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

A Novel Cache Coherence Protocol to Adaptively Power Gating LLC Banks in Tile-based Multi-cores

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

Master Thesis of:
Luca COLOMBO
Student n. 818211

Academic Year 2015-2016

Estratto in Lingua Italiana

Nell'epoca dei multi-core, l'ultimo livello di Cache (LLC) è una risorsa condivisa critica dal punto di vista energetico, prestazionale e in termini di spazio che occupa all'interno del chip. La tendenza ad aumentare la dimensione della cache per ridurre la contesa tra diverse applicazioni in esecuzione ha messo in luce il partizionamento della cache come tecnica per allocare efficientemente e dinamicamente lo spazio alle singole applicazioni. Infatti il sottoutilizzo della cache rappresenta un problema attuale, motivando lo spegnimento di alcune sue parti al fine di salvare energia.

Tale soluzione è possibile con cache monolitiche, mentre diventa impraticabile con cache fisicamente divise in tanti piccoli banchi, per cui la riduzione del consumo può essere eseguita solo a granularità di banco; a tale livello si rende necessaria l'uso di un'architettura DNUCA (Dynamic Not Uniform Cache Access), quindi un nuovo protocollo che assicuri la coerenza della cache.

La tesi propone *FlexiCache*, una nuova architettura per le cache in grado di spegnere e accendere banchi di cache. Viene presentato un nuovo protocollo di coerenza basato sul MESI, che supporta la possibilità di spegnere banchi di cache. Inoltre, vengono discusse una politica per il risparmio energetico, così come la logica addizionale per ottenere le informazioni di sistema. In particolare, una dettagliata analisi dei banchmark permette di progettare un'architettura agnostica dall'applicazione. Per ultimo, viene fornita una completa analisi di fattibiltà, per rafforzare ulteriormente la validità della soluzione proposta.

FlexiCache è stata valutata su un'architettura a 16 core eseguendo un sottoinsieme delle applicazioni appartenenti alla suite *Parsec3.0*. *FlexiCache* richiede in media il 30% in meno di energia rispetto a un protocollo MESI tradizionale, con un degrado prestazionale minore del 5%.

Abstract

In the multi-core era, the Last Level Cache (LLC) is a critical, shared, platform resource from the performance, the energy and the chip area viewpoints. The trend of increasing the LLC size to reduce the contention between different running applications highlights the cache partitioning schemes as viable technique to efficiency and dynamically allocate the LLC space to the application from the energy-performance perspective. Although the cache underutilization represents a standing issue that motivates the power-off of the portion of the unused LLC to save energy.

Such a solution is viable within monolithic LLC design, while becomes unfeasible when the LLC is physically split in multiple small banks for which the power gating can only be exploited at bank level. By enforcing the power gating mechanism at bank level, a Dynamic Not Uniform Cache Access (DNUCA) architecture is enforced thus imposing a novel coherence protocol to ensure the cache coherence.

The thesis proposes *FlexiCache*, a novel cache architecture that allows to dynamically power-OFF and -ON different LLC banks. A novel MESI-based inclusive coherence protocol is presented to support the LLC bank powering-off capabilities. Moreover, the power gating policy as well as the additional logic to gather the platform information are also discussed. In particular, the in depth benchmark analysis allows to design an application agnostic architecture. Last, a complete feasibility analysis is provided to further strengthen the validity of the proposed solution.

FlexiCache has been validated on a simulated 16-core architecture executing representative applications from the *Parsec3.0* suite. *FlexiCache* requires 30% less energy, on average, with respect to the traditional MESI protocol with an average performance overhead within 5%.

Acknowledgments

To my family, my real strength. Only one life will not be sufficient for me to thank my parents, who have gave me the opportunities to do everything I have wanted, who have always made me feel a priviliged person and who have gave me a good life. I hope with my constancy and my work to have rewarded in some way their choices and sacrifices made as parents. These pages belong also to them.

To my sister Valentina, for her essential presence.

To my grandmother Palmira, who I think today would be proud of me.

To Mauro, Fabio and Andrea, more than mates. In you I have found a true and sincere friendship.

To Simone, for his valuable advices.

To Davide, Matteo, Giorgio, Lara and Fabrizio, for their support.

To Marco, for his costant interest and all the talks at gym.

To Davide, for his hard teachings and his guidance.

To Luca, Giuseppe, Simone, Federico R., Michele, Francesca, Stefano, Nicholas, Federico T. and all the people of the HEAP Lab, who have made me feel part of an entire research group. Thank you all guys for the very good moments and the laughs together.

To professor Fornaciari, for the opportunity of working in the branch of research.

To all the people who, even once, have shown true interest during my way.

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

The multi-cores represent the de-facto standard to execute multiple applications and eventually enhance the platform utilization while imposing the share of several computing resources. For example the Last Level Cache (LLC) is usually shared between all the CPUs, thus easily becoming the performance bottleneck of the system. To this extent two different multi-core design trends emerged to limit the performance overheads induced by a shared LLC. First, the LLC size increased dramatically to make space for data of different concurrent applications. Second, the LLC design shifted from a monolithic LLC bank to multiple physically split LLC banks within a single address space. This latter trend reduces the access contention to the single LLC controller and allows the implementation of smarter data placement policy to keep each cache lines closer to the CPU that is using it.

The increse of the LLC size demands for fresh cache management strategies to optimally use the augmented resource. In this scenario, *the cache partitioning scheme* highlights a viable methodology to adapt the LLC cache space allocated to different applications that competes on the utilization of the shared LLC. However, several proposals in the open literature highlight the underutilization of the LLC [5, 18], thus the power gating mechanism is exploited to power-off the unused cache slices to limit the leakage power while preserving the overall system performance. The power gating of the LLC at slice level represents a solution until the architecture has a single monolithic LLC bank, while the current design trend is moving towards physically split LLCs [1] to keep data closer to each CPU and reduce the pressure on the single LLC interface / port. In such a scenario, a DNUCA approach is required to search data in the LLC. Moreover, a novel coherence protocol is

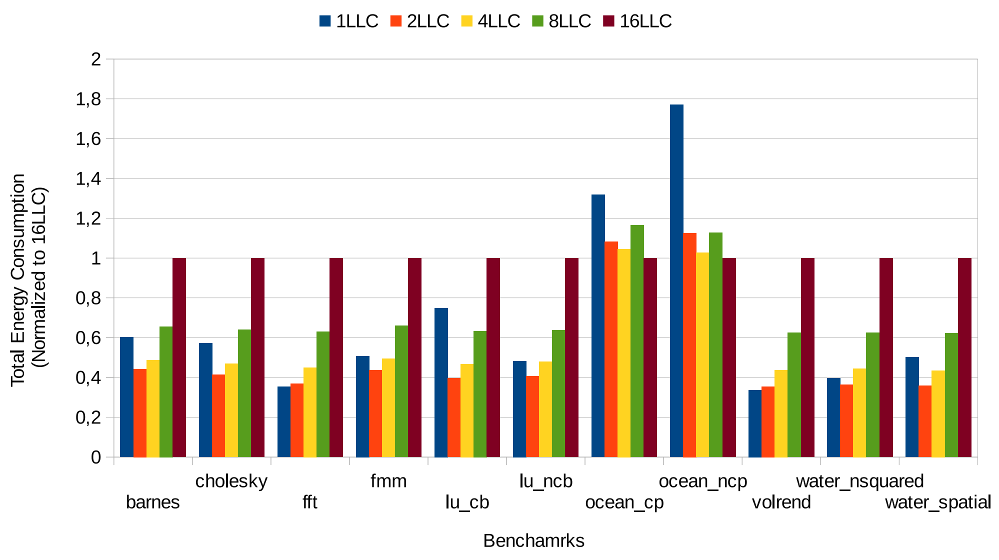


Figure 1.1: Behavior of all benchmarks for what concerns the Total Energy with different available LLC banks.

required to support the power gating of any LLC bank without violating the coherence properties.

The thesis presents *FlexiCache*, a novel cache architecture that allows to dynamically power-OFF and ON the LLC banks to save energy, while keeping the overall system performance close to the performance level that is achieved using a traditional coherence protocol, i.e., without power gating capabilities. The motivational example as well as the possible energy-performance benefits of such an architecture are discussed in Section 1.1. Section 1.2 highlights the novel contribution of the research while the organization of the manuscript is described in Section 1.2.

1.1 Problem Overview

The possibility to reduce the LLC size by power gating different LLC banks greatly improves the energy consumption while negatively affecting the performance. This section discusses the energy and performance metrics when different subsets of LLC banks are power gated. In particular, a subset of the Splash2x [31] are simulated on a 16-core architecture with 16 LLC banks shared to all the cores where each core implements a private L1 cache.

