger, due to the fact that it was sized to the worst access time in the system. Non-Uniform Cache Access (NUCA) architectures split the cache into multiple physical banks, in order to access the memory with an optimal cache latency [23].

This idea has been developed through the years and there are different ways in which NUCA architectures have been managed.

Static NUCA (SNUCA) architectures use static mapping policy to place the blocks into the LLC: lines are mapped to their home bank depending on their address. As already said, in split shared LLCs, the home bank is the one that is in charge to host and manage the block. This policy should evenly distribute blocks on the LLC banks; this is however not true in real applications because memory accesses are never uniformly spread over the memory address space. However, the position of the requesting core is not taken into account by the mapping policy, thus it is possible to experience an high latency due to the distance between home bank and the L1 requestor.

Dynamic NUCA (DNUCA) can dynamically place cache blocks between the banks, thus the same line can be found in different banks over time. This can be implemented dividing banks into banksets and introducing the possibility of placing a block in any bank within a given bankset. However, in these techniques the block has to be searched in the LLC before declaring a cache miss. This implies that a search mechanism has to be implemented in order to find a block. This is a key aspect due to the fact that a multicast or eventually a broadcast can be very performance degrading.

The DNUCA approach was originally proposed for a single-core system and then extended to multi-cores. Ping pong effects and race conditions can make the design hard and the performance poor.

[23] and [6] show search policies are not trivial for NUCA caches. The

possibility of mapping data in more banks often implies the implementation of block migration mechanisms in order to move data dynamically.

In DNUCA designs, many race conditions have to be solved in order to guarantee correctness and prevent deadlocks. For example the *False Miss* problem [17] has to be managed. As a consequence of the migration mechanism, there could be a time interval in which none of the banks involved in the migration process is able to provide the requestor with the referred block, thus resulting in a last-level-cache miss even if the block is actually on chip.

Another common race condition that can occur is the *Multiple Miss* one. When two or more processors simultaneously send a request for the same block and it is not in cache, multiple LLC misses and multiple requests to the main memory are sent for the same block. In this way, multiple copies of the block are retrieved from the memory and this is a problem.

Chapter 3

State of the Art

This chapter summarizes the state of the art considering different aspects and metrics in the LLC design, with particular attention on DNUCA architectures. The SoA on DNUCA architectures is presented in Section 3.1 while Section 3.2 discusses the power consumption management aspects in DNUCA caches.

3.1 Dynamic Non Uniform Cache Access

Shirshendu and Hemangee [29] show a DNUCA solution for Tiled CMP systems. Several specific problems of tiled-based architectures have been considered. T-DNUCA is proposed to optimally place each block as close as possible to its requestor. It splits the LLC in banksets. In case of an L1 cache miss, the requested block can be in any L2-bank (more generally LLC-bank) of the selected bankset. TDNUCA implements a multicast search mechanism within the bankset in case of miss in the Manhattan-Closest Homebank (MCH) to the requestor. The MCH is the closest bank in the bankset to the requestor. Moreover the block is initially placed in this bank when it is fetched from memory. TDNUCA allows blocks migration (in the same bankset) to reduce the access latency due to the distance between the L1 requestor and the LLC destination. Moreover, the migration policy tries to bring heavily used blocks in the MCH. The cascading replacement is an additional supported feature; instead of removing a block from the cache, TDNUCA tries to place it into another peer bank (another bank of the same bankset), using the migration mechanism. This is achieved considering a certain cascading

number. A replacement with a consequent migration can cause another replacement in the peer bank target. The cascading number is the maximum quantity of replacements than can be consequentially caused. The cascaded block policy ensures that a global victim will be removed instead of a local one. TDNUCA improves access latency due to block migration and outperforms Tiled SNUCA, considering this metric. However TDUCA can increase miss rate due to the migrated blocks that can cause LLC replacements. Cascading do not show significant improvements with small size applications or ones with low temporal locality. Differently from our solution blocks can be mapped and migrated only in the specific bankset and the initial mapping is done according the MCH. So the block can't be migrated closer than to the manhattan-closest homebank. Our solution manage differently the cache and is able to place a block everywhere.

[17] presents SNUCA and DNUCA designs and an analysis of performance dependencies in these kind of systems. Different scenarios are presented and some typical DNUCA problems are managed at protocol level. Different configurations of an 8 CPU architecture are considered for the assessment.

Moreover, two important well known DNUCA problems have been managed: the false misses and the multiple misses ones. The FMA (False Miss Avoidance) Protocol guarantees that during a migration at least one of the two banks knows that the block is on chip. This technique allows the banks to exchange messages to manage the eventuality that requests arrive while a block is on-the-fly. On the other hand, Multiple Miss problem is solved by allowing just one node in the system (called the Collector) to communicate with the main memory for a block. There is one for every bankset of the DNUCA cache.

The studies developed in this work highlight how mapping and topology heavily influence performance and application behaviour.

[22] proposes a novel DNUCA design that exploits the thread information to control the block migration.

This work aims to migrate blocks in order to boost the execution of critical threads and to decrease the execution time of applications.

Thread prOgress aware block Migration (TOM) is an epoch-based mechanism. The length of an epoch is a predefined parameter, decided by the user. A TCP (Thread Criticality Prediction) is carried out at the beginning of each epoch to recognize critical threads. Then the threads are ranked based on the collected critical information. These processes are done using

the techniques presented in [7]. TOM periodically collects criticality values in order to update the categorization. Moreover, all cores (one thread per core is supposed) spread their values broadcasting messages to all the cache controllers. After this first phase, whenever a request hit in the cache bank, the corresponding cache controller checks to which group the requesting thread belongs to. If the thread is a leading one and the block is not shared with a critical one, the block is migrated to a central bank-cluster. If the thread is the only critical thread that is accessing the block, it is migrated one step closer to the requestor. Again, if the thread is a critical one and it is not the only one, a saturation counter is used. Last, if it is different from zero the block is migrated, otherwise the block is kept at its position.

Considering 8 cores, the hardware overhead of the TCP is 72 bytes and each core has 64-bit criticality counters and one 64 bit interval bound register is used for all cores. Moreover each cache controller has a vector which have to store the criticality group of each core (N bits per vector, with N equal to the number of the cores).

[28] exploits thread migration mechanism to have useful LLC data closer to the requestors. This is done because remote accesses can result degrading for performance, due to messages that have to traverse the interconnect several times (typically almost two times, supposing a request with a successive response). Fine-grained hardware-level thread migration is proposed. The thread context is saved and is migrated through the interconnect. This work highlights how the mechanism is useful when the migration is followed by a lot of consequent accesses to the block. So it is possible to avoid an elevate number of remote accesses that would affect negatively performance. A predictor is used to decide whether to migrate a thread.

[32] proposes a NoC reconfiguration mechanism to support NoC virtualization in real scenarios. Dynamic network resources reassignment scheme allows to dynamically adapt the NoC to applications needs. This is done exploiting the benefits of another precedent work, the Logic-Based Distribute Routing (LBDR), which allows the NoC partitioning at the routing level. In LBDR a static situation where many different applications are running at the same time has been analyzed. It relies on two bits sets. One defines the connection pattern of the region (one bit per port), the other one defines the allowed turns according the applied routing algorithm (two bits per port).

The first set is actually represented by connectivity bits: each output port has a connectivity bit C_x indicating whether a switch is connected through

the x port; the second set indicates the allowed turns in the next switch. When a new partition is enabled, those bits have to be reviewed and if necessary updated. This is a static reconfiguration: it affects parts of the network in which applications are terminated and new ones are enabled, so no messages are travelling for those applications. These aspects are crucial to avoid deadlocks.

[24] proposes a data search algorithm for DNUCA designs in CMP architectures, called HK-NUCA, to reduce both miss latency and on-chip network contention. The LLC is organized in banksets, where each one logically contains more banks. An L2 bank is part of a single bankset. When a block is fetched from memory, it is mapped on the home bank, which is statically predetermined by the lower bits of its address. However it can be mapped on every bank within the bankset. The use of block migration can thus redistribute blocks on banks which are not the static home bank for that specific block to exploit both spatial and temporal locality.

The data searching mechanism has to keep information about which other banks have at least one of the data blocks that it manages. Thus, all banks have a set of HK-NUCA pointers. Each of them has a single bit for each bank in the bankset; if it is set, that bank is storing at least one of the data blocks which are managed by (which are originally mapped in) the current bank. In order to increase the accuracy, every cache-set in bank is equipped with an HK-PTR.

The proposed solution has different stages. The first one is the fast access one, in which the closest bank is accessed, supposing that migration should have brought blocks near to requestors. The second one is the Call Home one, in which the home bank is accessed and in case of miss, HK-PTRS are used to access other banks. Last, the parallel access stage, using HK-PTRS, in which if the block is still not found the request is sent to the memory. However, this mechanism can be optimized in some specific cases. Obviously pointers has to be updated in case of eviction, migration and fetch from memory. Furthermore HK-NUCA has different stages in order to effectively search in the bankset and reduce the time of the single search. Our work aims to reduce the number of times in which it is required to search in the last-level cache. We have implemented bits in L1 caches (not in LLC) to keep information on migrations and on the effective position of blocks in LLC; moreover our solution provides smart broadcast.

[34] extends HK-NUCA [24]. It duplicates the number of bits dedicated

to HK-PTRS to represent if the block has been migrated or not (moved or none state). Moreover it exploits these information splitting the Parallel Access phase in two new stages. The first one, the Parallel Access to Moved State stage, forwards the request only to caches with the "migration" bit set to 1, (between the ones with the presence bit set). If we have a miss in this phase, the second one, the Parallel Access to None State will access to other banks with presence bit set.

[9] proposes a novel approach in cache hierarchy. It adds a shared L1 cache bank for each core to split accesses between classic private (L1P) and the new shared L1 cache (L1S). L1P cache stores only private and shared read-only data and so actions are performed locally without the need to communicate with other cores: this heavily simplifies coherence, because no action is needed, due to the fact that there is only one copy of the data in every cache level. This permits to not use directories and snooping requests.

Accessing shared data presents extra latencies, due to the fact that part of the L1 is shared and distributed among the cores. It is necessary to transmit data through the interconnect, which should be a low latency one, in order to overcome this drawback.

However, the described mechanism needs an accesses classification at the time of the execution of the instruction and this can be done in several ways. An OS-based classification marks pages as private or shared, according to the core ID which is accessing the page and the one that has accessed it previously.

If the core is not the same, this is a shared access. So, the P/S bit for private/shared information is introduced also in MHSR entries. This kind of classification works at page level and this could be ineffective.

Another used kind of classification is the compiler-assisted one. It inspects the code at compile time and conservatively classifies accesses considering the nature of the target data or of the code regions initiating the access. However classifying data statically poses challenges because of statically unknown events.

The proposed dedicated cache has been evaluated using the cycle-accurate GEMS simulator, using a single level of private cache and two levels of on-chip caches. Results are compared with the ones of a traditional MESI directory-based coherence protocol. Latency due to the dependency on the interconnect and the low accuracy of classification mechanisms represent relevant limitations; however this approach can be useful as the number of cores in-

creases and coherence protocol faces scalability issues. Differently from us, this approach tries to simplify coherence, adding hardware.

[18] proposes a scheme that provides cache sharing by adapting migration, insertion and promotion policies to the dynamic cache access behaviours of the applications. This scheme is based on a previous work, the DSR (Dynamic Spill-Receive) architectures, which provide low-overhead caching between different partitions. Moreover in this work each running application have been classified as "Giver" or "Taker", if the application is less or more cache-demanding. A dynamic migration policy is proposed to migrate a line of a remote partition in the local one according to access ratios. This possibility is managed to guarantee two policies: the no-migration one and the migration one. Nevertheless it is necessary that a configuration remains active for a certain amount of time, in order to avoid continuous policy changes for a core, before converging on a good one. Dynamic insertion and promotion policy is proposed too.

In order to avoid that the streaming applications load a great amount of poorly used lines, causing replacement of more useful lines, the LRU replacement policy has been modified and a promotion policy has been introduced. When a streaming application receives a cache hit, the line is promoted of a certain number of positions in the priority list, instead of becoming the actual MRU. Similarly, when a line is loaded in cache it can be inserted in a particular point of the priority list.

[25] proposes the Runtime Home Mapping (RHM). It is a new dynamic approach where DNUCA is exploited to use all the LLC and to determine the home bank at runtime. This is done with the purpose of mapping blocks as close as possible to the requestor, in order to lower message latencies. Moreover Runtime Home Mapping allows blocks migration and replication mechanisms in the LLC.

In RHM each block can be mapped in all the LLC banks. If the block is not found in the L2 bank target of the request, all the banks have to be accessed to find the requested block. Moreover, the initial L2 mapping is done by the Memory Controller considering an algorithm that maps the blocks as close as possible to the L1 requestors.

In particular, for every L1 miss, a request is sent to the local L2 bank (the one in the same tile). If the block is found, it is delivered to the L1 cache and it means that the L2 bank is the home for that particular block. In case of miss, a broadcast mechanism is used to send a request to all other

L2 banks. In case of hit in a remote bank, a signal is used to notify the L2 bank that triggered the broadcast; moreover the line is sent to the L1 requestor. However, every remote bank has to send back an ACK. When all the ACKs are collected by the L2 bank that triggered the broadcast, the hit signal is checked. If the hit signal has not been received, the block is not in the last level cache and the block has to be retrieved from the memory.

When the MC receives a request for a block, it fetches the line from the main memory and computes the new home for the incoming block; after that it notifies the new L2 home bank so it can start a replacement, if necessary.

In all the other cases, RHM mechanism follows the typical MESI coherence protocol implementation.

The presented home search policy has very high network resources demand due to the triggered broadcasts. In order to manage this problem, the Gather Control Network (GCN) has been designed. It is a purely hardware dedicated solution and it is crucial to make the RHM method effective and efficient. It can be logically seen as 16 one-bit wide subnetworks, one per tile. Each subnetwork is a tree of AND gates, connecting the destination tile with other ones. Broadcast requests from the local LLC bank are sent through the Network On Chip to all other banks. When this request is received, the bank triggers the output signal of the GCN for that tile. The Gather Control Network can be implemented as a very specific combinational block and ACKs are seen as signals and not as messages. This avoids routing, flow control, arbitration in the NoC and all latencies that are related to this mechanisms. GCN has different implementations.

Regarding the Mapping algorithm performed by the Memory Controller, it basically scans the banks starting from the local one in distance order and maps blocks considering the cache utilization statistics. It aims to map the block as close as possible to the requestor, without causing replacements. If all the banks are full this mechanism tries to balance the number of allocations.

The initial mapping algorithm represents a sort of implicit migration of blocks that are replaced. When a block is fetched from memory, it is placed in its new home, which is the tile as close as possible to the requestor. However, a true migration mechanism has been implemented and permits to move blocks in all the last level cache.

After a broadcast request, a migration is triggered when the block is found in a remote bank and, due to the access, one of its directional counters

(one for each of the four directions) exceeds a fixed threshold. Thus, the remote L2 bank notifies the bank which triggered the broadcast using the MIGR signal of the GCN and upon receiving the signal, this one performs an internal copy of the block from the L1 requestor cache. Successively the remote L2 home bank deallocates the block.

Moreover, a LLC block replication mechanism is provided. The mesh is divided into four parts and each of them can have a copy of the block. The multiple copies have to be read-only ones. The home has to remain the same and the replication is permitted only to decrease read accesses latencies. In case of replacement the new home is chosen between these copies.