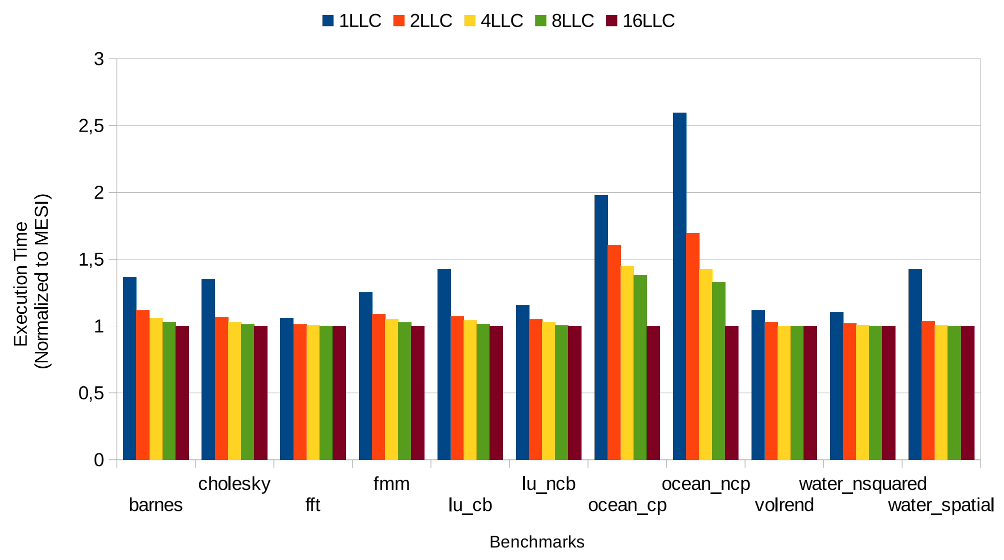


Figure 1.2: Behavior of all benchmarks for what concerns the Execution Time with different available LLC banks.

Figure 1.1 reports the energy consumption of the multi-core per each considered benchmark. In particular, each benchmark has been simulated with a different number of active LLC banks. For each simulation the number of active LLC banks is kept for the entire simulation. For each simulation the energy is normalized to the one consumed with all the LLC banks active. In general, the reduction of the active LLC banks positively impact the consumed energy. However, few benchmarks report an higher energy when a single LLC bank is used instead of two or four banks due to the dramatic increase in the execution time then increase the total consumed energy of the benchmark.

As expected the reduction of the active LLC banks negatively impact the execution time as reported in Figure 1.2. For each benchmark the performance increases with the number of active LLC banks and a huge performance drop is observed when a single LLC bank is active. Moreover, each application highlights a specific performance overhead when executing with a reduced number of active LLC banks. In general, a negligible performance overhead is observed when the 8LLC bank configuration is used in place of the 16LLC one. Although, *ocean_np* and *ocean_ncp* show a strong relation between the performance metric and the number of active LLC banks.

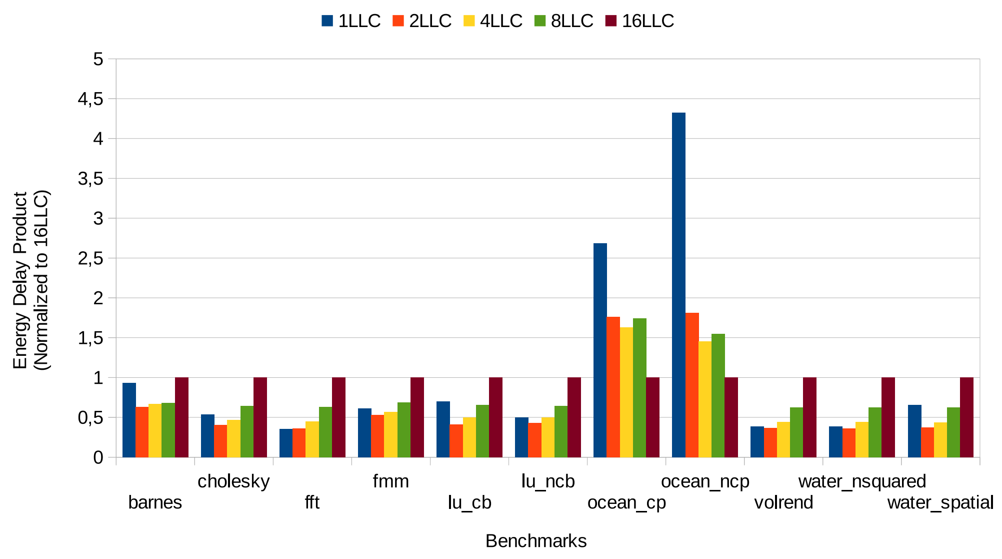


Figure 1.3: Behavior of all benchmarks for what concerns the Energy Delay Product with different available LLC banks.

The observed data coupled with the contrasting optimization directions for the energy and the performance eventually underpin the possibility of an optimal LLC configuration, i.e., number of active LLC banks, to get the optimal energy-performance trade-off. Figure 1.3 shows the Energy Delay Product (EDP) for the same benchmarks and the same LLC configurations already discussed. An optimal LLC configuration that minimizes the EDP emerges for each application and seems to be application dependant. In the rest of the manuscript the proposed architecture allows to optimally allocate the LLC configuration to minimize the EDP product using an application independent decision algorithm. In particular, the benchmark analysis will highlight that the application behavior can be shadow within few architectural parameters that makes the decision algorithm independent from the specific running application.

1.2 Goals and Contributions

The thesis proposes a novel coherence protocol that allows to dynamically resize the LLC size by power gating the LLC banks with 3 major contributions:

- a novel DNUCA coherence architecture based on the MESI coherence protocol, which allows to switch on/off LLC banks without any restrictions to optimize the energy-performance trade off.
- a non-blocking handshake mechanism for power gating LLC banks given a certain LLC configuration, i.e. the power state of all the LLC banks of the system; power commands do not freeze the system, that can execute the application during the LLC reconfiguration.
- a scalable and application independent policy architecture, that takes into account architectural metrics, supported by a signaling network with low overhead (i.e. few messages).

A 16-cores architecture is used, executing applications from the PARSEC3.0 benchmark suite [31]. The obtained results highlight the benefit in energy savings of 30% on average, while impacting on performance with a less than 5% degradation.

1.3 Thesis Structure

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

Chapter 2

Background and State of the Art

This chapter describes the background of the work, to ease the reading of the following chapters. In Section 2.1 is described the Network-On-Chip interconnection, while Section 2.2, Section 2.3 and Section 2.4 discuss basics about Cache architecture and hierarchy, with a brief description of MESI Coherence Protocol. Finally, Section 2.5 suggests an overview of State of the Art existing energy saving techniques for LLC.

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 [9], concentrated mesh [2], hybrid bus based [10, 30] and high radix [17].

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 [13, 11, 12], 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. 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 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 Coherence Protocols Basics

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 acknowledgments.

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 [29] 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.

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 biggest 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).

2.3.1 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 multicast support.

Figure 2.1 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.

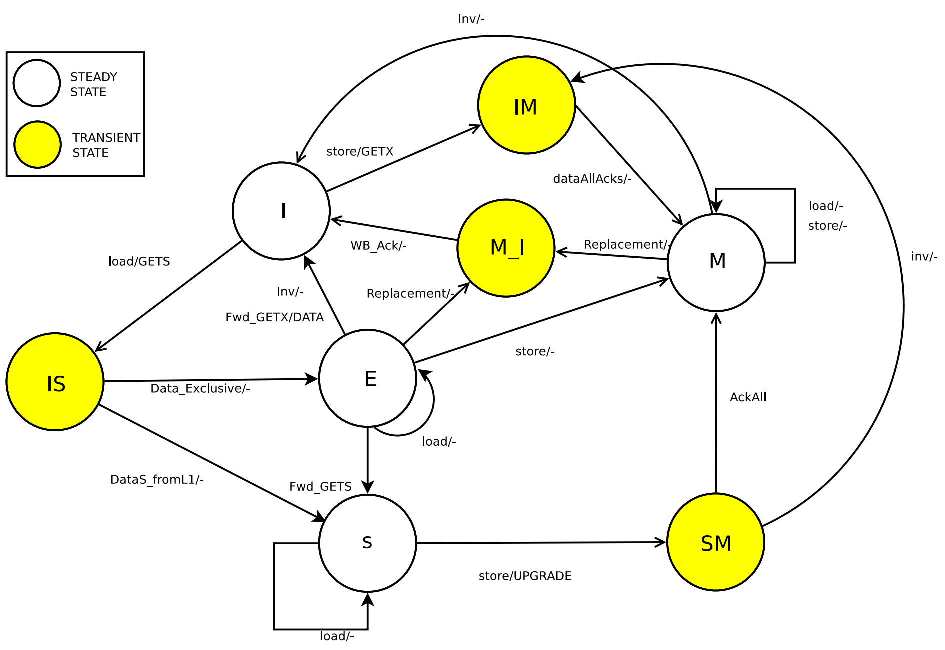


Figure 2.1: Simplified typical implementation of the MESI protocol; L1 cache FSM, including main transient states.

- 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 acknowledgment.

Considering the L2 cache controller, the stable states only are detailed below. The interested reader can find a complete description of the L2 cache controller in [26].

- 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.

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 [19].

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.

[19] 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 [15] 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.