3.2 The DNUCA Power Perspective

[3] analyses the problem of conflict hits in a specific DNUCA architecture and proposes a power efficient migration mechanism. In the work it is assumed a DNUCA last level cache partitioned in a matrix of independent banks in which blocks are distributed in a particular way. The entire address space is spanned on each column of banks and each bank belonging to a row behaves as a single way of a set-associative cache. A line can be found at the same index in every bank of the same row. The considered block migration is a mechanism that swaps cache lines between banks of the same rows in order to promote one of them. The block promotion technique allows to make lines closer to their requestor. In this scenario, the conflict hit is considered. It occurs when the access pattern cause several migrations between blocks, they are swapped several times and the considered cache lines haven't been effectively promoted. This work proposes a per block bit which is set when the cache line is promoted and then demoted; if the bit is set, a successive promotion will be avoided, solving the conflict hits issue. This bit will be reset when an access to that block occurs in its bank. This solution aims to reduce the additional accesses caused by conflict hits and so, to reduce the energy consumption. Our work is based on a different DNUCA configuration and LLC organization. The considered migration mechanism is different too. In this work the SPEC CPU2000 benchmarks behaviour have been considered and studied; we consider and analyse SPEC CPU2006 applications.

[5] presents the Way Adaptable DNUCA cache. It aims to shut down ways in a set-associative last level cache, to improve DNUCA power efficiency. In this work last level caches partitioned in a matrix of independent banks are

considered. Banks belonging to the same column behave like different ways of a set-associative cache. So, every column is a bankset and most accessed blocks are promoted and migrated closer to the requestor, maintaining the same column. It has been observed that hits in ways are unbalanced and the behaviour changes, varying the considered application and the time phase of the single application.

This is true as a consequence of the promotion and demotion mechanisms. Thus, not all the cache ways are needed during the execution of an application and they can be turned off to achieve a reduction of the cache static power consumption. This can't be done statically by using an architecture with decreased associativity. Accordingly, the behaviours change dependently on the application and on its time phase. The caches with high associativity can easily contain all the working set of the application, but they can be a waste of resources and a source of power consumption. Thus, this work opts for using a large and high-associativity cache and provides a mechanism able to switch off the unused ways dynamically. In order to achieve this purpose, a predictor is needed. The number of hits in farther way and the hits in the closest one are used as metrics. They are compared with defined thresholds to turn off the farthest way, to maintain the current configuration or to turn on the closest powered off way. This mechanisms show not relevant performance degradation in terms of IPC, but a significant decreased average number of used ways and so significant improvement in power efficiency. This work is based on a different LLC organization compared to our solution and SPEC CPU2000 suite is analysed. Differently from this work, in which a single core with a large last level cache is considered, we analysed multi-cores.

[4] is strictly related to the ones presented by [5]. In particular the same DNUCA configuration is considered and the mechanism proposed in [5] is described again. In addition, metrics and thresholds used in the predictor are detailed and motivated. Moreover, some results on multi-cores architectures are shown. The elaborated algorithm to estimate parameters considers an heuristic based on a reconfiguration event. Every K hits in LLC this event is triggered and a metric is evaluated. In order to identify the best reconfiguration sequence for the given workload and from this state, the execution is restarted in three different runs considering the same configuration kept, a way switched on and a way switched off.

At each step the selected reconfiguration event places restrictions on the values of thresholds and at the end a set of inequalities are obtained. If they

are not solvable the set is restricted and the accuracy is reduced. Otherwise thresholds are computed. The metric used to determine the optimal sequence is miss rate However it is possible to use other metrics such as IPC. Results of the works done on multi-cores is also shown. Two cores with their private caches are attached through a bus to the banked last level cache. Several applications of SPEC CPU2000 suite are classified and grouped, in order to run simultaneously benchmarks with variable requirements in terms of utilised ways. Results show how average associativity achieved in the multi-core execution is lower than the sum of the average associativities achieved in the single core execution. This is caused by the promotion mechanism and means that the Way Adaptable technique can be adopted also in multi-cores.

Chapter 4

The DNUCA design: Coherence Protocol Implications

Compared to the SNUCA, the DNUCA infrastructure requires three components to properly manage the LLC data. The broadcast mechanism is responsible for searching data in the LLC, due to the fact that the same block can be placed in different banks at different instants in time. Depending on how the system is designed, this mechanism can be a truly broadcast [25] or a multi-cast [24], [29]. The data migration mechanism is another important feature in order to fully exploit DNUCA architectures. It can be used to place blocks in different banks depending on the objective function. Last, the migration policy contains the logic to trigger both data migration and broadcast to efficiently exploit the LLC cache.

The chapter presents the broadcast and migration mechanisms design and the deadlock avoidance analysis. Moreover the migration policy and the Smart Broadcast are discussed.

The rest of the chapter is organized as follows. Section 4.1 overviews the MESI protocol, that is the baseline coherence protocol in this work. Section 4.2 discusses the broadcast mechanism, its deadlock avoidance analysis and the smart broadcast mechanism. Section 4.3 details the migration mechanism, as an extension to the MESI protocol and its deadlock avoidance analysis. Last, the migration policy is described in Section 4.4.

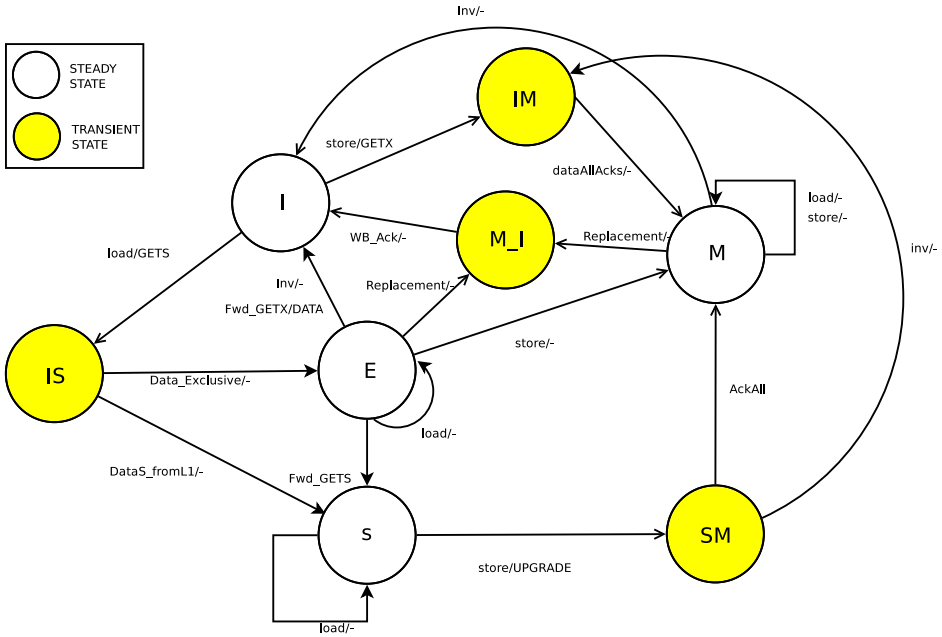


Figure 4.1: Typical simplified implementation of the MESI protocol; L1 cache FSM, including main transient states

4.1 The MESI protocol

The MESI protocol is composed of four steady states which give the name to the protocol (see Section 2.3). Additional transient states are added to these ones to complete the protocol Finite State Machine and to correctly manage the different possible race conditions. However, this is the description of the first cache level. Last Level Cache FSM is more complex and has to consider what is happening in higher (closer to the CPU) cache levels.

Some of the L1 cache FSM transient states will be now detailed in order to show how MESI protocol manages race conditions. Described states are useful to later support the discussion on to the added migration support.

Figure 4.1 shows a typical implementation of the MESI protocol finite state machine for the L1 cache. It is a simplification of the GEM5 simulator implementation of MESI directory protocol, which considers two levels of cache: the per core private L1 cache and shared L2 one.

This is a partial and simplified representation of the L1 cache finite state machine. However, shown transient states are useful to understand what is happening in the system and how the L2 cache FSM is implemented.

They are very important in order to avoid race conditions and to manage situations in which the cache is waiting for responses and/or data for a given line.

For example, in shown MESI FSM:

- IS means that a read request ($GETS$) has been issued for a cache block not present in cache and awaiting for response. The cache block is neither readable nor writeable.

- IM means that a write request ($GETX$) has been issued for a cache block not present in cache and awaiting for response. The cache block is neither readable nor writeable.

- SM means that the cache block was originally in S state and then a write request ($UPGRADE$) was issued to get exclusive permission for the block and it is awaiting response. The cache block is readable.

- M_I indicates that the cache is trying to replace a cache block in M state and the write-back ($PUTX$) to the L2 cache's directory has been issued but awaiting write-back acknowledgement.

Considering the L2 cache controller the stable states only are detailed below, since they are the starting point from which a migration action can take place. The interested reader can find a complete description of the L2 cache controller in [30].

- NP means that the cache block is not present in LLC.

- SS indicates that the LLC block is valid and readable and that it is present in potentially multiple private L1 caches, in only readable mode. It is similar to the S of L1 cache finite state machine.

- M means that the cache line is not present in no L1 cache and so the block has exclusive permission.

- MT means that the block is in one private L1 cache and it is the owner. Any request from other L1 caches needs to be forwarded to the owner.

4.2 The Broadcast Mechanism

We are considering a cache hierarchy system made of a per tile private L1 cache and a shared last level L2 cache, distributed among the tiles. In this kind of system a load or a store results in a request to the private L1 cache to retrieve the block. If the request triggers an L1 miss, a subsequent request is sent to the L2 cache bank in which the block is mapped. Last, the LLC bank sends a request to the memory if an L2 miss occurs. The block from the main memory is sent to the L2 requestor that is in charge of sending it to the requesting L1 cache.

The proposed DNUCA design is endowed with a L2 to L2 broadcast mechanism that is capable of searching for blocks in the last level cache. Thus the LLC bank to bank communication is introduced, as an additional layer in the coherence protocol to dynamically retrieve mapped cache blocks.

Figure 4.2 depicts a complete data request that involves the broadcast mechanism. An L1 cache miss triggers a request to the LLC that is selected using an SNUCA mapping from the required memory address. If an L2 cache miss occurs in the home bank, no request is immediately sent out to the memory, in order to retrieve the block. A broadcast request is sent out to all other L2 cache banks instead, due to the fact that a block can be placed in several banks, in DNUCA systems.

After receiving a broadcast request, each bank must reply to the L2 home bank which sent it. If the block has been found, it is sent to the requestor L1 cache and an HIT message is sent back to the home bank. If the bank does not have the block, it sends a NACK message back. If the home bank gets all NACK messages from all the other banks, a request to the memory controller is sent. This means that the block is not present in the LLC.

Obviously this mechanism has to be properly managed, to avoid race conditions and coherence issues. In particular, multiple broadcast processes for the same line can be active at the same time with a net effect of a recursive broadcast that can lead to deadlock scenarios.

For example, supposing a tiled system, it is possible to map blocks directly in the local LLC bank (the one belonging to the same tile of the requesting L1 cache). This is a good way of decreasing access latencies. However, this means that the block is mapped in the tile where it is requested, every time. Considering a DNUCA based system in which broadcast (or multicast) is used and where blocks in LLC can be migrated, we can have issues. It is

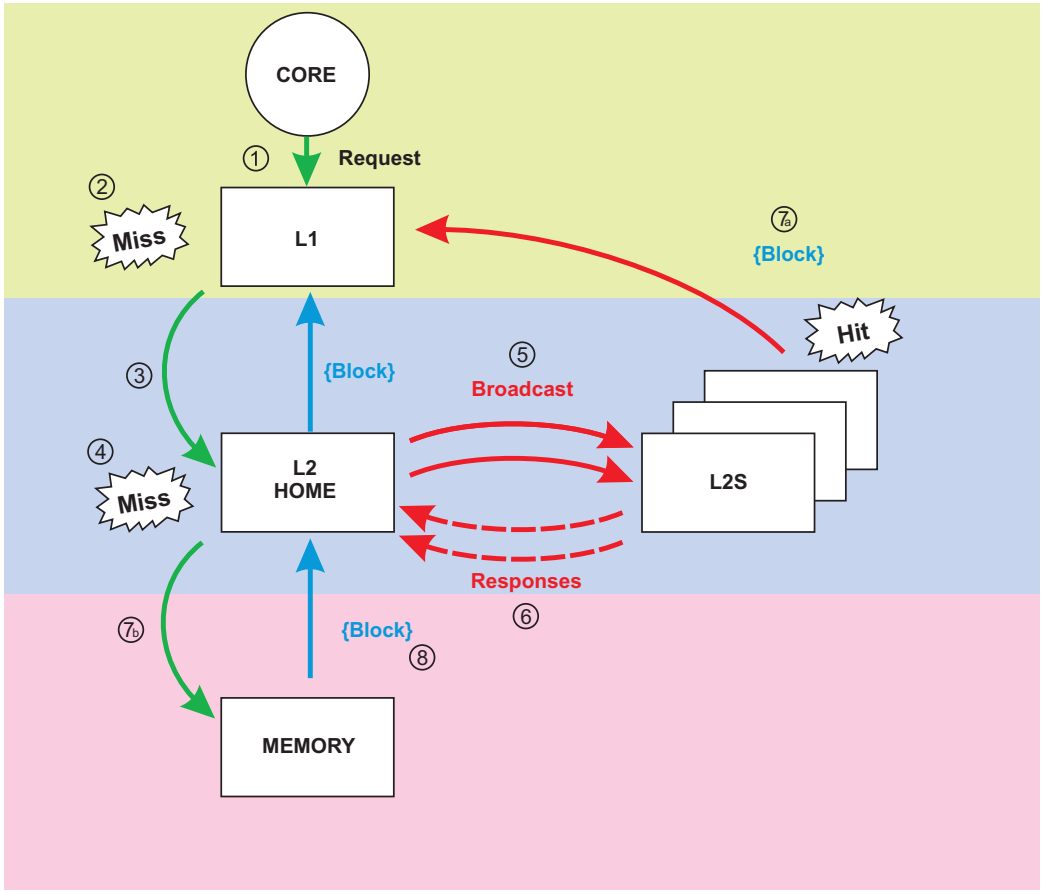


Figure 4.2: (1) request to L1; (2) miss in L1; (3) request to L2; (4) miss in L2; (5) broadcast to all other L2 caches; (6) other L2 caches send back a NACK or an HIT message; (7a) if a block is found a HIT message is set back and the block is sent to the L1 requestor; (7b) if all other L2 caches have sent back all NACK messages, a request to memory is sent; (8) the memory sends back the block; (9) the block is sent to the L1 requestor.

possible that multiple L2 banks receive requests for the same block from their local L1 caches. If the block is not present in the LLC, both the banks will send broadcast messages for the same line and after collecting all the NACKs both the banks will send a request to memory.

This situation can lead to have two L2 cache copies of the block. Considering classic inclusive coherence protocols, this situation actually breaks coherence.

In order to face this scenario maintaining the LLC local mapping, tricky and performance-affecting solutions are required. For example, it can be necessary to privilege a broadcast process and abort all the other ones.

Our design attacks the race condition issue in a simple way. As will be detailed in Section 4.3, for each private L1 cache bits have been implemented in order to permit to L1 caches to know in which bank an LLC cache block is situated. If the L2 block is replaced or migrated, these bits are reset and updated. If a L1 cache does not have the block, it sends requests to the static home bank for the block. When the line is received by the L1 requestor, the bits are updated.

Thus, an LLC home bank where the block is mapped and that is queried if the line is not present in L1 cache still exists. Moreover, if a broadcast is active and a request for the same line is sent, it will be received by the same bank that is managing the broadcast. So, it can make the new request wait the end of the active broadcast process before managing it. However, broadcasts for different blocks can coexist.

4.2.1 Deadlock Avoidance Analysis

The deadlock avoidance at protocol level is a key aspect to be considered after a change in the coherence protocol. Traditionally, the resource dependency graph is used to analyse possible deadlock sources. In particular, the deadlock at protocol level occurs due to the finite buffers in the cache controllers. Each incoming message is enqueued in the buffer of the receiving controller before being processed. Some messages impose a strict process order. However the FIFO nature of the controller input buffer coupled with the out of order deliver imposed by the NoC can force to serve a message that has a dependency on another message before its dependency. This leads to deadlock situations. To this extent different input buffers and virtual message classes are used to avoid such a scenario. Conversely, full FIFO

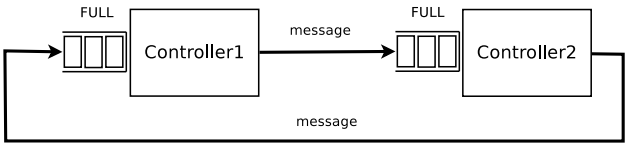


Figure 4.3: Typical example of deadlock caused by the use of the same resources.

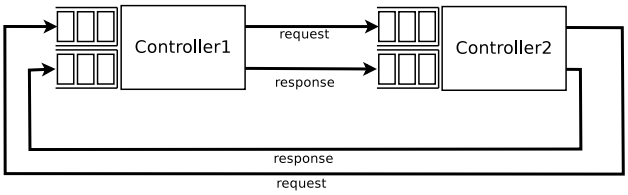


Figure 4.4: Typical example of deadlock free scheme in which different messages types use different resources.

queue cannot accept any message even those that should be processed before with the message already stored in the buffer itself, thus highlighting another deadlock scenario.

Figure 4.3 shows a simple example of deadlock. Two coherence controllers are responding to each others requests. However the FIFO queues are already full due to other requests and each controller stalls trying to send a response. Responses cannot pass requests, because the queues are FIFOs, thus the controller cannot switch to work on a subsequent request (or get to the response) and the system deadlocks.

Figure 4.4 shows how the use of separate networks and input buffers between different message classes can avoid deadlock situations. Two simple message classes are considered: requests and responses. In this way, for example, a response cannot be blocked by a request and the cyclic dependence is avoided.

The considered MESI protocol usually uses three networks to avoid deadlock: requests, responses and forwards. A request can trigger a forward message instead of a direct response, thus there are three message classes and each of them requires its own network.

Figure 4.5 shows the use of the 3 VNETs in the baseline architecture.

The request for the LLC block is sent through the $VNET0$, the one dedicated to requests. In case of hit the block is sent to the requestor L1 cache using the $VNET1$, which is dedicated to responses. In case of miss, another request is sent to the memory on the $VNET0$. The data from

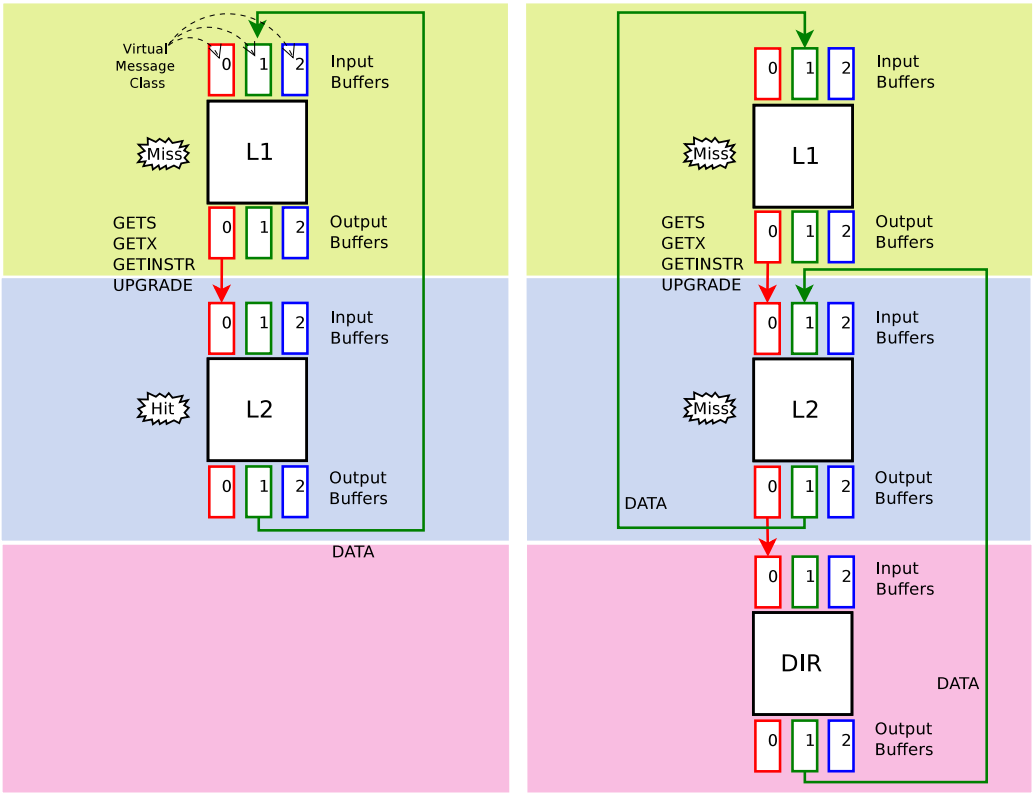


Figure 4.5: Virtual Networks management in MESI protocol implementation.

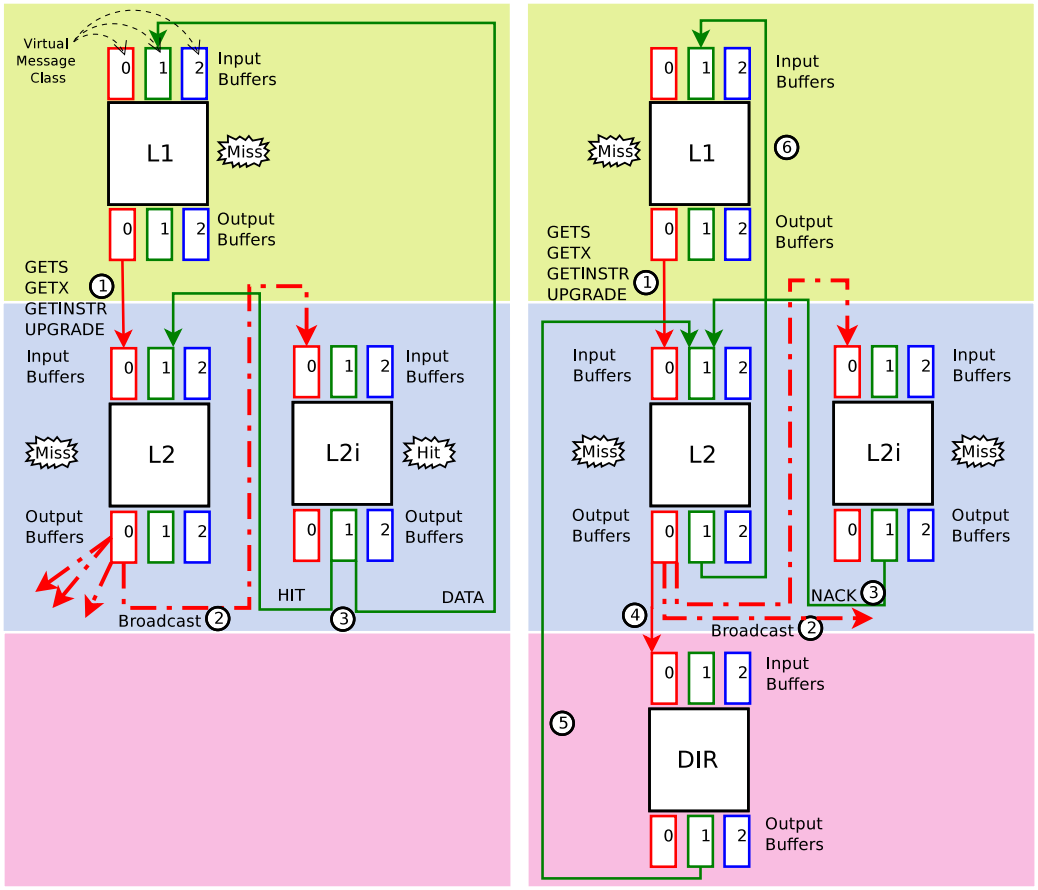


Figure 4.6: Virtual Network use in MESI protocol with broadcast support. 1) a request is sent to home LLC bank, due to a L1 cache miss (such as baseline MESI protocol); 2) L2 cache miss: broadcast process, using VNET0; every other L2 bank responds on VNET1 with NACK or HIT message. If the home bank received a HIT: 3) the block has been sent to the requestor (VNET1). If all NACKs were received: 4) a request is sent to the memory on VNET0; 5) Directory sends the block on VNET1; 6) the block is sent to the L1 cache requestor on VNET1.

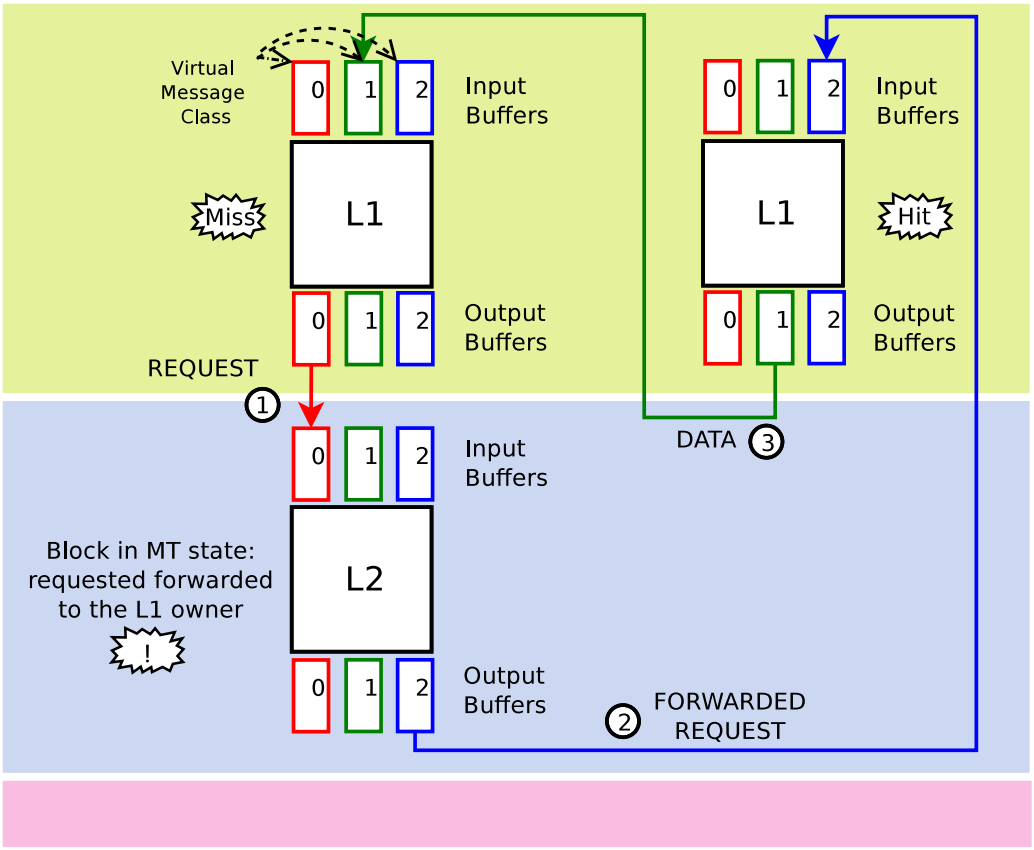


Figure 4.7: Example of the VNET2 forward in considered MESI protocol.

memory to the LLC bank and from the LLC bank to the requestor L1 cache is sent through $VNET1$.

The use of the three VNETs with the broadcast support is depicted in Figure 4.6 considering two scenarios. The left figure shows a successful broadcast transaction, where the block line is retrieved in the LLC.

In addition to the baseline architecture, the LLC bank forward the request to all other banks on the $VNET0$; these ones respond to the home bank on the $VNET1$, dedicated to responses and the block is sent to the requestor L1 on the same VNET.

Conversely, the image on the right side in Figure 4.6 shows a case where the requested block is not present in the LLC, thus the L2 has to trigger a memory transaction. Here, requests to and responses from memory are managed as in the baseline architecture.

Figure 4.7 shows an example of a forwarded request which implies the

use of the third message class.

4.2.2 The Smart Broadcast

As already anticipated and how will be shown, the broadcast imposes a non negligible, additional traffic to the NoC, thus effecting both the power and performance metrics. Considering the DNUCA scheme proposed in this chapter, each block is initially loaded from memory and mapped into the Static NUCA home bank if not present in the LLC. Later on, it can be migrated to another LLC bank. Thus, the broadcast is only necessary if the block is migrated. Otherwise the block can be in its home bank or not present il the LLC at all. The proposed smart broadcast implements a counter for each set that is incremented every time a migration for that set is triggered. If a request in the LLC home bank results in a miss, a broadcast is generated only if the counter for the set of the requested block is greater than 0; otherwise the home bank will retrieve the line directly from memory. This adaptive solution is able to cut off the broadcast overheads when the required block is not in the LLC.

If a block is invalidated or replaced and it is not in its home bank this means that it has been previously migrated; every time this situation occurs, a coherence message is sent to the static home to decrement the pending set counter.

However, considering the destructing impact of the broadcast, a smarter mechanism is considered. It stores the TAG address of each migrated block to narrow the broadcast actions to the blocks that have been really migrated. In particular, the static home bank for a specific block is notified back once the migrated block has been replaced in another cache bank.

4.3 Data Migration for DNUCA Support

The block migration technique is exploited in DNUCA architectures to decrease the block access latency by reducing the distance between the L1 cache requestor and the LLC bank where the block is stored. Moreover, it allows the dynamic block placement. A migration policy is obviously necessary to exploit the mechanism and to make the block closer to the requestor or to better utilize the cache.

This section presents the proposed migration mechanism coupled with a simple policy to steer it. The policy aims to keep useful blocks in the last level cache to decrease the congestion in the most accessed sets of specific banks.

4.3.1 Migration Mechanism

The migration mechanism complements the DNUCA scheme by allowing to dynamically move an already mapped block from a bank to another. While the DNUCA scheme can avoid data migration, such mechanism can greatly increase the utilization of the cache hierarchy. This section describes the data migration mechanism implemented on the top of the baseline protocol supporting a broadcast mechanism as the one described in Section 4.2. In particular, the extension to the MESI coherence protocol as well as the coherence states from where a migration process can start are discussed.

The key aspect that have to be considered managing block migration is the coherence one.

For example, let's consider a cache line IS (I to S) state in the L2 bank. This means that a request to the memory has been sent and data is not in the bank yet. The block can't be migrated in a such transient state because it is not physically present in the LLC cache.

The L2 finite states machines describing the coherence protocol have been extended with specific migration states. Moreover additional messages have been introduced to support the data migration without triggering races.

In the rest of this section two L2 banks are assumed to be the sender and the receiver of a migration transaction, respectively, and both the sender and the receiver coherence protocol extensions are discussed.

Figure 4.8 shows the extensions to support the sender side of the data migration mechanism.