2.5 Energy Saving Techniques in the LLC

[5] presents an evaluation of UCAs (Uniform cache architecture) and NUCAs (Non uniform cache architecture), from the performance and the energy consumption view point. After explaining how NUCA (both Static, where lines are mapped in a single predetermined bank, and Dynamic, where a line can dynamically migrate in different banks) architectures are fundamental for reducing the access latencies to L1's lower level caches and how they outperform UCA ones, they point out the large portion of space that in modern microprocessors L2 caches occupies, and consequently the big impact in the total power consumption they have. This work focus on a more important key point, that motivates also our work: in NUCA architectures energy consumption is dominated by static component, due to leakage currents; especially in D-NUCA ones, despite of the major traffic and data movement generated, energy consumption is in any case dominated by the static part. So, in the context of cache energy savings, the reduction of static component deserves most of the attention. Nevertheless, this work focuses on a single processor system with a single L2 cache, internally divided in multiple banks connected by a switched network; so they do not consider an architecture where the L2 is physically partitioned and placed in different points, like the tiles in the Network-On-Chip which we studied. Furthermore, in the context of D-NUCA architectures, they consider movement and mapping of

data only in a group of banks and not, in a general way, in any of the bank in the network: in our work we assume that a data can be placed in any of the portion of the L2, depending on the configuration of the system.

[7] repeats as the most important component of chip power in Chip Multi Processor is that of last level cache banks, due to the big portion of area they occupy in the chip. In this work it is explained that the two most important reduction of cache power techniques are of two types: controlling power supply or resizing directly the cache. Since different applications doesn't need the same space of memory (i.e. the same number of LLC banks), this work focused on techniques that switch off the less utilized banks, to reduce the total power consumption of the LLC cache. In particular, they start working from an existing technique: a policy that monitors the performance degradation of the system and that, based on a threshold value, decides if a bank (in particular the less utilized) can be shutdown. This technique causes that requests to the powered off bank are forwarded to another active one, called the "target" bank. Since this target bank has to manage the additional load of the powered off bank, this paper tries to find an optimization for improving its performance and by reducing forwarding cost. In particular, once a bank is decided to be shutdown, the valid blocks of the selected bank are relocated to its target bank and future request to this bank are forwarded. In our work we decide to flush the cache to be shutdown and change the vision of active L2s to map blocks, instead of forwarding requests. So, eventual blocks that have to be reloaded in cache are distributed among all the available switched on banks and they do not stress a particular target bank. Doing so, we do not need any type of optimization for the target bank to better manage the requests of one or more switched off bank. Moreover, our technique can switch on/off different slices of L2, depending on the application behavior and of its phases.

[14] tries to improve the idea of "accounting cache" in order to save dynamic but also leakage power of a cache. The idea under accounting cache is to split the ways of the cache in two partitions and to access in first place only one of them, accessing the other only in case of a miss in the first partition. Doing so, dynamic power due to access in the second partition is avoided in a number of cases, but both partitions are kept active every time, not saving leakage power. This mechanism is also "phase adaptive", i.e. the number of ways inside each partition can change dynamically depending on the needs of the application: by taking into account a cost function based

on dynamic power consumption, the best configuration is selected. So, the improvement offered by this work is to make accounting cache more efficient, considering and trying to save also leakage power: this is done by putting the secondary partition in a drowsy state, powered at a lower level with respect to the primary partition. Data are searched in the primary partition and, only if not found then are searched in the second one; if data is found then data is swapped with the first partition, otherwise an access to the lower level of memory is needed. In general, this mechanism can add extra latency, due to the fact that the second partition is accessed after the first one. The cost function is computed with some extra logic and consider also, phase by phase, the leakage power consumption. Again, this work focuses on the energy/power optimization of a cache at a way-granularity level. The scenario considered is that of a single processor with a UCA private L2 cache and a L3 cache, not taking into account a TCMP architecture, where L2 banks are physically splitted.

[24] proposed a technique that control resizing of the cache at a way-level granularity, trying to allocate the right portion of space to each thread for improving energy efficiency and performance gain. The idea of this work is to, starting from the working set size estimation based to the cache accesses monitored in a certain period, by capturing misleading behaviour in the trend of accesses, allocate the right number of ways to running threads. This is made through a majority voting among locality assessment results of several short sampling periods (called voting periods): this permits to detect exceptional behavior in the accesses and to ignore them in the decision of number of ways to be allocated. The mechanism makes use of a sample period, during which it collects cache accesses, and of an adaptation opportunity, in which can decide effectively to change the number of allocated ways to each thread. Three functions are used to support this second step: the first that decides if a thread effectively needs more ways or not; the second that decides to resize cache based only on the current signal or accounting also for the past signals; the third and last considers to change the allocation of ways when all of them are yet used and some thread requires more resources. Evaluation is made by considering one only L2 cache and an incremental number cores in the chip (1,2,4,6), so not considering a tiled CMP architecture. Also, the focus is an optimization in the performance, not inspecting the power/energy problem.

[23] studies the main problem of partitioning the shared last-level on-chip

cache among multiple competing applications. In this paper it is proposed a partitioning mechanism based on utility, in the sense of benefit, and the misses that the application has. The claim of this investigation is that it makes sense to partition the cache based on how much the application is likely to benefit from the cache rather than the application's demand for the cache. To partition the cache based on application's utility for the cache resource, they propose a Utility-Based Cache Partitioning (UCP). This mechanism exploits a monitoring circuit which is able to collect information about utility of cache resource for all the running applications. This circuit is separated from the shared last-level cache, allowing it to collect informations about every application for all the ways in the cache, independently of the contention generated by other applications on other cores. Informations collected by this circuit is then used by the partitioning algorithm to decide the amount of cache allocated to each competing application. This circuit uses an Auxiliary Tag Directory, which has the same associativity of the main tag directory of the main tag directory of the shared cache, and hit counters of the ways, one group for all the sets for optimizing the area overhead. Overhead are then additionally reduced by sampling only few sets instead of all of them. This work consider a bus-architecture with only one single processor and a single L2 shared cache among all the cores, so it does not analyze a tile chip multi processor architecture where the shared cache is physically splitted in slices. Moreover, it does not study the energy/power optimization in partitioning the cache, but it focuses only on the performance; finally, the explained mechanism need less than 2kB of more space, due to all the support structures to the mechanism.

[25] 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. TD-NUCA 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 TDNUCA 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 to the MCH, so the block can't be migrated closer than to the manhattan-closest homebank. Our solution is a completely DNUCA one, where a block can reside potentially in any bank of the chip; our multicast search can space from one single bank (optimal case) to the entire NoC (bad case). Moreover, this work does take into account only a performance analysis, not proposing anything to save power.

[4] proposes a solution which adapt the number of active ways w.r.t the needs of the application. This mechanism, called way adaptable D-NUCA (WA-DNUCA), mantains the same performance of a UCA and S-NUCA scheme, while improving power savings. It switches on and off ways of a d-nuca cache as a function of the associativity level needed by the execution phase of the application: an algorithm tries to predict the working set size of the application using two counters that consider hits in the first and the farthest powered-on way. Every a certain number of hits, the ratio of the counters is computed and evaluated between thresholds in order to decide the optimal cache configuration for the next phase of the application (if switch on/off a way or stay in the current configuration). This prediction mechanism can be implemented in hardware with three counters plus the logic for the algorithm. The logic is said to be embedded in the cache controller. Thresholds are heuristically computed based on the average behavior of the considered benchmark.

[3] is strictly related to the ones presented by [4]. In particular the same DNUCA configuration is considered and the mechanism proposed in [4] 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.

So for both [4] and [3], these work consider a single L2 cache bank, made of 128 banks divide in 16 rows and 8 columns, not physically splitted in slices; moreover, the power optimization is proposed at a way-granularity level. Working-set-size estimation is made trough a thresholds based algorithm, while our policy for system reconfiguration does not take into account any type of threshold. A multi-core scenario is evaluated, but still considering one bank of L2 internally splitted and considering a bus-based interconnection, not considering for example a Network-on-Chip.

[18] analyzes and examines policies and implementations for reducing the leakage power of a cache, by turning off and invalidating lines when they contain data that probably will not be reused. The basis of the approach is to consider the cache line usage: cache lines tipically firstly see a burst of accesses and then a period of dead time between the last access and the time when a new data is loaded in. The idea of this work is to switch off the line during this dead period, so that leakage can be reduced. Some policies are proposed for this purpose: the first is a time-based, called cache decay, which

switch off a line after an established number of cycles elapses since the last access; each line is turned on until the static energy it has dissipated since the last access is equal to the dynamic energy that would be dissipated if turning the line off induces an extra miss. Switch off of a line is realized through the "gated Vdd" technique while recency of a cache line's accesses is realized with binary counter, which is reset every time a line is accessed. If no accesses to the cache line occur and it reaches its maximum value, than it means that dead time is elapsed and so the cache line can be switched off. Since decay intervals should be in the range of thousands of cycles and such large time intervals are too much large to be counted by counters, a hierarchical counter mechanism, where counters ticks at a much coarser level and a single global cycle counter set up to provide the ticks for smaller counters of the cache-lines, is used. Another implementation use a capacitor to store the recency of a cache line's access, by grounding the capacitor every time the line is accessed and so discharging it. Adaptive implementation of decay cache is also proposed, in which the decay interval is set initially with a value (also wrong) and then is modified according to the behavior of the cache line. Multiple levels of cache hierarchy and multiprogramming scenarios are also cited. However, the main analysis evaluates a unique block for L2 cache and focuses on a cache-line granularity switching on/off method.

[28] proposes a mechanism called Cooperative Partitioning. This mechanism aims to partitioning the Last Level Cache in Chip MultiProcessors at runtime, trying to reduce both dynamic and static energy and maintaining performance. It forces data owned by each core to be way-aligned among all the sets: energy savings are achieved in the dynamic part, because ways are accessed only by the core that owns them effectively, and in the static part, because whole ways unaccessed by any of the core are turned off. For doing this, a component monitors cache usage and establishes the partition of cache that fits demand of each core; then, this informations are used to effectively reconfigure the partitions. Two additional registers for each way contains access permission of that way for each of the core: these bits helps enforcing the cache partitioning by restricting accesses of cores to only the ways that they own; they allow energy savings because cores access only ways for which they have permission; they enable to turn off way whose permission are reset, meaning that none of the cores owns that way. Instead, reconfiguration of the cache is done through a technique called "cooperative takeover"; in this technique, exploiting informations of the two registers mentioned above, two

cores (named the ’donor’ and the ’recipient’) cooperates together to transfer the ownership of a certain way. In this scheme, an additional bit vectors is used for ensuring to flush data towards the memory of the older owner of the way. Two and Four-cores system are evaluated with the L1s private to cores and a single shared L2 cache, connected together with a bus. Again, this works do not consider the scenario of a single L2 physically splitted and the granularity which is considered for energy savings is that of the ways.

Chapter 3

The FlexiCache Architecture: Coherence Protocol and Policy Architecture

This section presents *FlexiCache*, a scalable, inclusive coherence protocol architecture to dynamically resize the LLC for energy-performance optimization. The implementation is application agnostic and highly independent from the architectural parameters. However, the exploited reference architecture used as a starting point to highlights the necessary changes introduced with *FlexiCache* is depicted in Figure 3.1. The reference architecture represents a generic tile-based multi-core with a private L1 cache level and a shared but physically split LLC. A generic interconnection fabric without global ordering capabilities is considered, to increase the re-usability of the coherence solution. In particular, the described *FlexiCache* implementation exploits a Network-on-Chip as the interconnection fabric between the L1s the LLCs and the memory controllers. *FlexiCache* is made of four components that work together to allow a dynamical resize of the LLC space by powering-OFF and -ON the LLC banks to optimize the energy-performance trade-off. The *Multicast Mechanism*, as described in Section 3.1, supports the DNUCA architecture that emerges from an SNUCA system when the cache banks can be switched OFF and ON at run-time. The *LLC Prefetch Scheme* (see Section 3.3) complements the *Multicast Mechanism* to shadow the additional overhead to locate the cache block within the LLC. In particular, the *Multicast Mechanism* tries to speculatively fetch the required cache line from the main memory without breaking the coherence properties. The *Handshake*

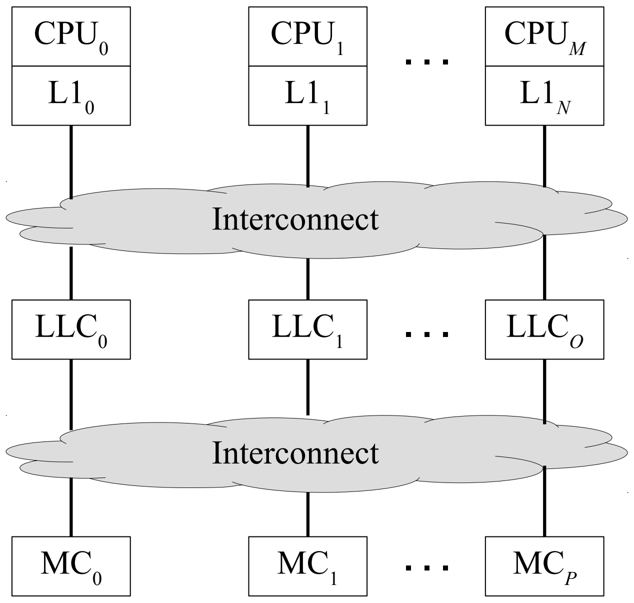


Figure 3.1: Baseline architectural overview, made of M CPUs, N private L1s, O physically splitted LLC banks and P Memory Controllers.

Mechanism detailed in Section 3.2 is a non-blocking protocol integrated in the coherence architecture that allows to dynamically reconfigure the LLC by powering OFF and/or ON the target LLC banks. It is worth noticing the *Handshake Mechanism* implements a non-blocking protocol that allows the system to continue the application executing within an LLC reconfiguration stage. Last, the *Policy Architecture* implements the actual decision policy and the additional logic to gather the required system information to compute the power commands, i.e., ON OFF, as well as the target LLC banks. It is worth noticing the *Policy Architecture* makes *FlexiCache* an application agnostic solution for inclusive cache coherent multi-cores with physically split LLC banks. Moreover, an in depth deadlock freedom methodology is described in Section 3.5 while the related results are devoted in Section 4.2.

3.1 Multicast Mechanism

The multicast mechanism enables the DNUCA support for the baseline MESI architecture, thus allowing to map each cache line in the LLC. This aspect is of primary importance to support the proposed cache architecture, that can

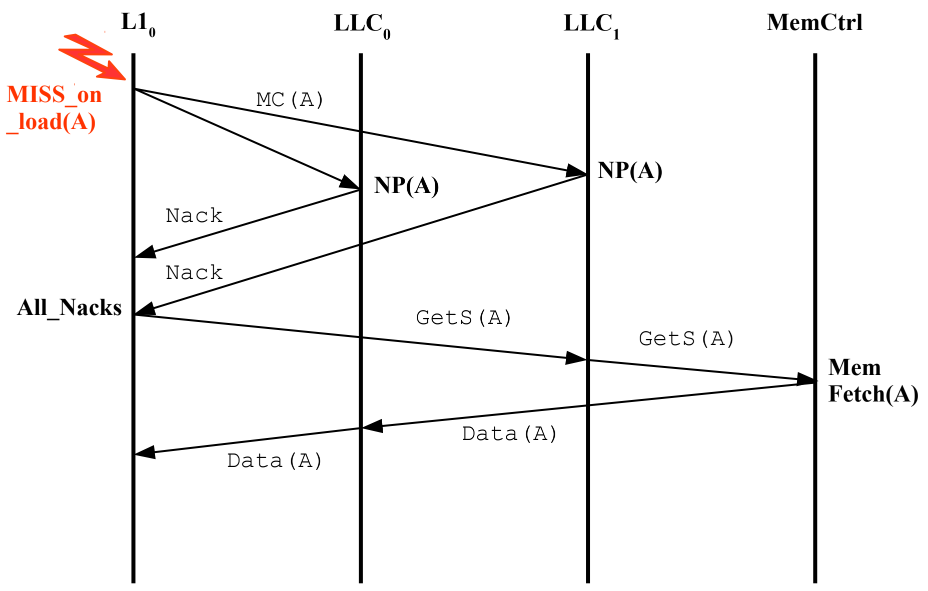


Figure 3.2: Behavior of the L1 Multicast mechanism for supporting research of cache line in lower level of cache, case data NOT present.

dynamically switch on and off different LLC banks. In particular, Multicast mechanism is triggered at the L1 on a Miss event and a MC request is propagated to the LLC active banks, seeking for the required data. The response to the Multicast can be the required data sent by the LLC bank that owns it or all Nacks. In the latter case a unicast request is issued to the target LLC bank as discussed in the rest of the section.

Multicast has been integrated in the MESI cache coherence protocol with changes in the L1 and LLC controllers, while no changes are required for the Directory Controller. In particular, each cache line in the L1 has been augmented with few bits used to store the LLC bank that owns the cache line itself. These bits are meaningful if and only if the corresponding validity bit is set, thus allowing other requests for the same cache line to be served without issuing other multicast requests. In the case of a 4x4 topology this bits are precisely equal to 4, so that it is possible to identify every of the 16 slices of LLC cache where the block can be present; in general these bits are equal to the logarithm of the number of active LLC banks. When the cache line is evicted due to an Invalidation or a Replacement command, these bits are reset, causing the issue of multicast request for future Load/Store requests on that line. Each new L1 miss triggers a multicast action, eventually adding itself to the LLC target requestor register if a multicast transaction is already in progress for the same line.

In Figure 3.2 details the basics of the implemented multicast Mechanism. Each LLC bank issues a NACK response since data A is not present in LLC. The L1 keeps track of all incoming responses and eventually issues a unicast request if all the responses to the Multicast are NACKs. The target LLC of the unicast request is chosen, according to the address of the cache line and the active LLC banks at that moment in the system. Since no bank has the data, the one that receives the unicast request issues a fetch request of the data to memory and, after receiving it, it forwards to the requesting L1.