Sender States, Transactions and Events - The sender represents the initiator of a migration transaction, that owns the data block. The migration process can start when the block is in a steady state of the last level cache FSM. The sender sends a migration request to the designated receiver to start the migration process. Moreover the sender enters in a newly designed transient state for the considered data block. This is the $Xmig^D$ state; the LLC bank has communicated that a migration is started to the receiver and it is waiting an ACK in order to properly start the process.

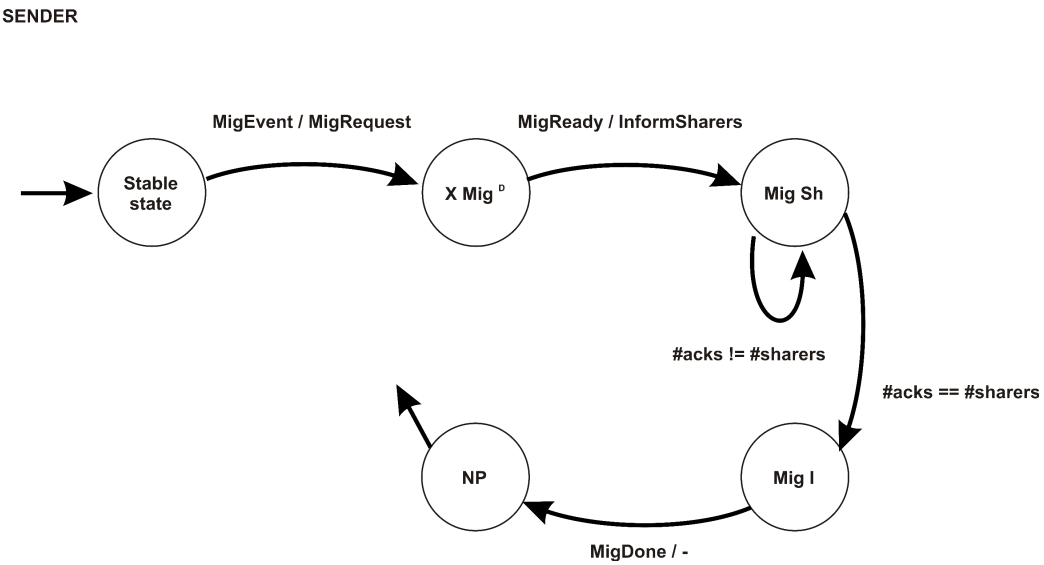


Figure 4.8: Sender Finite State Machine

When the sender receives the response, it enters in a new transient state, i.e. $MigSh$. In this state it sends a new type of message to all the L1 sharers of the block (or to the owner, dependently on the starting state we are considering) to inform them about the location change of the block. Now, it has to wait the ACK responses from all the sharers before physically sending the line to the receiver.

The sender waits for all the ACK responses from the L1 caches before moving to th the $MigI$ state that represents the time when the sender relinquish the block ownership and sends it to the receiver. It is semantically similar to typical transient replacement/invalidation states in traditional coherence protocols. The block is not valid in sender's cache while it is stored in the Miss Handling Status Register (MHSR) and the bank does not respond any more for it. In this state the sender bank is waiting for the message from the receiver which acknowledges the end of the migration. Once received such message the sender switch the block state to Not Present (NP) and the migration process is over.

Receiver States, Transactions and Events - Additional states to represent the receiver behaviour are fewer and simpler than the ones added for the sender.

When the migration request arrives to the receiver bank, the state of the block in the receiver is not present (state NP). Only one copy of the line can

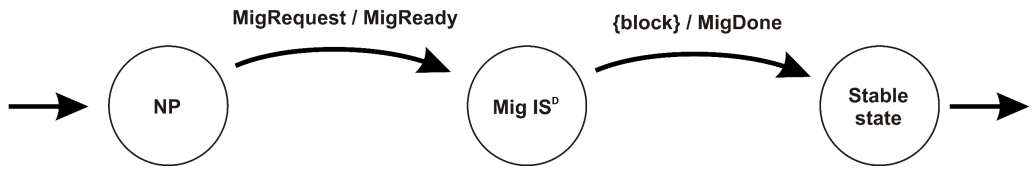


Figure 4.9: Receiver Finite State Machine

be found in the last level cache. The receiver can accept or not the request. In particular, the proposed implementation can handle a single migration per set at once, regardless if the bank is a sender or a receiver. An abort message is sent back to the sender and the migration transaction ends if the receiver is already involved in another migration process. Otherwise, the receiver bank allocates the space for the block, i.e. replacement procedure, and sends back an ACK message to the sender. Moreover the $MigIS^D$ is set in the allocated cache line. This state is semantically similar to the IS one, in which the bank is waiting for data from memory. Similarly, the receiver is waiting for the cache line from the sender bank.

When the block is received, the receiver sends a *migration done* message to the sender. Now, the new state is the steady one in which the sender bank had the block before the migration process.

Additional L1 Bits - Traditionally, SNUCA solutions force the L1 caches to statically request and map each LLC block. However, the migration allows a specific block to be mapped and remapped to different LLC banks. Thus, the broadcast mechanism plays a central role to retrieve the cache line in the LLC. However, the L1 has always to request a missing block to the static home bank first, thus triggering a broadcast if the cache line is not present in the home bank.

In order to avoid to query the home and to cause a broadcast every time, the L1 cache line status bits are augmented to store the location of a line in the LLC banks. The LLC sends the updated bits to the sharers L1s during a migration to inform the L1s on the new cache line location. Moreover, they are updated in the L1 every time the block is retrieved from the LLC. In this way a L1 cache will cause a broadcast process only the first time it queries the home, i.e. when the L1 cache does not have the block.

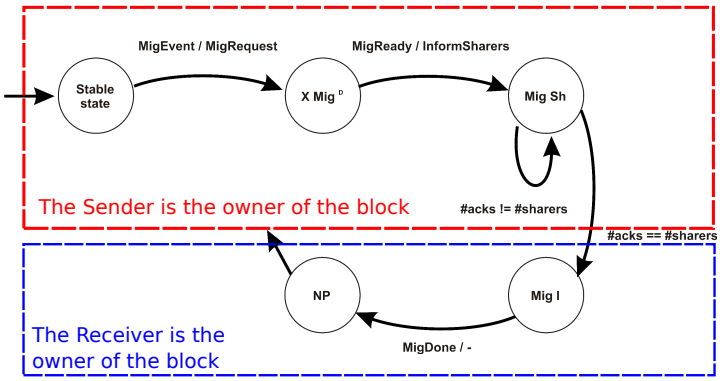


Figure 4.10: Sender Finite State Machine with Ownership details

The L1 bits are reset every time the L1 replaces the line or when an invalidation message is received from the LLC.

Retry Mechanism - During the migration process both sender and receiver can receive L1 requests for the migrating block. In particular, the sharers continue to send requests to the sender for the specific block before receiving the new data block location. Conversely, the some sharers that have already received the location update for the migrated block can start requesting to the receiver bank that is possibly still waiting for the block. Thus, a *RETRY* mechanism has been designed to manage such scenarios. Every time a request is received in a migration state, it is sent back to the L1 requestor, that will later resend the message. The requests that have been sent back by the LLC will be sent it back to the right bank (the receiver) sooner or later thanks to L1 bits that will be updated. So, the receiver with the line in steady state will receive the request and will manage it.

4.3.2 Deadlock Avoidance Analysis

The data migration mechanism introduces several threats to the coherence protocol design. While deadlock avoidance still represents a key required property for the final scheme, data ownership, data duplication and starvation are three other issues to be addressed.

Figure 4.10 shows how the sender remains the owner of the block until all sharers have been informed of the migration and until they have responded to the sender. Figure 4.11 highlights how the receiver becomes the owner of the block only when the it is received and Migration Done message is sent

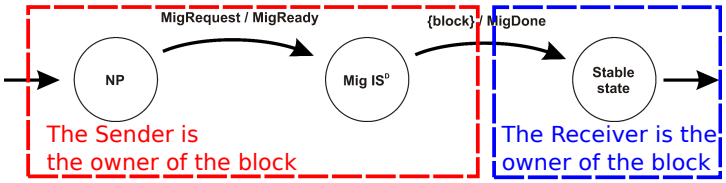


Figure 4.11: Receiver Finite State Machine with Ownership details

to the sender.

This marks a specific point in time during the migration transaction to avoid having multiple caches to answer to the L1 controller request for the migrated block.

Moreover, two sets of requests that are sent during the block migration are described below:

- requests sent by the sharers/owner;
- requests sent by other L1 caches which are not sharers or owner.

Considering the $MigSh$ sender state, messages for the block sent by sharers/the owner can be received either by the sender either by the receiver, due to the fact that they are being informed of the migration and this process can be not finished yet.

Moreover, these messages can arrive with different orders independently on when they have been sent, if we consider a not ordered interconnect (i.e. a NoC).

The previously explained $RETRY$ mechanism together with L1 cache bits forces these requests to be managed only by the Receiver, when the migration is done.

The sharers/owner L1 caches will send requests to the bank where the block is mapped. If it is one of the actors of the on the fly migration, the Sender and the Receiver send $RETRY$ messages to the L1 requestors. This mechanism permits to change ownership, to inform sharers/the owner and to complete migration, without the requests interference. They are sent back again until they arrive to the receiver, in the steady state.

Considering requests sent by other L1 caches, they will be received by the home bank. If it is one of the migration actors, again the $RETRY$ mechanism is used. At the end of the migration process, from there, if the block has been migrated, it can be retrieved by using broadcast.

4.4 A Novel Migration Policy

The migration policy embeds the logic to trigger the data migration by exploiting the hardware mechanisms described in Section 4.2 and 4.3. The cache hierarchy in modern architectures fills the performance gap between the CPU and the memory where the latter is usually slower. However the multi-cores highlight the tile based architectures as a viable solution to increase the core count in the chip. Thus, the L2, that usually represents the LLC, is physically split in multiple banks even if it remains shared between the whole system. To this extent, the accesses to the LLC banks can be far from uniformly distributed or a single cache line can be mapped far from its requestor. In this scenario the migration policy allows to remap the cache lines to better use the LLC from both the energy and the performance viewpoint.

Moreover the reduced number of accesses to memory, due to the better LLC utilization, improves the overall system performance. Thus, it is important to guarantee that the heavily accessed blocks remain in the LLC.

Starting from a tiled multi-core with split shared last level cache and static mapping, two different observations are the key pillars for the proposed policy design. First, accesses are unbalanced between banks. This is due to the address based static mapping: if applications access some blocks more than others, eventually some banks are accessed more than others.

Second, accesses are very unbalanced between the sets of some banks. Some sets are more accessed than others and this leads to additional conflict misses and reduces the capacity ones. So, to avoid replacements and to keep most accessed blocks in LLC, our policy migrates lines belonging to the most accessed sets to tiles where these sets are less accessed. However, the least recently used block is migrated in order to avoid to cause additional broadcasts for most accessed blocks. This allows to keep data in the LLC, without the need of paying massive memory access costs. Moreover it allows to better exploit the capacity of the last level cache. The policy sits on the set congestion metric that is defined as:

$$M_{i,s} = replacements_{i,s}/accesses_{i,s} \tag{4.1}$$

for tile i and set s.

Accordingly, a migration is triggered in the bank i, for the set s when :

$$M_{i,s} >= \mu_i + L * \sigma_i \tag{4.2}$$

where μ_i is the average value of the metric for tile i, considering all the sets and σ_i is the standard deviation of the metric for tile i, considering all the sets. Last, L is the level of migration which is desired, it can be equal to 1, 2 or 3 (*high*, *medium*, *low*).

This expression considers a Gaussian distribution of the $M_{i,s}$ congestion metric, due to the Central Limit Theorem [26], which states that considering a large amount of samples, a unknown distribution can be considered approximatively as a Normal one of expected value equal to the sample average and standard deviation equal to the one computed on samples.

This assumption is justified by the fact that every considered sample is the value of the congestion metric for a specific set in a tile. Thus, the number of samples corresponds to the number of sets and the considered architectures present much more than 30 sets. This number is usually used to define a set of samples big enough to be characterized as a Normal distribution [27].

Thus, a migration is triggered when the value of the metric $M_{i,s}$ for the tile i and the set s is greater than the expected value of the distribution μ_i for the tile i plus L times its standard deviation. In a Normal distribution only 0.15% of the data is greater than the expected value plus three times the standard deviation. So the value of 3 is the one setted as the "low level" of migration. The level L can be setted to 2 as a "medium level": out of the range $\mu - 2 * \sigma, \mu + 2 * \sigma$ there are the 5% of data. Finally the level L can be setted to 1 as a "high level" of migration due to the fact that out of the range $\mu - \sigma, \mu + \sigma$ there are the 32% of data.

However, a migration can be "refused" by the receiver. A migration is accepted only if the value of the metric of the sender is greater than the one of the receiver, as stated by the following equation:

$$M_{i,s} >= M_{j,s} \tag{4.3}$$

i is the sender bank, j is the receiver one.

Another optional condition can be used in order to accept migrations or not. The level of congestion of the set of the receiver can be taken in consideration and it is defined as:

$$M_{j,s} < \mu_j + \sigma_j \tag{4.4}$$

In order to reach this scope the value of the metric of the receiver has to be less than the expected value considering all the sets plus the standard

deviation. This can be meter of not congestion for that set in the receiver.

Chapter 5

Analysis: DNUCA and Application Behaviour

This chapter delivers a comprehensive analysis encompassing different aspects of a DNUCA system. The classification of a representative subset of SPEC 2006CPU benchmark suite is discussed focusing onto the application phases and their impact to the DNUCA architecture. Moreover, the hardware mechanisms to support the DNUCA solutions, i.e. broadcast and data migration are explored. Last, a complete DNUCA system is considered.

To this extent, the rest of the chapter is organized as follows. Section 5.1 describes the simulation setup, the target architecture and the benchmarks. Section 5.2 overviews the application behaviour analysis. Section 5.3 shows the pure broadcast mechanism penalties. Section 5.4 details performance and additional traffic results considering the baseline architecture and the smart broadcast based one; moreover, the migration mechanism is evaluated and additional observation on the conjuncted mechanisms are discussed.

5.1 Simulation Setup

Starting from the SNUCA architecture that will be referred as "baseline" in the rest of the chapter, different mechanisms and policies are added and are analysed. First a broadcast mechanism is implemented on the top of the baseline and the resulting architecture is referred as "broadcast-base". Then, the smart broadcast solution presented in Chapter 4 is replaced to the plain broadcast (smart-broadcast-base architecture). Last, a simple DNUCA

Processor Core	1GHz, In-Order Alpha Core, 1 cycle per execution phase
L1I Cache	32kB 4-way Set Associative
L1D Cache	32kB 4-way Set Associative
L2 cache	256kB per bank, 8-way Set Associative
Coherence Prot.	MESI (3 VNET protocols)
Main Memory Access Latency	200 cycles
Topology	2D-mesh 4x4 at 16 Cores
Technology	45nm at 1.0V
Real Traffic	Subset of *SPEC CPU2006* benchmarks.

Table 5.1: Experimental setup: processor and technology parameters common to the considered architectures.

policy coupled with the coherence protocol extension discussed in Chapter 4 are introduced to deliver a complete DNUCA testing architecture.

The migration policy is executed at each time a cache request is received by a LLC bank. We assume a MESI-based coherence protocol that enforces 3 Virtual Network to avoid protocol-level deadlock. The protocol changes to avoid deadlocks, using broadcast and migration mechanisms are detailed in Section 4.2.1 and in Section 4.3.2.