In Figure 3.3 is described a different situation. In the same initial scenario of Figure 3.2, one LLC bank has the requested data. The one which owns the data verifies its presence and the Multicast request is managed following the transitions and behavior established by the base MESI protocol. Since a LLC bank has the data, sooner or later the L1 will receive it. In this second case, the L1 can receive the data after (case '1' in Figure 3.3) or before receiving the NACKs from all the remaining banks (case '2' in Figure 3.3); no matter the order in which responses (data itself or NACK) are received, the L1 waits all the responses before closing the Multicast transaction.

3.2 The Handshake Protocol

The Handshake Mechanism (HM) defines the extensions to the cache coherence protocol to change the LLC configurations. In particular, the LLC configuration is defined as the state of each LLC bank that can be in ON or OFF, i.e., power gated, state. Hence, the *HM* changes the actual LLC configuration by issuing power-off/on commands to selected LLC banks.

At low level, the *HM* manages the LLC configuration change through a sequence of three stages that implement a non-blocking update mechanism, thus allowing the running application to continue its execution and without freezing the entire system waiting for the new LLC configuration to take place.

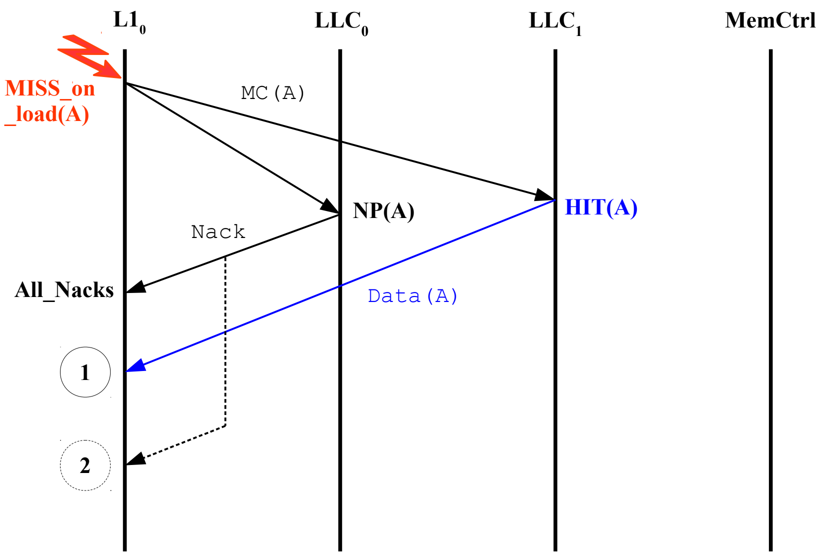


Figure 3.3: Implementation of the L1 Multicast mechanism for supporting research of cache line in lower level of cache, case data present.

This section details the *HM*, while the policy that actually decides the power commands to send to the LLC banks is discussed in Section 3.4.2.

Figure 3.4 overviews the *HM* as organized in three stages independently from the chosen policy. It is worth noticing that the *HM* is implemented in the memory controller (see *MemCtrl* in Figure 3.4) to enhance the flexibility and scalability of the proposed solution as discussed in Section 3.4.3. Moreover, the described *HM* sends a single power command to a single LLC per each change of the LLC configuration, while the generalization is trivial from the cache coherence viewpoint and is already supported by the proposed architecture. The power command can be either *power-off* or *power-on*, to force a specific target, i.e., LLC bank, to switch off or on, respectively. From a different but related viewpoint, the *power-on* increases the LLC size while the *power-off* decreases it.

In the *Send-New-LLC-Conf-to-L1s* stage the new LLC configuration is sent to every L1 cache controller. Starting from the running LLC configuration, the new one differs by one LLC bank that is either powered off or on. Note that the new configuration is computed by the policy as discussed in Section 3.4. The L1 cache controllers receive the new LLC configuration at different time instants. Once an L1 controller receives the new LLC Configuration message starts draining the active multicast transactions without allowing newer ones. The multicast is detrimental during an LLC reconfiguration phase and can induce protocol-level deadlock, thus before acknowledge the new LLC configuration message each L1 controller waits the termination of all the transactions, that started with a multicast and were active at the time when the new LLC configuration message has been received. This means that all the L1 misses are stalled until the new LLC configuration becomes active, while all the other system transactions are not affected nor suffer any performance degradation. Thus the *HM* is defined as a non-blocking LLC reconfiguration protocol. Each L1 sends an acknowledgement message to the memory controller once all the pending multicast transactions are over, while the memory controller waits to collect all the acknowledgement before moving to the next stage in the LLC reconfiguration. This behavior mimics an hardware barrier to prevent the start of the LLC reconfiguration when multicast requests are in progress thus preventing protocol-level deadlocks as detailed in Section 3.5. The *HM* enters the *Signal-Power-Cmd-to-LLC* stage once all the acknowledgement messages from the L1 controllers have been collected. This stage actually signals the target LLC that has to acknowledge at the end of its actuation phase. In particular, the target LLC controller reacts to the power command message depending on its state: if in OFF state, it starts activating the LLC bank while if in ON state, it starts switching off the LLC bank. Note that the LLC cache controller is always active and only the LLC cache memory is power gated, since it is actually the critical source of leakage.

The LLC switch from the OFF to the ON state only requires the trigger to the power gating network, thus the latency depends on the implemented technology. Conversely, few changes are required for the cache coherence protocol when the LLC bank is switched from the ON to the OFF state, i.e., the *LLC Dump* stage. The *LLC Dump* stage is transparent to the rest of the *HM* and it is performed by the switching OFF LLC bank that has to write-back all its cache lines. In particular, each cache line that has the validity bit set is a target of the *LLC Dump* mechanism. A cache line is written back by forcing a replacement with a void cache line. From the coherence protocol viewpoint the system is not frozen during the *LLC Dump*

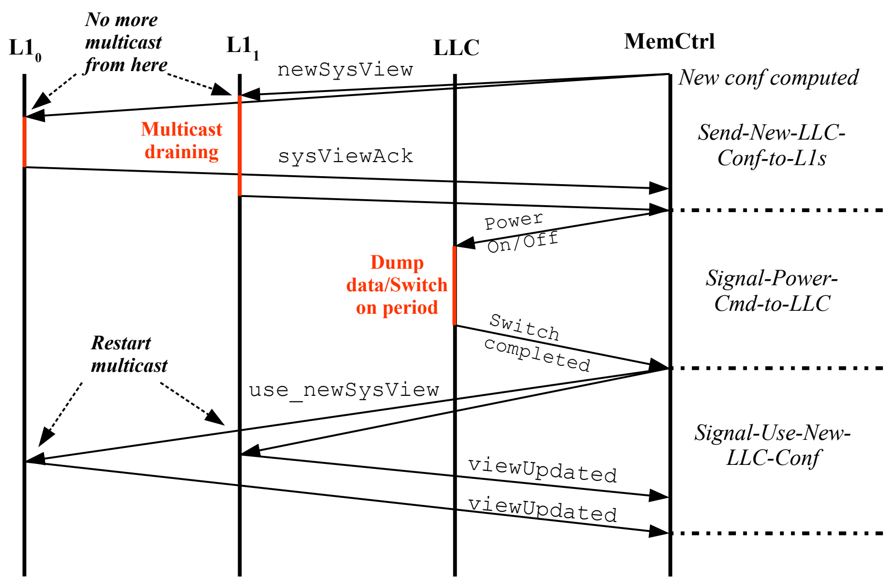


Figure 3.4: Overview of handshake mechanism.

stage, thus unicast requests can reach the dumping LLC from the L1s and the memory controllers. No multicast requests are possible, since the L1s stall all the misses until to the end of the LLC reconfiguration. A unicast request coming to a dumping LLC bank is processed depending of the state of the required cache line highlighting three different scenarios:

- **The cache line is present** and not busy, thus the LLC controller process the incoming request returning the data.
- **The cache line is not present** because it has already written back, the LLC controller sends a message to the L1 telling to reissue the request to a different LLC using the new configuration. This is critical, since all the L1 controllers force a load of the cache line freshly from the memory without having any other copy in the LLC cache, since the only copy has been deleted by the dumping LLC. Moreover, each L1 that is asking for a dump cache line is instructed to issue a get transaction using the new configuration, thus all the L1s are loading one copy of the data in a single LLC bank at most. Last, when the line is written

back all the L1 sharers are invalidated before, thus forcing them to eventually request the same cache line to the new LLC controller.

- **The cache line is busy** because is either target of the dump action or due to the processing of another unicast request. The LLC controller respond with a retry message to the requesting L1 to avoid protocol-level deadlocks. Moreover, the line can be in a dump state, thus the next stable coherence state will be not present then the answer to a re-issued request will fall in the *cache line not present* scenario described above. Conversely, if the cache-line is in busy state due to the processing of a previous unicast request, it can become available in the future in a stable state, thus a subsequent request is served according to the *cache line is present* scenario.

Once the LLC cache controller completes the *LLC Dump* stage acknowledges the *HM* at the memory controller that can move to the final stage,i.e. *Signal-Use-New-LLC-Conf*, of the *HM* LLC reconfiguration protocol that signals to all the L1s to start using the new configuration and re-enable the use of multicast mechanism. All the L1s acknowledge the *HM* to close the *handshake* protocol.