The SPEC CPU2006 suite provides integer and floating point single-threaded benchmarks able to stress the cache hierarchy and the system main memory [20].

Twelve applications are shown in Table 5.2 among the whole suite [20]. They belong to the High Performance Computing Domain.

The cache configuration is a 32KB L1I caches, a 32KB L1D caches and a 256KB L2 caches per bank. The main considered NoC topology is a 4x4 2D-mesh with 16 cores and NoC routers with a 3 VC.

The architectures have been integrated in the GEM5 cycle accurate simulator [8]. Moreover, an enhanced version of the simulator [39, 36, 39, 38, 10] has been used to extract data.

SPEC CPU2006 BENCHMARKS

Application	Category	Inputs	Outputs
429.mcf	Combinatorial optimization. Integer benchmark.	Time-tables and dead-head trips, times, costs.	Log information, checksum and values.
456.hmmer	Search a gene sequence database. Integer benchmark.	A database and reference workloads.	Four output files contain a ranked list of matches.
450.soplex	Simplex Linear Program Solver. It solves a linear program using the Simplex Algorithm. Floating point benchmark.	Test uses a 497x614 grid. Train uses a 582x55515 grid. Ref uses a 2586x920683 grid and a 83060x270095 one.	The objective function or its value.
445.gobmk	Artificial Intelligence, game playing. The program executes Go and executes a set of commands to analyse Go positions. Integer benchmark.	"SmartGo Format" files.	ASCII description of a sequence of Go moves.
444.namd	Classical Molecular Dynamics Simulation. It is derived from NAMD, a parallel program for the simulation of large bio-molecular systems. Floating point benchmark.	Input file created with NAMD 2.5.	Checksums on the calculations.
464.h264ref	Video compression. Integer benchmark.	Files in uncompressed video data format.	Encode logs.
435.gromacs	Chemistry and Molecular Dynamics. It performs molecular dynamics and simulations. Floating point benchmark.	The file gromacs.tpr with identical setup but different number of steps.	Average potential energy, system temperature and the number of operations performed.
471.omnetpp	Discrete event simulator. It provides a simulation of a large Ethernet network. Integer benchmark.	The topology of the network and the structure of hosts, switches and hubs.	Extensive statistics.
473.astar	Computer games, artificial intelligence, path finding. Implementation of different path-finding algorithms. Integer benchmark.	The input file is a map in binary format.	The number of existing ways and the total way lenght to validate correctness.
437.leslie3d	Computational Fluid Dynamics. Floating point benchmark.	Three different stack sizes. Grid size, flow parameters and boundary conditions.	Analysis information.
470.lbm	Computational Fluid Dynamics. It simulates incompressible fluids. Floating point benchmark.	Number of time steps and choice between two simulation setups.	The 3D velocity vector for each cell.
459.gemsFDTD	Computational Electromagnetics. Floating point benchmark.	The problem size, number of time steps and several parameters.	ASCII file containing the requested data.

Table 5.2: The used subset of the SPEC CPU2006 benchmarks.

LLC MISSES

LOW	MEDIUM	HIGH
MCF(<700ms) [<10 000]		MCF(>700ms) [=50 000]
	NAMD [18 000]	
SOPLEX (<50 ms) [<<10 000]	SOPLEX (>50 ms) [10 000 20 000]	
ASTAR(150-200ms) [10 000]	ASTAR(30-70, 120-150ms) [20 000]	ASTAR(0-30ms, 70-120ms) [50 000]
OMNET[5 000]		
GEMS[=0]		
GOBMK(>380ms)[5 000]		GOBMK(<380ms)[30 000-50 000]
GROMACS(<1400ms) [1 000]	GROMACS(>1400ms) [10 000 - 30 000]	
H264[<10 000]		
HMMER[10 000 20 000]		
		LBM [50 000]
		LESLIE [50 000 250000]

Table 5.3: Characterization of SPEC applications phases according to the first described metric.

5.2 Benchmark Analysis

The benchmark analysis represents a key stage to be integrated in the DNUCA system design. An accurate classification of the application behaviour and its phases as well as the impact they have onto the cache hierarchy is crucial to deliver a successful architecture. In this section the miss rate and the cache accesses are used at both L1 and L2 to characterize the application behaviour in the two cache levels. The LLC miss rate measures the proportion of the activated broadcast actions, since a broadcast is triggered after an LLC miss, before accessing the main memory in the worst case.

However, the miss rate is not enough to completely represent the application behaviour. It does not consider the number of requests that have been issued to the last level cache.

For example, an high miss rate is possible even with few total accesses. A low number of accesses poorly affects the broadcast impact over the system even if the miss rate is high. Accordingly, in order to characterize the behaviour of the application in time, the number of misses is a good metric.

The second chosen metric pair is the number of L1 cache misses and the relative miss rate.

First of all, it details the ability of the L1 cache of filter the CPU requests and complement the system view on the number of L2 accesses, detailing what is happening in L1 cache. Again, the L1 miss rate is not relevant alone.

Table 5.3 shows as SPEC CPU2006 benchmarks phases can be classified using LLC misses and how much a single application can vary its behaviour

L1 CACHE MISSES

LOW	MEDIUM	HIGH
MCF(<700ms) [<5 000-10 000]		MCF(>700ms) [80 000 100 000]
NAMD[1 000 - 2 000]		
SOPLEX (<50ms) [5 000]		SOPLEX(>50ms) [100 000]
ASTAR(150-200ms) [10 000]	ASTAR(30 70ms, 120-150ms) [20 000]	ASTAR(0-30ms, 70-120ms) [50 000]
	OMNET (<100ms) [20 000 30 000]	OMNET (>100ms) [70 000]
GEMS[2 000]		
	GOBMK(>380ms) [20 000 60 000]	GOBMK(<300ms) [50 000 100 000]
GROMACS(<1400ms) [<<10 000]	GROMACS(1400-1800ms) [15 000 40 000]	GROMACS(>1800ms) [80 000]
	H264 (0-1300ms, >1600ms) [20 000 - 50 000]	H264 (1300 - 1600ms) [50 000 - 150 000]
	HMMER [15 000]	
		LBM[50 000]
		LESLIE[50 000 - 250 000]

Table 5.4: Characterization of SPEC applications phases according to the second described metric.

in time. A time period of 2 seconds is considered.

Table 5.4 details the characterization of the benchmarks using the second metric, L2 misses.

Table 5.5 shows a combined vision of the two proposed classifications.

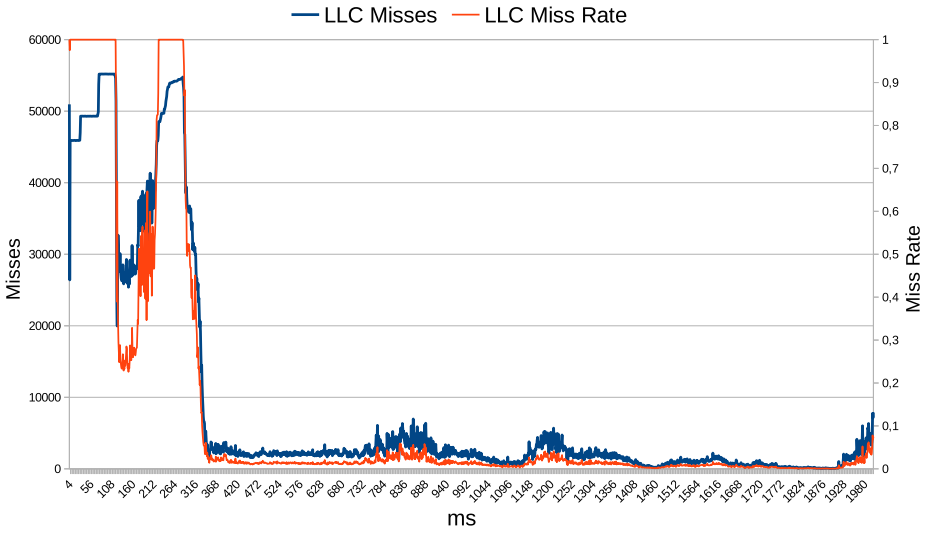
5.2.1 Applications Phases Analysis

This section discusses the application phases by considering the above defined cache hierarchy metrics. A single application can traverse multiple phases during the execution. Thus the DNUCA architecture should take such changes into account to better use the cache resources.

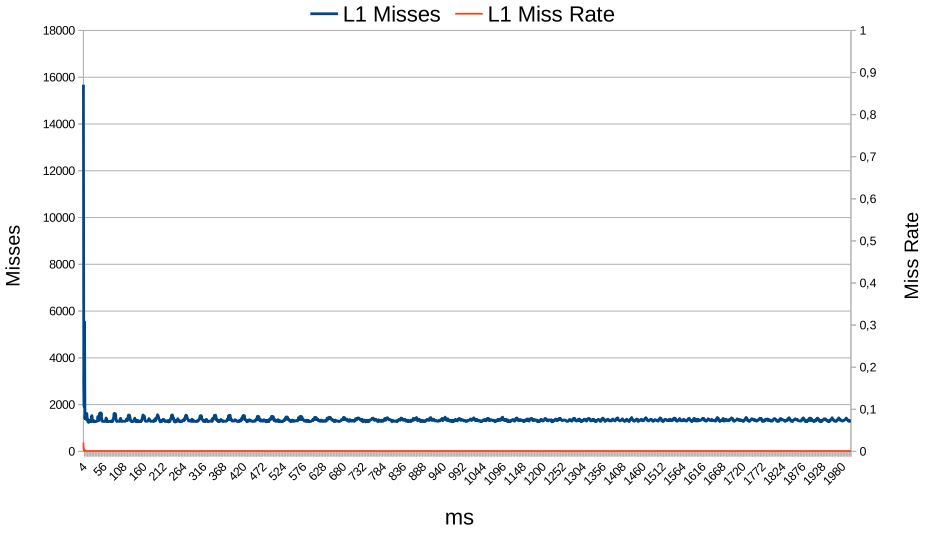
Applications with several phases are described; some of them highlight no significant phases. Results are reported considering the initial 2 seconds of the execution for each application with a sample rate of 1 ms.

GOBMK

Figure 5.1(a) and Figure 5.1(b) highlight 2 phases in the application and the transition point is at sample 340. The first one is characterized by an high number of LLC misses and high LLC miss rate. The second one presents less LLC misses and a very low miss rate. The same two different phases are visible also considering the second described metric. A first phase is characterized by more L1 misses and higher L1 miss rate than a second one. Taking the first phase apart due to the application setup, in the second phase the LLC seems to supply the majority of the requested data. A low miss rate in the LLC means a limited broadcast impact. However, L1 significantly filters the CPU requests to the L2, thus positively limiting the broadcast



(a) LLC misses in GOBMK benchmark



(b) L1 misses in GOBMK benchmark

Figure 5.1: Number of misses during the execution of two seconds of the GOBMK benchmark. We are considering the baseline architecture.

	LLC MISSES			
L1		**LOW**	**MEDIUM**	**HIGH**
M I S S E S	**LOW**	MCF(<700ms) SOPLEX(<50 ms) ASTAR(150-200ms) GEMS GROMACS(<1400ms)	NAMD GOBMK(>380ms)	
	MEDIUM	OMNET (380ms) H264 (0-1300ms, >1600ms) HMMER	ASTAR(30, 70ms, 120-150ms) GROMACS(1400-1800ms)	
	HIGH	OMNET(>100ms) H264 (1300 -1600ms)	SOPLEX,(>50 ms) GROMACS(>1800ms)	MCF(>700ms) ASTAR(0-30ms, 70-120ms) GOBMK(<380ms) LBM LESLIE

Table 5.5: Combined characterization of SPEC applications phases

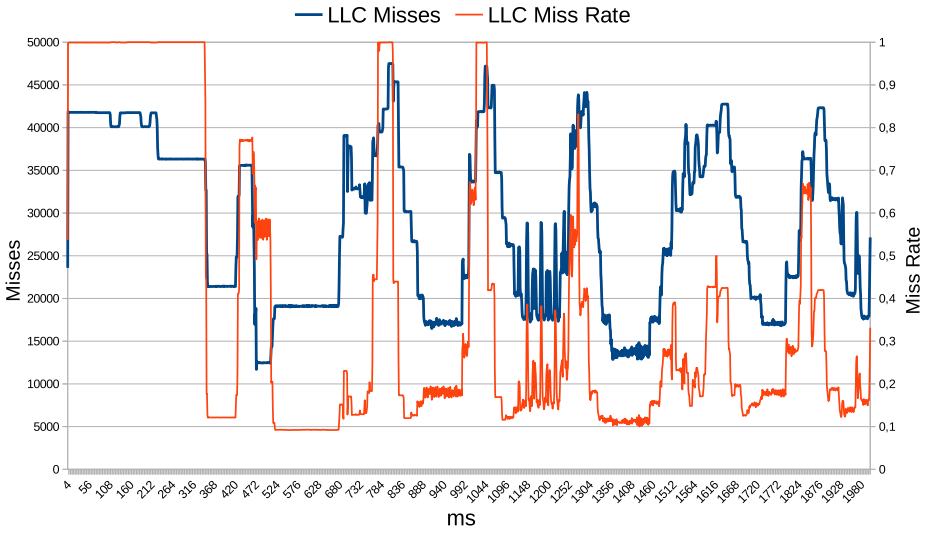
actions.

LESLIE

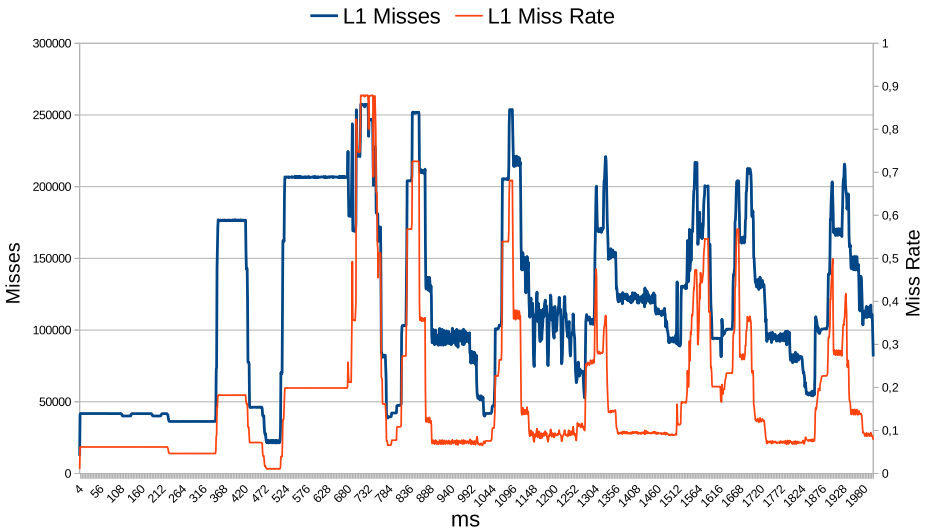
Figure 5.2 shows that several phases can be detected, considering both the described metrics. All the considered time window is characterized by high L1 and LLC misses. However, despite this observation, there are numerous peaks characterized by an higher number of misses and the miss rates are variable. It is very interesting noticing how not all LLC misses peaks corresponds to peaks in L1 cache misses; this is related to the fact that nearly all (the relatively few) blocks not present in private caches are not in LLC neither in those phases. They are new blocks that application is retrieving from the main memory. Considering the first metric, the broadcast mechanism will affect negatively the entire execution of the application. Phases characterized by more LLC misses will be affected even more heavily.