3.3 The LLC Prefetch Scheme

The proposed *LLC prefetch scheme* allows to partially shadow the augmented, worst-case request$_{on-miss}$-response latency induced by the multicast mechanism. In particular, each coherence request from an L1 miss requires a multicast action and eventually a unicast message to actually retrieve the required data. Conversely, the *LLC prefetch scheme* starts fetching the cache line before the multicast is over thus positively impacting the overall latency. In particular, for each pair LLC configuration-required data address, a single LLC bank is defined the Static NUCA (SNUCA) home bank, i.e., the bank to which the get request should be directed if the system would not allow LLC reconfigurations. The LLC SNUCA home bank for the requested multicast address can take a memory prefetch action to preload the data, since if the data is not present in all the other LLCs it receives a unicast get request from the multicast requester. The *LLC prefetch scheme* is totally safe from the protocol-level deadlock viewpoint, since the LLC configuration cannot change when a multicast request is in progress, thus for a specific address there is always the same LLC bank that acts as SNUCA home bank and triggers the data prefetch. However, minor changes are required to the directory

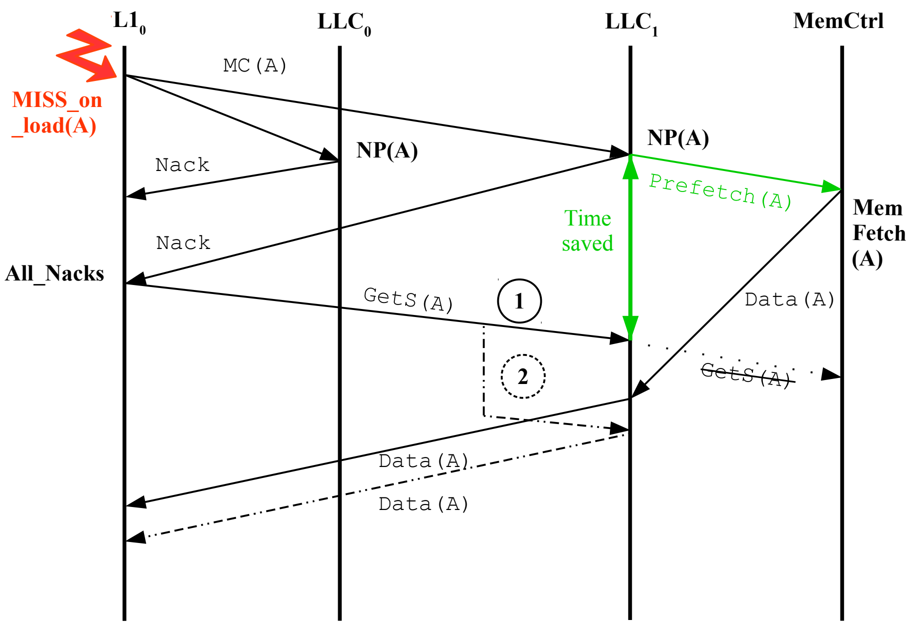


Figure 3.5: The LLC Prefetch mechanism.

controller to avoid data duplications in the LLC, since the prefetching LLC is not aware if another LLC bank already has a copy of the data. To this extent the *LLC prefetch scheme* implements a speculative memory fetch and the directory controller in front of the memory has to eventually sink the prefetch request if the data is already present in the LLC, i.e., in a different LLC. In particular, the directory controller sends a not acknowledge message back to the prefetching LLC bank each time it has an entry for the requested address.

Figure 3.5 details the eventually reduced latency due to the LLC prefetch scheme. The $L1_0$ multicast for data A that is not present in the LLC. All the LLCs notify with a message back to the $L1_0$, while LLC_1 as the SNUCA home for A tries to prefetch it. At the same time $L1_0$ collected the last negative acknowledge message and triggers a unicast get request to the same SNUCA home LLC bank that is already waiting for the prefetched data.

3.4 Policy Architecture

This section details the *policy architecture* as made of three parts. The *reference energy model* is discussed in Section 3.4.1 and represents the golden model for the energy consumption of the considered architecture. It is used to validate the power actions taken by the policy to optimize the overall energy-delay product (EDP). Conversely, Section 3.4.2 describes the actual decision policy that is responsible of taking the power command decision thus dynamically changing the LLC size. The *policy* exploits different information to estimate the consumed energy with respect to the statistics used to compute the *reference energy model.* Those information are made available to the policy through a low overhead signalling scheme that leverages the already implemented NoC to convey the information to the policy controller. The feasibility analysis as well as the details of such a signalling scheme are presented in Subsection 3.4.3.

3.4.1 The Reference Energy Model

The total energy consumed by the application a that runs to completion on the architecture is defined as:

$$E_{a,tot} = \sum_{\substack{i \in \{NoC, Cache, Mem\} \\ j \in \{Dyn, Leak\}}} E_{a,i,j} \tag{3.1}$$

where for each sybsystem,i.e., NoC, cache hierarchy, memory controllers and CPUs, both the dynamic (dyn) as well as the static ($leak$) energy contribution are considered. It is worth noticing that the CPU energy is not considered in this work and left as a future work.

In particular, $E_{a,tot,i}, i \in 0, ..., N$ defines the energy consumed by the application a during the observation window, i.e. epoch, i, thus allowing to expand Equation 3.1 to highlight the consumed energy in a specific time epoch as follows:

$$E_{a,tot,t} = \sum_{\substack{i \in \{NoC, Cache, Mem\} \\ j \in \{Dyn, Leak\}}} E_{a,i,j,t} \qquad t \in TimeEpoch \tag{3.2}$$

Moreover, a different definition of the total energy required by the application

a that is equivalent to the one provided in Equation 3.1 is:

$$E_{a,tot} = \sum_{\substack{i \in \{NoC, Cache, Mem\} \\ j \in \{Dyn, Leak\} \\ t \in 0,..,N}} E_{a,i,j,t} \tag{3.3}$$

where t spans over all the time epochs of the application. Starting from the overall definition of the energy required by an application a to execute, the rest of this section details the energy contribution due to each subsystem. In particular, the discussion is focused on the consumed energy per epoch to enhance the readability of the policy description that operates a power decision on an epoch base. Moreover, the energy defined over an epoch allows to analyze the dynamic of the energy as a discrete temporal series. For each cache bank b, the dynamic and static energy consumed within the time epoch t are defined by Equation 3.4 and Equation 3.5, respectively.

$$E_{cache_b,dyn,t} = \#Access_{cache_b,t} * DYN_{access} \tag{3.4}$$

$$E_{cache_b,leak,t} = P_{cache_b,leak} * EpochLen_t \tag{3.5}$$

In particular, $P_{cache_b,leak}$ defines the average static power of the cache bank during times the duration of the considered time epoch t, i.e., $EpochLen_t$. Moreover, the total energy in the time epoch t due to the cache hierarchy is defined as:

$$E_{cache,tot,t} = \sum_{i \in \{L1s,..,LLCs\}} (E_{i,dyn,t} + E_{i,leak,t}) \tag{3.6}$$

Moreover, the energy due to each memory controller $m \in (1...M)$ in time epoch t is defined as:

$$E_{mem_m,tot,t} = (E_{mem_m,dyn,t} + E_{mem_m,leak,t}) \tag{3.7}$$

where the dynamic ($E_{mem_m,dyn,t}$) and static energy ($E_{mem_m,leak,t}$) for the memory controller are defined in Equation 3.8 and Equation 3.9, respectively.

$$E_{mem_m,leak,t} = P_{mem_m,leak} * EpochLen_t \tag{3.8}$$

$$E_{mem_m,dyn,t} = reads_{m,t} * E_{read\_mem_m} + writes_{m,t} * E_{write\_mem_m} \tag{3.9}$$

Moreover, the total energy in the time epoch t due to the memory controllers $m \in (1...M)$, is defined as:

$$E_{mem,tot,t} = \sum_{m \in \{1,..,M\}} (E_{m,dyn,t} + E_{m,leak,t}) \tag{3.10}$$

Finally, the total energy in the time epoch t due to the interconnect, i.e. to each router $r \in \{1,..,R\}$ and link $l \in \{1,..,L\}$ is defined as:

$$E_{noc,tot,t} = \sum_{\substack{r \in \{1,..,R\} \\ l \in \{1,..,L\}}} (E_{r,tot,t} + E_{l,tot,t}) \tag{3.11}$$

where the total energy of a single router r and a single link l is defined as:

$$E_{r,tot,t} = E_{r,dyn,t} + E_{r,leak,t} \tag{3.12}$$

$$E_{l,tot,t} = E_{l,dyn,t} + E_{l,leak,t} \tag{3.13}$$

where, for what concerns a router, the static ($E_{r,leak,t}$) and dynamic energy ($E_{r,dyn,t}$) are defined as:

$$E_{r,leak,t} = P_{router_r,leak} * EpochLen_t \tag{3.14}$$

$$\begin{aligned} E_{r,dyn,t} =& reads_{r,t} * E_r + writes_{r,t} * E_w + swIn_{r,t} * E_{sw\_in} + \\ &+ swOut_{r,t} * E_{sw\_out} + xbar_{r,t} * E_{xbar} \end{aligned} \tag{3.15}$$

where $reads_{r,t}$, $writes_{r,t}$, $swIn_{r,t}$, $swOut_{r,t}$ and $xbar_{r,t}$ are the number of events of the different stages of the router r in epoch t. E_r, E_w, E_{sw_in}, E_{sw_out} and E_{xbar} are the energies consumed by one single event of the respective router stage.