MCF

Figure 5.3 highlights 2 visible phases. The first one is characterized by a low number of LLC misses and low LLC miss rate. The second one presents much more LLC misses and a very high miss rate. The same two different phases are visible also considering the second described metric. Moreover in the first phase the L1 cache filters very well requests and presents a very low miss rate. Only a part of the blocks not present in the L1 cache are not in

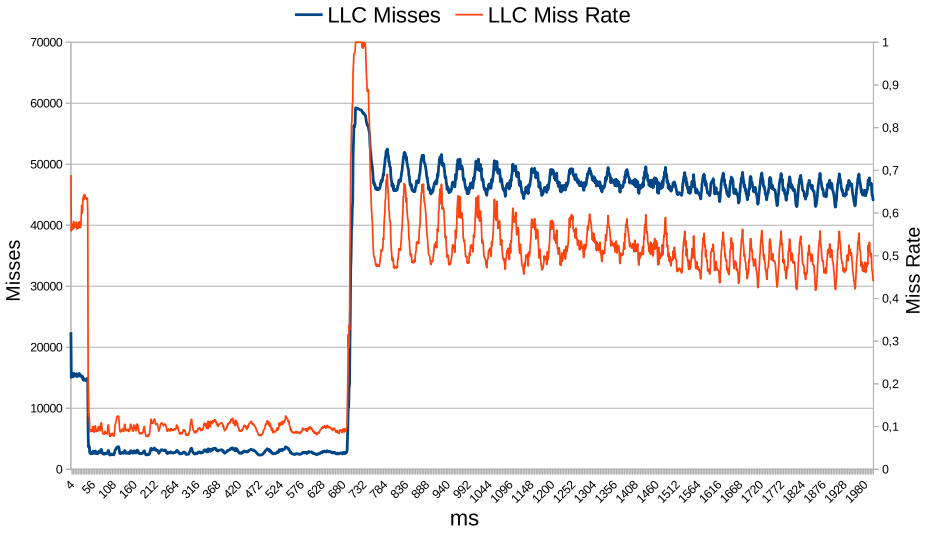


(a) LLC misses in LESLIE benchmark

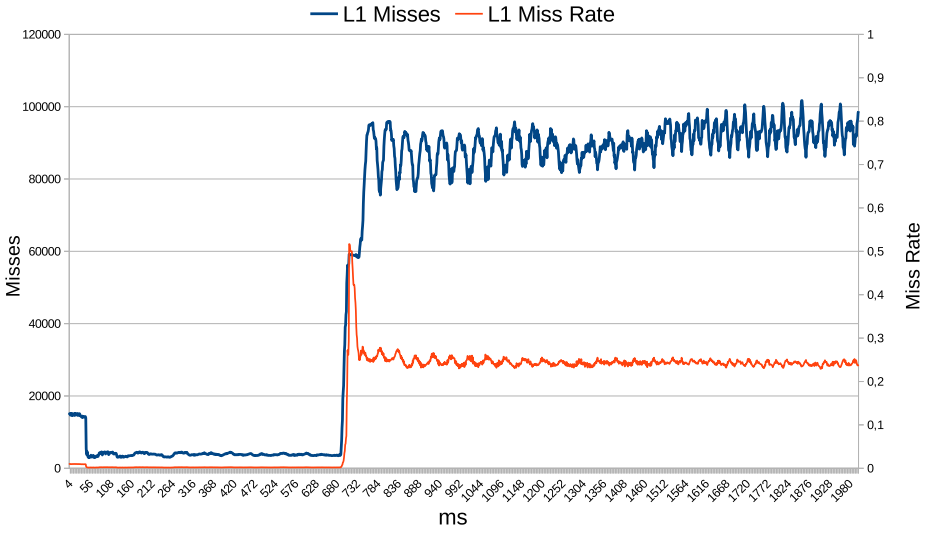


(b) L1 misses in LESLIE benchmark

Figure 5.2: Number of misses during the execution of two seconds of the LESLIE benchmark. We are considering the baseline architecture.



(a) LLC misses in MCF benchmark



(b) L1 misses in MCF benchmark

Figure 5.3: Number of misses during the execution of two seconds of the MCF benchmark. We are considering the baseline architecture.

the L2 one. In the second phase the situation is completely different. The L1 and the L2 caches present a very high number of misses.

Thus, the broadcast mechanism will affect negatively performance and power consumption much more in the second phase than in the first one, due to the high number of LLC and L1 misses.

GEMS - Low Broadcast Impact Counterexample

Figure 5.4 highlights that the entire observed period presents a very low number of misses and low miss rate either considering the LLC either the L1 cache. The broadcast mechanism will not affect negatively performance and power consumption during the observed period.

LBM - High Broadcast Impact Counterexample

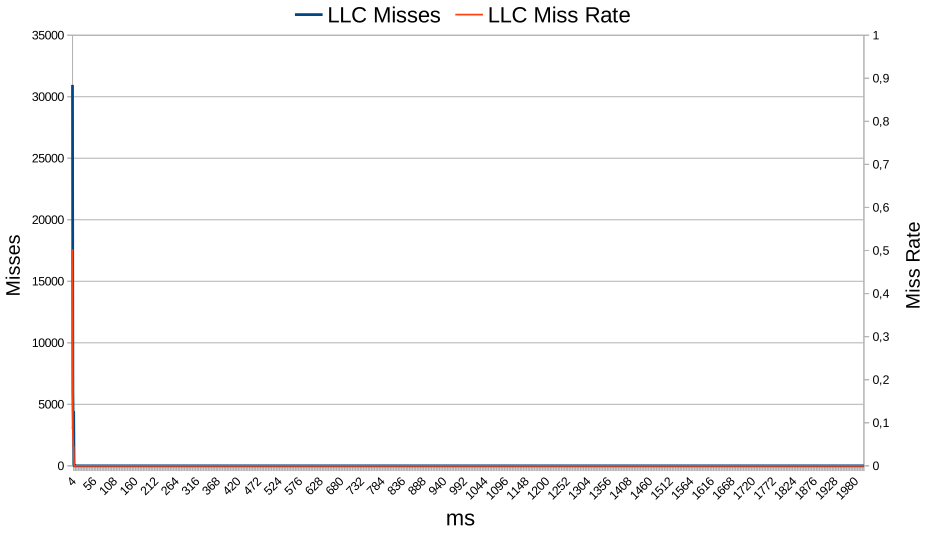
Figure 5.5 shows a very high number of misses and high miss rate either considering the LLC either the L1 cache. Considering the first (and the second) metric, the broadcast mechanism will heavily affect performance and power consumption during the observed period.

5.3 Broadcast Analysis

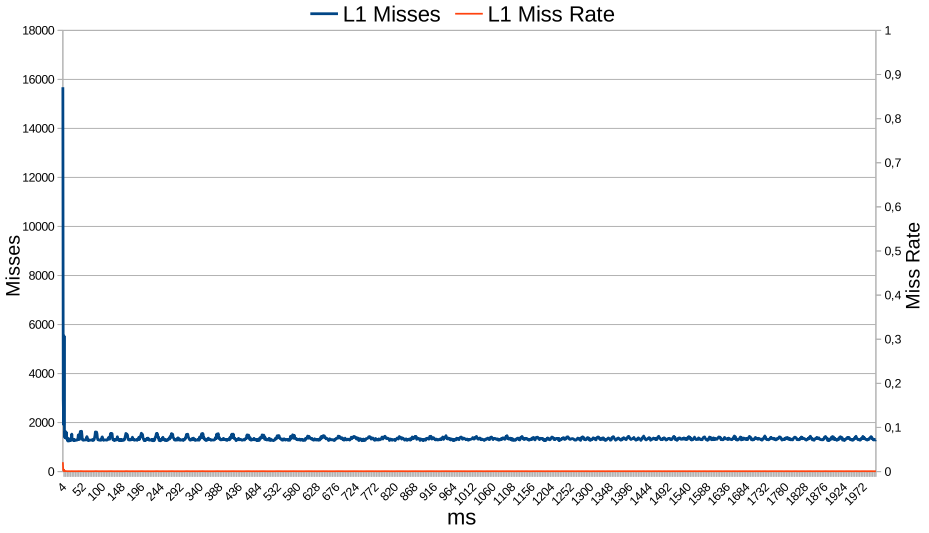
Starting from the analysed L1 and L2 metrics, this section explores the induced broadcast transactions and the performance and traffic overheads. The broadcast based architecture is considered with no migration mechanism to fully highlight the pure broadcast penalties. Thus, every LLC miss triggers a broadcast search to retrieve the block. However, no data will be find in the other LLC banks, since no migration has been done. In particular the block can only be found in the home bank or in the main memory. The considered simulated phases are shown in Table 5.6. It is worth noticing that not all the benchmarks simulated 2 seconds, due to the prohibitive simulation time that they require due to the broadcast mechanism.

5.3.1 Performance Degradation

Results are reported in Figure 5.6. Twelve SPEC CPU2006 benchmarks are reported on the x axis. One column representing the broadcast-base architecture is shown for each application. The executed instructions for a given execution time are reported on the y axis, normalized to the baseline architecture. However, for the SOPLEX benchmark the complete execution time on the y axis is reported, normalized to the baseline architecture. This

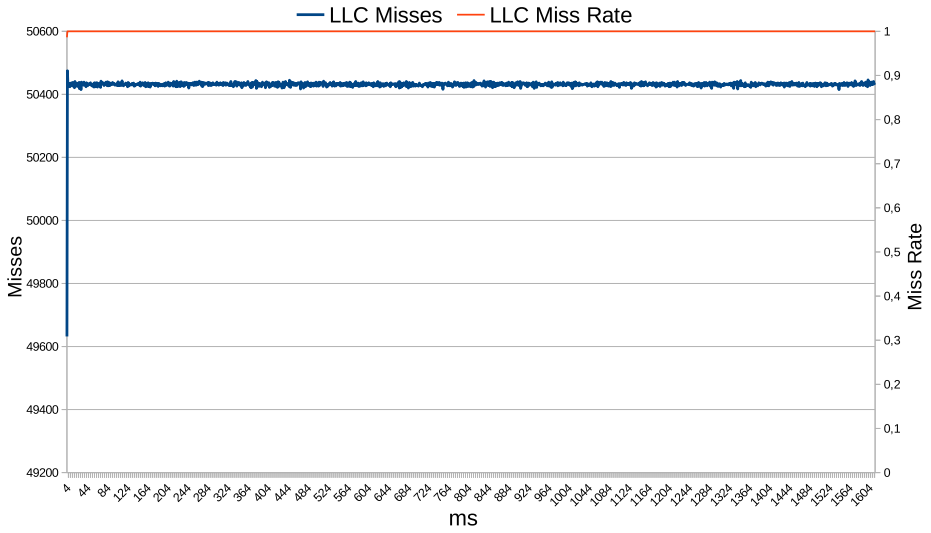


(a) LLC misses in GEMS benchmark

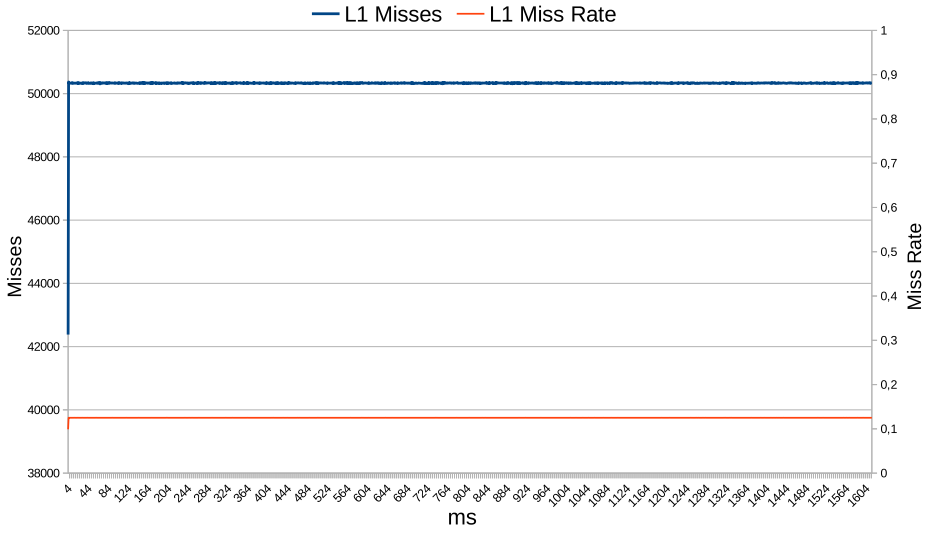


(b) L1 misses in GEMS benchmark

Figure 5.4: Number of misses during the execution of two seconds of the GEMS benchmark. We are considering the baseline architecture.



(a) LLC misses in LBM benchmark



(b) L1 misses in LBM benchmark

Figure 5.5: Number of misses during the execution of two seconds of the LBM benchmark. We are considering the baseline architecture.

ASTAR	170 ms
GEMS	2000 ms
GOBMK	2000 ms
GROMACS	352 ms
H264	2000 ms
HMMER	2000 ms
LBM	1615 ms
LESLIE	2000 ms
MCF	2000 ms
NAMD	1189 ms
OMNET	2000 ms
SOPLEX	Run to completion: 237 ms [baseline] / 259 ms [broadcast]

Table 5.6: Simulated periods: time phases that are considered in the broadcast results.

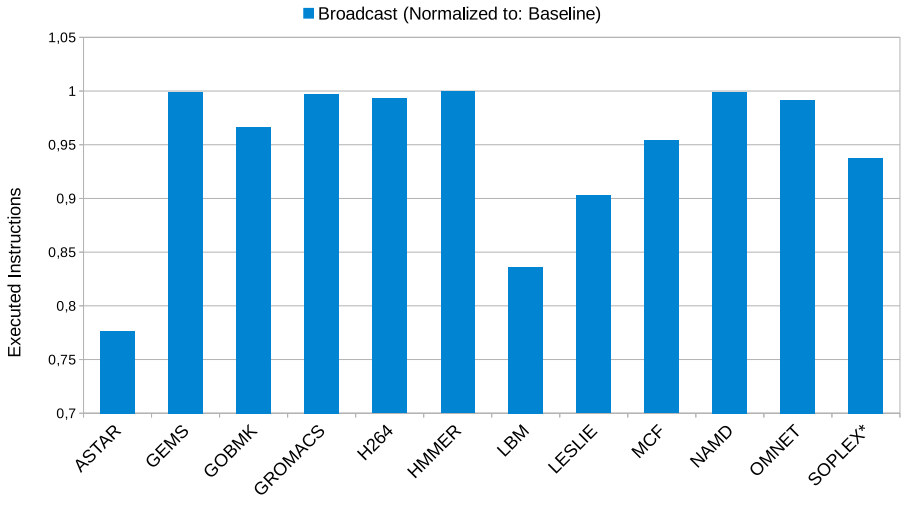


Figure 5.6: Performance penalties considering the broadcast based architecture.

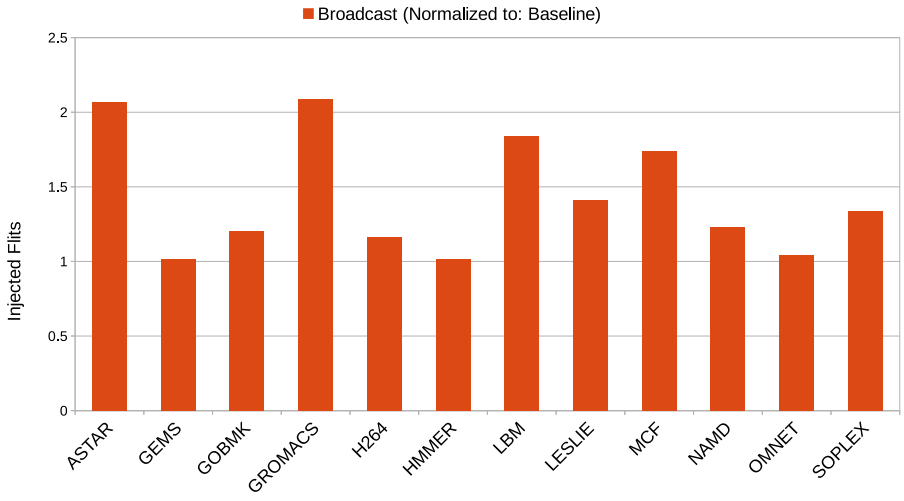


Figure 5.7: Injected Flits Increment observed in the broadcast-based architecture.

benchmark is analysed in base of the execution time, due to the fact that all the application execution is considered.

We are considering the broadcast-based architecture with no migration mechanism, in order to highlight the pure broadcast penalties. Thus, this architecture is outperformed by the baseline one by 6 % on average with a peak of 23 % with ASTAR benchmark.

Moreover, it is possible to observe how the application phases and the previously described behaviours can heavily influence broadcast performances. In particular, some applications are not heavily affected by broadcast. For example, GEMS benchmark is not affected by broadcasts due to its low number of LLC misses (see Section 5.2.1). Conversely, HMMER and OMNET show no performance degradation due to the broadcast. This is due to the fact that the considered time period presents few L1 and L2 misses.

5.3.2 Injected Flits Increment

The broadcast mechanism based architecture is compared to the SNUCA baseline one considering additional injected flits. Results are reported in Figure 5.7. The SPEC CPU2006 benchmarks are reported on the x axis, while the y shows the injected flits normalized to the baseline architecture. The broadcast causes a 43 % additional flits on average with a peak of 108 % (see GROMACS). The number of injected flits heavily influences the in-

terconnect traffic and directly impacts power consumption.

While the broadcast strongly influences the additional injected flits for the majority of the considered applications, GEMS, OMNET and HMMER shows a limited impact due to the broadcast. This is caused by the absolute low number of LLC misses (see Section 5.2.1 and Section 5.3.1).

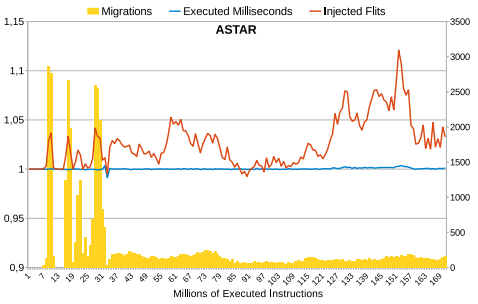
Moreover, GROMACS, H264 and NAMD injected flits are negatively affected by migration, despite the fact that their performance does not present a clear degradation. The LLC misses are few enough to not influence the performance. However, the additional traffic caused by broadcasts impacts on the overall injected flits of the considered rime period.

5.4 Smart Broadcast with Simple Policy

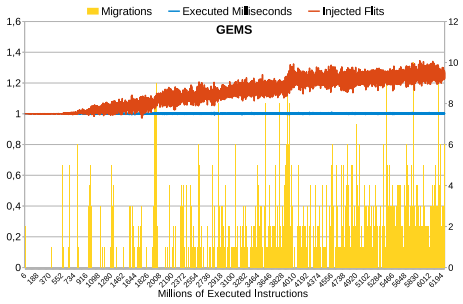
This Section analyses a complete DNUCA architecture within a 4 by 4 2D mesh multicore. The smart broadcast and the migration mechanism have been implemented considering the migration policy detailed in Section 4.4. Figure 5.8 and Figure 5.9 reports the simulation results for 12 SPEC CPU2006 benchmarks. The execution time and the injected flits per a million of instructions are reported on the y axis, normalized to the baseline architecture. The number of the triggered migrations is reported on the second y axis. Note that a migration can farther trigger several broadcast actions if the same data is accessed from an L1 miss.

The smart broadcast based architecture cuts off the performance penalties due to the standard broadcast. In particular the Smart Broadcast shows the same performance of the baseline architecture when no migrations are active. In every analysed application, results show how performance is very similar to the baseline and how the ratio between Smart Broadcast and Baseline presents a negligible degradation. The Smart Broadcast use its LLC block search mechanism only if the required block has been migrated, avoiding useless and performance degrading broadcast actions. This mechanism nearly completely overcomes the performance degradation of the standard broadcast.

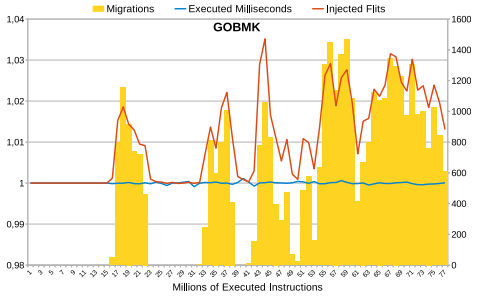
The injected flits analysis is detailed below. The smart broadcast based architecture measures an higher number of injected flits compared to the baseline architecture, on average. However, the entity of this increment depends on the considered benchmark and on the number of triggered migrations.



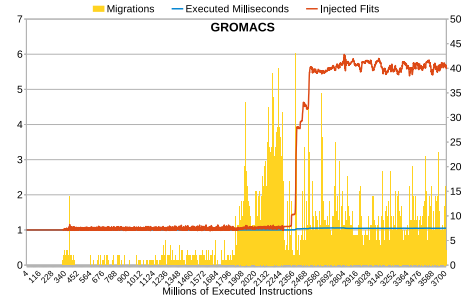
(a) ASTAR Results



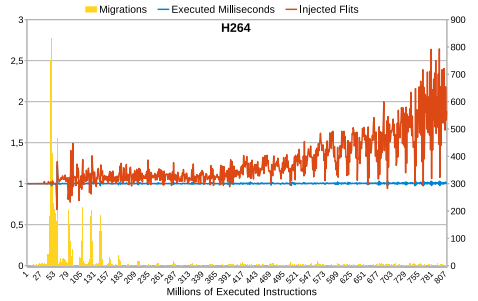
(b) GEMS Results



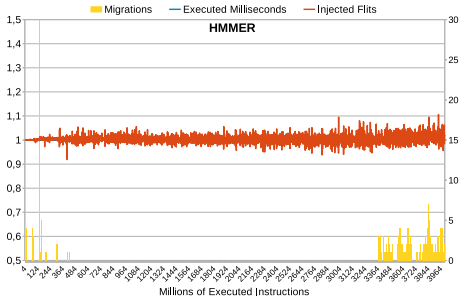
(c) GOBMK Results



(d) GROMACS Results

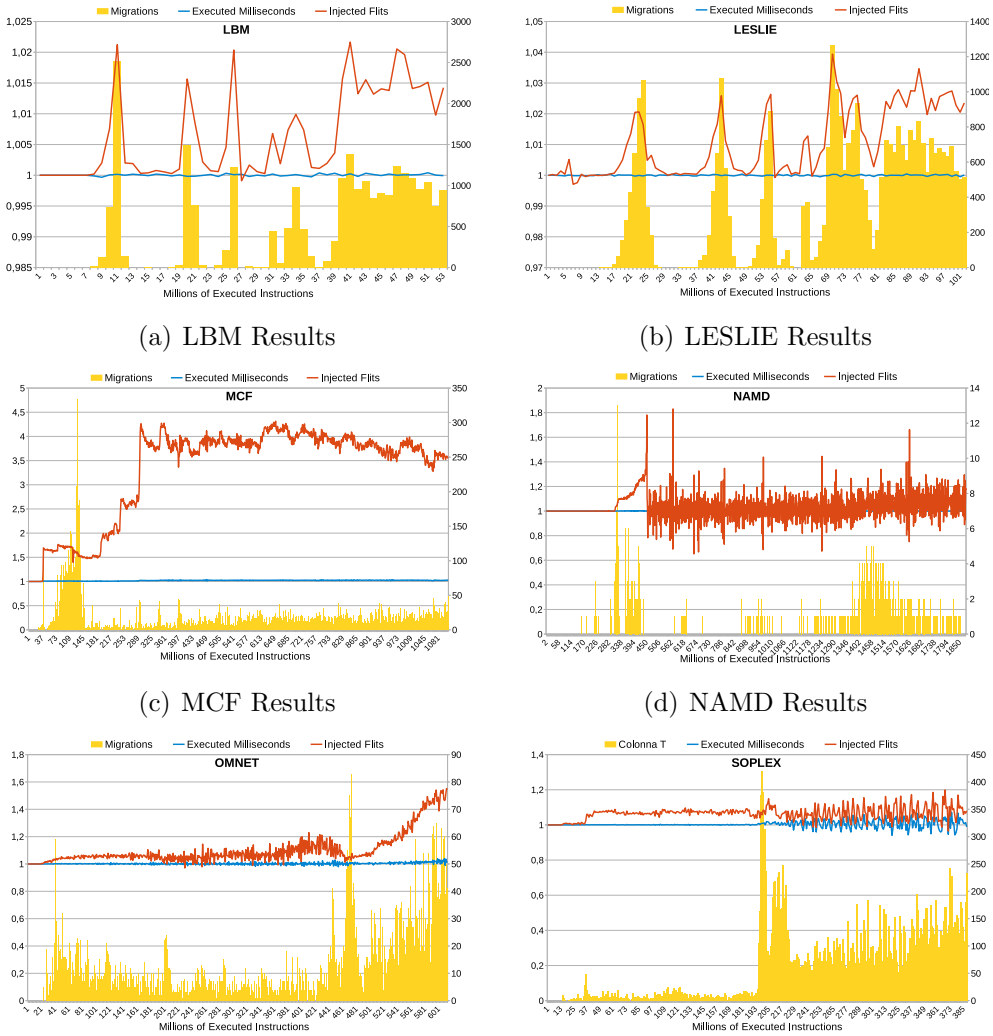


(e) H264 Results



(f) HMMER Results

Figure 5.8: Performance and injected flits results. They are evaluated considering execution time and injected flits for every million of executed instructions, **normalized to the Baseline architecture**. Blocks migrations are shown.



(a) LBM Results

(b) LESLIE Results

(c) MCF Results

(d) NAMD Results

(e) OMNET Results

(f) SOPLEX Results

Figure 5.9: Performance and injected flits results. They are evaluated considering execution time and injected flits for every million of executed instructions, **normalized to the Baseline architecture**. Blocks migrations are shown.

ASTAR, GOBMK, LBM, and LESLIE show limited broadcast penalty. Moreover the peaks of injected flits are a direct consequence of the peaks of the triggered migrations. However, considering these benchmarks, the smart broadcast achieved the objective of limiting the broadcast penalties, even if the migration benefits are still hidden. In these benchmark an high number of migrations has been observed.

SOPLEX presents a decrement of the additional injected flits with respect to the broadcast-base architecture and shows two migration phases: the first one characterized by a low number of migrations and a second one characterized by an high number of migrations. However, the first phase is more affected by the additional traffic.

HMMER presents few differences from the baseline architecture. Moreover, few migrations have been done. However, these few migration make the injected flits oscillate of the 10 % with respect of the baseline architecture.

OMNET presents a considerable increment of the injected flits when several migrations are triggered.

GEMS, H264 and NAMD show a slow increment of the injected flits and a negligible number of migrations. These results are notable because the considered benchmarks behaviours registered a low number of LLC miss. These applications show a negligible broadcast impact considering the broadcast-base architecture.

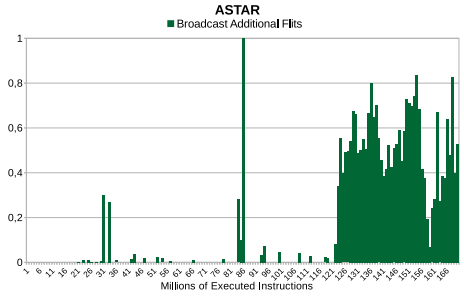
MCF and GROMACS show an important increase of the injected flits. This trend is confirmed by considering the baseline architecture and the broadcast-base one (see Section 5.3).

The increment in the injected flits is justified by the fact that the migrations change the application behaviour. However, the observed overheads are mostly caused by the additional broadcasts and not directly by migration mechanism.

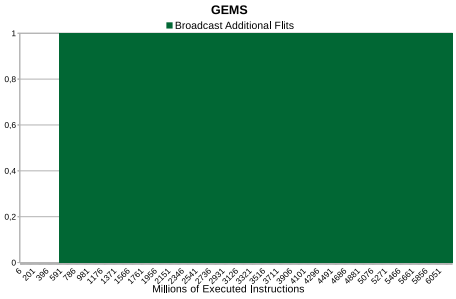
5.4.1 Additional Traffic Explanation

Considering the additional injected flit observed for the DNUCA solution compared to the baseline architecture, this section details the possible traffic sources.

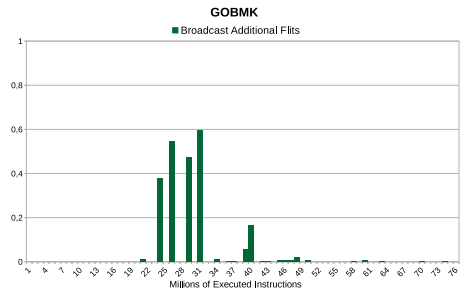
Figure 5.10 and Figure 5.11 show the percentage of the additional injected flits that are caused by additional broadcast searches, per million of executed instructions. By comparing the results in Figure 5.8 and Figure 5.9,