For waht concerns a link, the static ($E_{l,leak,t}$) and dynamic energy ($E_{l,dyn,t}$) are computed as:

$$E_{l,leak,t} = P_{link_l,leak} * EpochLen_t \tag{3.16}$$

$$E_{l,dyn,t} = P_{link_l,dyn} * EpochLen_t \tag{3.17}$$

where $P_{link_l,dyn}$ is the dynamic power of one single link, assuming an average utilization for all the links in a certain epoch.

Finally it will be show also the *Energy Delay Product*, a metric that considers in a joint way both power and performance. EDP for an architecture a and a benchmark b is defined as:

$$EDP(a,b) = EPI(a,b) * CPI(a,b) \tag{3.18}$$

where $EPI(a,b)$ is the energy per instruction dissipated by the system in the architecture a during the execution of application b, while $CPI(a,b)$ is the average number of clock cycles necessary to architecture a to execute an instruction of benchmark b.

3.4.2 Energy Saving Policy

The general goal of the policy is to reason about different LLC configurations and to take appropriate decisions regarding the state of the system, trying to minimize the overall energy consumption.

It has to be established a practical and reliable way for comparing distinct LLC configurations, in particular *how many* and *which.* For these reasons, the energy-aware policy is *delta* based: it means that the energy consumption of the *current* configuration of the system, i.e. how many energy the system is consuming with the *current* number of active LLC banks, is compared with the energy of the configurations that, starting from the *current* one, would have one more ($+1$ *hypothesis*) and one less bank (-1 *hypothesis*). This allows the policy to reason only about changes from the *current* configuration, so to consider only three states of the LLC: the actual one and the closest ones to it. Indeed, the reasoning would be unfeasible if all the possible configurations were considered, making the policy too heavy.

The reasoning of the policy rests on estimations in energy consumption, in particular on those of the three configurations mentioned above. Depending on the accuracy of these estimations, the policy can be more or less effective. A critical aspect in the computation of these estimations is the proper selection of the most important aspects of the system to track. Thus, the energy estimations of the *current*, $+1$ and -1 configurations are based on the statistic of *replacement* in the LLC banks, which are used as metric for observing the system behavior. Indeed, LLC replacements have a direct impact on the interconnect, because they affect the overall network traffic; on the Memory, because they affect on the number of reads/writes towards it; on the system performance, because replacements worsen the execution time of the application; and on L1s, because replacements in LLC banks will issue invalidation of the same data on the higher level of cache hierarchy, making the data no more available for L1s.

The policy assumes the system, in each time window, to be in a constant and stable state, i.e. with the system having a certain number of LLC replacements. This is possible because the behavior of LLC replacements changes slower than the policy actuation. This makes capture very well the dynamic of the application.

Given the LLC replacements as the monitored metric to capture the behavior of the system, the energy consumption estimation of a certain configuration C of the LLC is computed as:

$$E(C) = \sum_{i \in \{NoC, Cache, Mem\}} E_i(C) \tag{3.19}$$

where $E_i(C)$ is, following the same reasoning of the Reference Energy Model described in Section 3.4.1, the estimated energy consumption of configuration C for the interconnect, cache and memory component.

For the *current* configuration, the components are defined as:

$$E_{Cache}(current) = (LLCs_{On} * E_{bank\_on}) + (LLCs_{Off} * E_{bank\_off}) \tag{3.20}$$

$$\begin{aligned} E_{Mem}(current) =& E_{memWrite} * LLCs\_dirty\_replacement+ \\ &+ E_{memRead} * LLCs\_replacement \end{aligned} \tag{3.21}$$

$$\begin{aligned} E_{NoC}(current) =& E_{R+L} * n_{flit\_for\_pkt} * \overline{HC} * (LLCs\_replacement+ \\ &+ \hat{\overline{S}} * LLCs\_replacement) \end{aligned} \tag{3.22}$$

Equation 3.20 evaluates the *Cache* component, by considering simply the energy consumed by the powered on and powered off banks.

Equation 3.21 evaluates the *Mem* component, by taking into account number of writes towards memory as result of *dirty* replacement in LLC, i.e. replacement of modified data, and number of reads from memory as result of data replaced and reloaded, according to the assumption that the system is in a stable state.

Finally, Equation 3.22 evaluates the network-traffic energy consumption; this is made by considering that previously replaced lines are reloaded in

LLC bank and then sent back to higher cache level components which owned that cache line. E_{R+L} is the summary energy for traversing a router and a link; $n_{flit\_for\_pkt}$ is the number of flit for a packet; $\overline{HC}$ is the average Hop Count that a flit has to traverse; $\hat{\overline{S}}$ is a coefficient that capture the estimated number of sharers to which the reloaded cache lines will be sent back again. It is computed as follows:

$$\hat{\overline{S}} = \frac{\sum_{b=1}^{LLCs_{On}} \hat{S_r(b)}}{LLCs_{On}} \tag{3.23}$$

i.e. as average overall the number of LLC banks of $\hat{S_r(b)}$, computed as:

$$\hat{S_r(b)} = \overline{S(b)} * (\overline{\frac{\overline{S_{r_R}(b)}}{\overline{S_r(b)}}}) \tag{3.24}$$

which is the *estimated replacement sharers* for bank b.

$\overline{S(b)}$ is the average number of sharers for a line l in bank b. It is defined as:

$$\overline{S(b)} = \frac{\sum_{l=1}^{N} S_l(b)}{N} \tag{3.25}$$

where N is total number of lines in bank b, $S_l(b)$ are the *sharers* of a line l in bank b.

$\overline{S_r(b)}$ is the average number of sharers for a *replaced* line in bank b, defined as:

$$\overline{S_r(b)} = \frac{\sum_{l=1}^{M} S_l(b)}{M} \tag{3.26}$$

where M is the number of replacement observed in bank b.

$\overline{S_{r_R}(b)}$ is the average number of *real* sharers for a *replaced* line in bank b, defined as:

$$\overline{S_{r_R}(b)} = \frac{\sum_{l=1}^{M} S_{l_R}(b)}{M} \tag{3.27}$$

where $S_{l_R}(b)$ is the number of *Real* sharers for a line l in bank b.

The distinction between *sharers* and *Real sharers* of LLC replacements is due to the following reason: MESI protocol allows an L1 to *silently* evict a cache line, i.e. to replace a line without informing the LLC bank which contains that line too. This implies in other words that the LLC bank which contains that cache line could potentially have, among the L1 sharers, someone which will not own anymore that block. Thus, the protocol has been modified in a proper way, so that to distinguish between invalidation ack message coming from L1s that effectively have from that which have not the data. So, when a line in a LLC block is evicted, *sharers* indicates the sharers according to the knowledge of the LLC bank, while *Real sharers* are the L1s that, among the *sharers*, has not silently evicted (i.e. really owned) the data.

So said, the ratio in Equation 3.24 takes into account the problem of silent eviction of data explained before; it tries to compute, on the average number of *sharers* for a replacement in bank b, how many of them were *Real*, i.e. how many have not silently evicted the data and so they really owned it. This ratio is then multiplied for the average number of sharers of the active present lines in the LLC bank b.

Substantially $\hat{S_r(b)}$ tries to estimate how many *Real* sharer a line of LLC bank b should have, based on the past history and the actual average number of sharers; it estimates the number of real owners for a line in bank b to which i should send again the line if the line was replaced at the moment of the observation. For considering then the behavior of all the banks, $\hat{\overline{S}}$ then is computed.

For the *+1 hypothesis* configuration, energy components are defined as:

$$\begin{aligned} E_{Cache}(\mathit{+1}) =& ((LLCs_{On} + 1) * E_{bank\_on}) + \\ & + ((LLCs_{Off} - 1) * E_{bank\_off}) \end{aligned} \tag{3.28}$$

$$\begin{aligned} E_{Mem}(\mathit{+1}) =& E_{memWrite} * LLCs\_dirty\_replacement * \mu + \\ & + E_{memRead} * LLCs\_replacement * \mu \end{aligned} \tag{3.29}$$

$$\begin{aligned} E_{NoC}(\mathit{+1}) =& E_{R+L} * n_{flit\_for\_pkt} * \overline{HC} * (LLCs\_replacement * \mu + \\ & + \hat{\overline{S}} * LLCs\_replacement * \mu) \end{aligned} \tag{3.30}$$

$E_{Cache}(\mathit{+1})$ is computed by assuming one more bank in the number of active banks (and consequently one less bank in the switched off ones) w.r.t the *current* configuration. $E_{Mem}(\mathit{+1})$ is computed as the *current* configuration, with the difference that *LLCs_replacement* and *LLCs_dirty_replacement* are weighted by a coefficient μ, computed as:

$$\mu = \frac{(LLCs_{On} + 1)}{n\_total\_LLCs} \tag{3.31}$$

The idea behind this coefficient is to weight replacement according to the hypothesis that one more bank would be powered on w.r.t the *current* configuration. Assuming that the system would benefit from this additional space, the most important aspect is that μ is able to differs situation in which there are a small number of powered on banks from that in which there are already a big available number of active banks. This aspect is crucial for distinguishing situation in which one more active bank can be more necessary than others. μ is a number less or equal to 1, closer to 0 as long as the number of active LLC banks is really small and closer to 1 with the increasing number of banks. It emphasizes the fact that, when the system has a very small number of active LLC banks, it would really benefit from one more active bank by decreasing of a big portion the number of total replacement; on the other side, when the system has a number of active banks very closer to the maximum number, one more active banks would improve the number of replacement but not in the same manner as if we have only one active bank, being the configuration very closer to the one with the maximum available space. Finally, in $E_{NoC}(\textit{+1})$ is equal to $E_{NoC}(\textit{current})$, with the difference that also here the replacement are weighted for the μ coefficient.