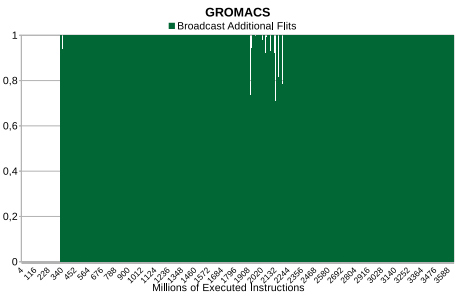
(a) ASTAR Additional Flits



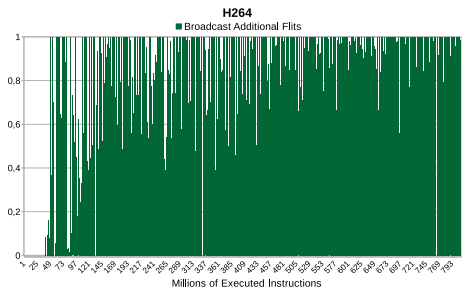
(b) GEMS Additional Flits



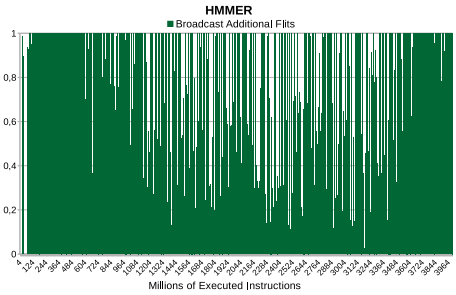
(c) GOBMK Additional Flits



(d) GROMACS Additional Flits

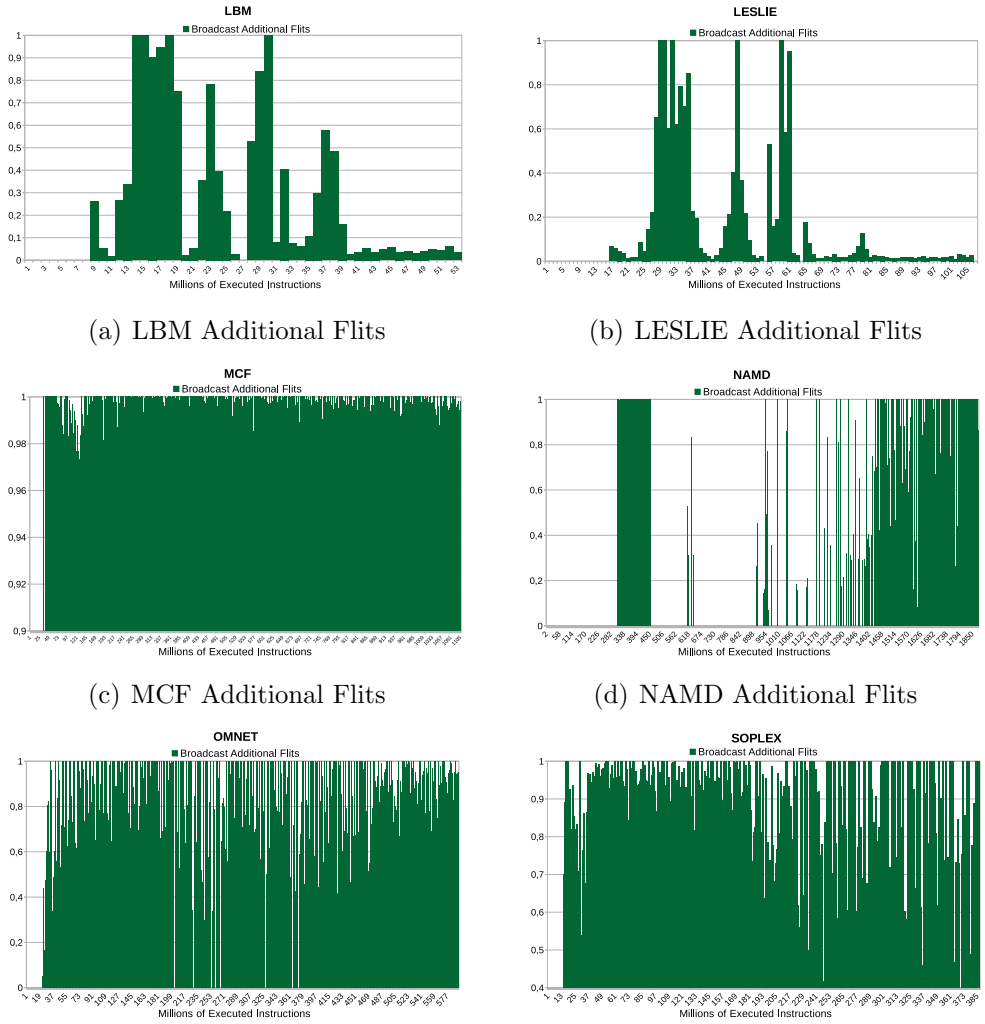


(e) H264 Additional Flits



(f) HMMER Additional Flits

Figure 5.10: Ratio of the additional injected flits caused by broadcast searchs.



(a) LBM Additional Flits

(b) LESLIE Additional Flits

(c) MCF Additional Flits

(d) NAMD Additional Flits

(e) OMNET Additional Flits

(f) SOPLEX Additional Flits

Figure 5.11: Ratio of the additional injected flits caused by broadcast searchs.

ASTAR, GOBMK, LBM and LESLIE present a limited overhead on the injected flits, that is concentrated in some instructions periods. Figure 5.10(a) and Figure 5.10(c) shows that this trend is also caused by the high number of migrations and their coherence messages. The instruction periods which present additional traffic are affected by both the broadcast and migration. However, it is clear how the migration mechanism impact is evident only with a huge number of migrations and it is much lower than the one caused by the broadcasts. These benchmarks are the ones which present a significant improvement on the broadcast-base architecture in terms of injected flits.

The rest of the analysed benchmarks present an high number of additional injected flits, despite the much lower number of triggered migrations. The additional flits are due to the additional broadcasts, that originates from an L1 miss on a previously migrated block. The trend is shown in Figure 5.10 and Figure 5.11.

It's very interesting considering how some applications, which weren't affected by the pure broadcast penalties shown in Section 5.3, present additional injected flits with respect to the baseline. Clear examples of this situation are GEMS, OMNET and NAMD.

Moreover, MCF and GROMACS present much more injected flits than the broadcast-base architecture. However, Figure 5.11(c) and Figure 5.10(d) show that they are caused nearly totally by additional broadcast searches.

Thus, the additional coherence traffic is mostly caused by the additional broadcast caused by migrations. At this point, the causes of the additional broadcasts need to be analysed.

The architecture considered in Section 5.3 does not provide a migration mechanism since the broadcast penalties were under investigation. However, the migration actions can modify the number of triggered broadcasts.Considering the broadcast with no migration support, a block search is triggered every time a request results in a LLC miss.

Considering precedent migrations, a request for a block can result in a broadcast if it has been migrated. Thus, we have to pay in terms of injected flits even if we have a hit (not in the home LLC bank) and it is an additional scenario in which a search is triggered respect the considered pure broadcast in Section 5.3.

Additional flits due to the broadcast are observed if an L1 relinquishes a migrated data that will access in the future. L1 bits are reset every time the block is replaced in L1 cache: if the benchmark is characterized by an high

number of L1 cache misses, blocks are replaced and the migration information are lost, thus triggering a broadcast action as a consequence of an L1 miss.

These problems can affect injected flits specially if the block is heavily accessed or if the application reuses data.

Let's consider the observed smart broadcast results. MCF presents a phase characterized by an high number of L1 (and L2) misses (see Section 5.2.1). This phase is characterized by an increment of injected flits (four times), despite a limited number of migrations. This is due to combined action of the LLC and L1 misses which cause broadcasts for different reasons.

GEMS presents a low number of L1 and LLC misses, instead (see Section 5.2.1). Moreover, it is characterized by a very low number of migrations. Nevertheless, it presents an increment of the injected flits with the smart broadcast and the migration mechanism. This is due to the high data reuse in GEMS. In particular, the few blocks which are not present in the L1 cache are continuously retrieved in the LLC: if they have been migrated, continuous broadcasts are triggered. Note that OMNET and NAMD share a similar behaviour.

Chapter 6

Conclusions and Future Works

In NoC based DNUCA architectures several factors influence the system performance and the broadcast/migration mechanisms efficiency.

The application behaviour is the most important one. First of all, depending on the considered application, the number of LLC misses (and the miss rate) can be very different. Moreover, considering the same application, the distribution of the misses in time can change and several phases can be observed. This heavily influence performance and in particular the number of injected flits in the NoC. In fact every LLC miss triggers a broadcast action to retrieve the block and in the worst case (the line is not in the LLC) this brings pure penalties. Only considering this situation, with no active migration mechanism, the SNUCA architecture outperforms the broadcast-based one by 6 % on average. Moreover, the broadcast causes a 43 % additional flits on average with a peak of 108 %.

These overheads are cut off by the smart broadcast mechanism, that avoids useless searches and triggers broadcast actions only if the block has been migrated.

However, the number of LLC misses is not the only factor that negatively affects the DNUCA system behaviour, causing additional broadcast searches. The number of L1 cache misses and the application temporal locality, coupled with the migration mechanism, can heavily affect the system. In fact, every time a miss occurs in the L1 cache, the block has to be searched, if it has been previously migrated. Moreover if the line is heavily accessed in LLC and the L1 cache presents an high number of misses, the number of broadcasts for the line explodes. This scenario causes more additional coherence traffic, that means a negative impact over the power consumption.

The results and the analysis carried out by the thesis can't be ignored, designing future DNUCA architectures. The search mechanism overheads are important aspects to be considered, in order to achieve global performance improvements on the SNUCA architectures. Otherwise, all the possible benefits of the blocks dynamic placement are overcome by broadcast/multicast costs.

6.1 Future Works

The presented work analyses application behaviour and required mechanisms impact on the DNUCA architectures. The smart broadcast and the simple migration policy partially limit the search mechanism overheads. However, given the proposed analysis, a stronger connection between the dynamic block placement benefits and mechanisms overheads is under investigation to provide a novel power-performance policy. In particular, additional ways of limiting the search mechanism and placing LLC data will be considered to design a policy that can dynamically move cache blocks to obtain performance benefits, with an acceptable NoC traffic.

Bibliography

[1] Rajeev Balasubramonian, Norman P. Jouppi, and Naveen Muralimanohar. *Multi-Core Cache Hierarchies*. Morgan & Claypool, 2011.

[2] J. Balfour and W.J. Dally. Design tradeoffs for tiled cmp on-chip networks. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 187–198. ACM, 2006.

[3] A. Bardine, M. Comparetti, P. Foglia, G. Gabrielli, and C. A. Prete. A power-efficient migration mechanism for d-nuca caches. In *2009 Design, Automation Test in Europe Conference Exhibition*, pages 598–601, April 2009.

[4] A. Bardine, M. Comparetti, P. Foglia, G. Gabrielli, C. A. Prete, and P. Stenstrm. Leveraging data promotion for low power d-nuca caches. In *Digital System Design Architectures, Methods and Tools, 2008. DSD '08. 11th EUROMICRO Conference on*, pages 307–316, Sept 2008.

[5] A. Bardine, P. Foglia, G. Gabrielli, C. A. Prete, and P. Stenström. Improving power efficiency of d-nuca caches. *SIGARCH Comput. Archit. News*, 35(4):53–58, September 2007.

[6] B. M. Beckmann and D. A. Wood. Managing wire delay in large chip-multiprocessor caches. In *rocs. of the 37th International Symposium on Microarchitecture*, 2004.

[7] A. Bhattacharjee and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. pages 290–301, 2009.

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower,

Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[9] J. Cebrin, R. Fernndez-Pascual, A. Jimborean, M. Acacio, and A. Ros. A dedicated private-shared cache design for scalable multiprocessors. 2016.

[10] S. Corbetta, D. Zoni, and W. Fornaciari. A temperature and reliability oriented simulation framework for multi-core architectures. In *2012 IEEE Computer Society Annual Symposium on VLSI*, pages 51–56, Aug 2012.

[11] W.J. Dally and B.P. Towles. *Principles and practices of interconnection networks*. Elsevier, 2004.

[12] R. Das, S. Eachempati, A.K. Mishra, V. Narayanan, and C.R. Das. Design and evaluation of a hierarchical on-chip interconnect for next-generation cmps. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 175–186, Feb 2009.

[13] J. Duato. A new theory of deadlock-free adaptive routing in wormhole networks. *Parallel and Distributed Systems, IEEE Transactions on*, 4(12):1320–1331, Dec 1993.

[14] J. Duato. A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks. *Parallel and Distributed Systems, IEEE Transactions on*, 6(10):1055–1067, Oct 1995.

[15] J. Duato and T.M. Pinkston. A general theory for deadlock-free adaptive routing using a mixed set of resources. *Parallel and Distributed Systems, IEEE Transactions on*, 12(12):1219–1235, Dec 2001.

[16] J. Flich, G. Agosta, P. Ampletzer, D. A. Alonso, A. Cilardo, W. Fornaciari, M. Kovac, F. Roudet, and D. Zoni. The mango fet-hpc project: An overview. In *Computational Science and Engineering (CSE), 2015 IEEE 18th International Conference on*, pages 351–354, Oct 2015.

[17] P. Foglia, F. Panicucci, and M. Prete C. A. an Solinas. Analysis of performance dependencies in nuca-based cmp systems. pages 49–55, 2009.

[18] Fazal Hameed, Lars Bauer, and Jorg Henkel. Dynamic cache management in multi-core architectures through run-time adaptation. 2012.

[19] S.M. Hassan and S. Yalamanchili. Centralized buffer router: A low latency, low power router for high radix nocs. In *Networks on Chip (NoCS), 2013 Seventh IEEE/ACM International Symposium on*, pages 1–8, April 2013.

[20] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[21] M. Huang, M. Mehalel, R. Arvapalli, and S. He. An energy efficient 32nm 20 mb l3 cache for intel xeon processor e5 family. In *Proceedings of the IEEE 2012 Custom Integrated Circuits Conference*, pages 1–4, Sept 2012.

[22] L. Jianhua, A. Xin, O. Yiming, and Wangwei. Thread progress aware block migration for dynamic nuca. pages 422–426, 2016.

[23] C. Kim, D. Burger, and Keckler S. W. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *SIGOPS Oper. Syst. Rev., vol 36*, pages 211–222, October 2002.

[24] J. Lira, C. Molina, and A. Gonzalez. Hk-nuca: Boosting data searches in dynamic non-uniform cache architectures for chip multiprocessors. In *IEEE International Parallel Distributed Processing Symposium*, pages 419–430, 2011.

[25] Mario Lodde and Jos Flich. Runtime home mapping for effective memory resource usage. 2014.

[26] M. Rosenblatt. Central limit theorem for stationary processes. In *Proceedings of the Sixth Berkeley Symposium on Mathematical Statistics and Probability, Volume 2: Probability Theory*, pages 551–561, Berkeley, Calif., 1972. University of California Press.

[27] Sheldon M. Ross. *Introduction to probability and statistics for engineers and scientists (2. ed.)*. Academic Press, 2000.

[28] Keun Sup Shim, Miezsko Lis, Omer Khan, and Srinivas Devadas. Thread migration prediction for distributed shared caches. In *IEEE Computer Architecture Letters, Vol. 13, No. 1*, pages 53–56, January-June 2014.

[29] D. Shirshendu and K. Hemangee. Exploration of migration and replacement policies for dynamic nuca over tiled cmps. pages 1–6, 2015.

[30] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.

[31] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ISCA '86, pages 414–423, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[32] F. Trivino, F. Alfaro, J. Sanchez, and J. Flich. Noc reconfiguration for cmp virtualization. pages 219–222, 2011.

[33] A.N. Udipi, N. Muralimanohar, and R. Balasubramonian. Towards scalable, energy-efficient, bus-based on-chip networks. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, Jan 2010.

[34] Kartheek Vanapalli, Hemagee K. Kapoor, and Shirshendu Das. An efficient mechanism for dynamic nuca in chip multiprocessors. 2015.

[35] A. Varma, B. Bowhill, J. Crop, C. Gough, B. Griffith, D. Kingsley, and K. Sistla. Power management in the intel xeon e5 v3. In *Low Power Electronics and Design (ISLPED), 2015 IEEE/ACM International Symposium on*, pages 371–376, July 2015.

[36] D. Zoni, S. Corbetta, and W. Fornaciari. Hands: Heterogeneous architectures and networks-on-chip design and simulation. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '12, pages 261–266, New York, NY, USA, 2012. ACM.

[37] D. Zoni, J. Flich, and W. Fornaciari. Cutbuf: Buffer management and router design for traffic mixing in vnet-based nocs. *IEEE Transactions on Parallel and Distributed Systems*, 27(6):1603–1616, June 2016.

[38] D. Zoni and W. Fornaciari. Modeling dvfs and power gating actuators for cycle accurate noc-based simulators. *Journal of Emerging Technologies in Computing Systems*, pages 1–15, 2015.

[39] D. Zoni, F. Terraneo, and W. Fornaciari. A dvfs cycle accurate simulation framework with asynchronous noc design for power-performance optimizations. *Journal of Signal Processing Systems*, pages 1–15, 2015.

[40] Davide Zoni, Federico Terraneo, and William Fornaciari. A control-based methodology for power-performance optimization in nocs exploiting dvfs. *Journal of Systems Architecture*, pages 1–15, 2015.