Finally, for the *-1 hypothesis* configuration, energy components are computed in this way:

$$\begin{aligned} E_{Cache}(\textit{-1}) =& ((LLCs_{On} - 1) * E_{bank\_on}) + \\ &+ ((LLCs_{Off} + 1) * E_{bank\_off}) \end{aligned} \tag{3.32}$$

$$\begin{aligned} E_{Mem}(\textit{-1}) =& E_{memWrite} * (LLCs\_dirty\_replacement + \overline{D_{lines}}) + \\ &+ E_{memRead} * (LLCs\_replacement + \overline{D_{lines}}) \end{aligned} \tag{3.33}$$

$$\begin{aligned} E_{NoC}(\textit{-1}) =& E_{R+L} * n_{flit\_for\_pkt} * \overline{HC} * ((LLCs\_replacement + \overline{D_{lines}}) + \\ &+ \hat{\overline{S}} * (LLCs\_replacement + \overline{D_{lines}})) \end{aligned} \tag{3.34}$$

Algorithm 1 Energy-Aware Policy

```
1: $E(current) := E_{Cache}(current) + E_{Mem}(current) + E_{NoC}(current)$
2: $E(-1) := E_{Cache}(-1) + E_{Mem}(-1) + E_{NoC}(-1)$
3: $E(+1) := E_{Cache}(+1) + E_{Mem}(+1) + E_{NoC}(+1)$
4:
5: if $E(current) > E(+1)$ then
6:     $b :=$ selectTargetBank(LLCs_current_config)
7:     sendSwitchOn(b)
8: else
9:     if $E(current) > E(-1)$ then
10:        $b :=$ selectTargetBank(LLCs_current_config)
11:        sendSwitchOff(b)
12:    else
13:        noChange()
14:    end if
15: end if
```

$E_{Cache}(\text{-}1)$ now is computed by assuming one less bank in the number of active banks (and consequently one more bank in the switched off ones) w.r.t the *current* configuration. $E_{Mem}(\text{-}1)$ takes into account additional dump lines, the ones of the switched off bank. This value is represented by the $\overline{D_{lines}}$ factor in the formula, computed as:

$$\overline{D_{lines}} = \frac{\sum_{b=1}^{LLCs_{On}} lines(b)}{LLCs_{On}} \tag{3.35}$$

with $lines(b)$ number of *not null* lines in bank b. So $\overline{D_{lines}}$ denotes the average number of active entries among all the active LLCs. Finally, in $E_{NoC}(\text{-}1)$ is taken into account the contribution of the average number of lines that would be dumped by switching off one bank, through $\overline{D_{lines}}$.

The reasoning of the energy-aware policy is detailed in Algorithm 1. The *selectTargetBank()* function establishes to which LLC bank sending the power command (On-Off); in the proposed policy, LLC banks are chosen *sequentially*, in the sense that they are powered Off in a decreasing order and they are powered On in an increasing order of LLC bank *index b*. The *noChange()* function captures the situation in which a LLCs reconfiguration is not needed.

3.4.3 Architecture Feasibility and Policy

Regarding the energy policy, two additional aspects have to be discussed. The first is where the policy is placed; we have decide to put the reasoning of the policy in the Memory Controller. This choice has been made with the intent to make the system simple and scalable, being the policy placed in only one point and allowing to modify and implement it only in one single position.

The second aspect is that a Signaling Network is exploited to send *probe* messages to the Memory Controller in charge of running the policy; the important point is that it is used the existing network taken for general injected traffic. This probe messages contain the most important statistics that will be used in the computation made by the energy policy, for deciding if and which LLC bank switching on/off. These probe messages can be sent every predefined time interval elapses: this value can be established and assigned inside the LLC Controllers.

The main statistics used by the policy and contained in each probe messages sent by the LLC Controllers are the following:

- number of replacements
- number of dirty replacements
- sharers of replacements
- Real sharers of replacements
- active entries in LLC bank
- average sharers for active entries in LLC bank

All these statistics are collected by the Controllers until the probe message is sent to the memory controller, then they are reset and recollected for the next time interval. Each of these statistics takes a certain number of bits inside the *probe* message.

Assuming a 4x4 topology, the one considered in this work, for *256kB* bank size number of replacements and *dirty* replacement have been assumed not to be greater than half of the bank size; so, this two statistics takes 11 bits in the *probe* message, plus 1 bit for taking into account an $overflow$ case. Sharers, *Real* sharers and average sharers for active entries takes 4 bits, being the topology a 4x4. Active entries in the bank can be at most equal to the number of lines of the bank: so it takes 12 bits which, for a $64B$ line size, correspond to 4096 lines. These considerations make the probe messages of 48 bits.

For *1MB* bank size assuming again a 4x4 topology, the differences w.r.t to the previous case is in the replacements and *dirty* replacements, which are considered taking 13 plus 1 bits, and in the active entries, which in this case can be at most equal to 16384, thus resulting in 14 needed bits. This makes the probe messages of 54 bits.

For both, they have to be considered also 4 bits to identify the Sender of the message and 4 bits to identify the Receiver; moreover, 2 bits are used for the type of flit and 6 bits for the type of message. This made, for the *256kB* and *1MB* bank size cases, the probe message long 64 and 70 bits respectively, thus fitting the general network message of 128 bits.

It is worth noticing that when a LLC bank is switched off, it is stopped to send its probe messages; on the other side, when a bank is switched on, it restarts to send probe messages.

3.5 Deadlock Avoidance Analysis

During the changing in the protocol behaviour, while implementing Multicast, Handshake and Prefetch, some cases of deadlock have been solved. For Multicast mechanism, deadlock-freedom is granted by waiting to get all the responses from the LLC banks. Other requests for the same cache line are stalled, waiting that multicast transaction finishes and that the line ends in a steady state.

Another deadlock scenario is shown in Figure 3.6. In the Handshake mechanism, unicast requests incoming to the target LLC bank during the ON-OFF transition are managed depending on the state of the cache line requested. If the cache line is in a transient state, request is bounced between the L1 and the bank until the line will move in a steady state. If the cache line is in a steady state, there can be two possibilities: the line is in a Not Present state or is in a state different from Not Present. In the first case the request is sent back to the L1, informing it to use the new configuration: this is due to the the fact that the LLC bank is flushing its content for moving in a Switched Off state, so it does not want to accept new request otherwise it could not really reach convergence in the dumping of cache lines. In the

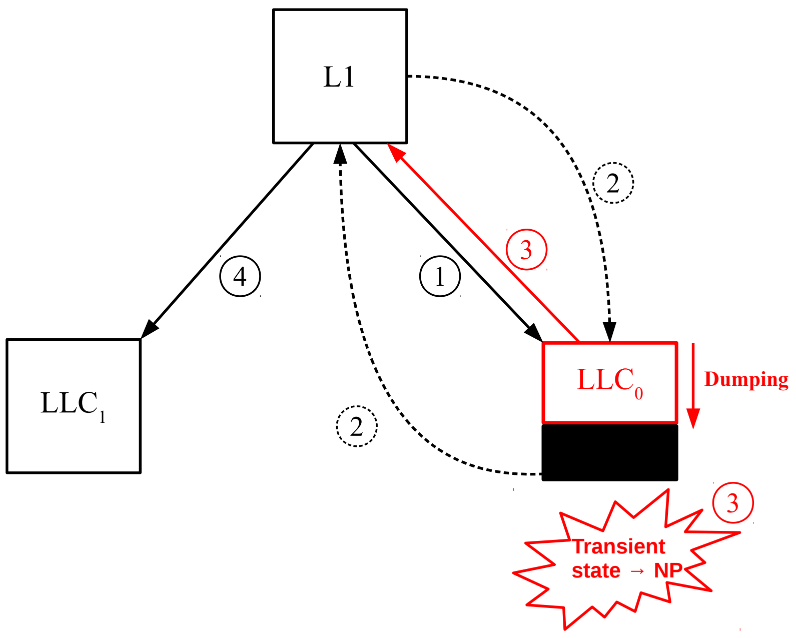


Figure 3.6: Possible deadlock in the handshake mechanism: (1) L1 makes a unicast request to a LLC-0; (2) LLC-0 is dumping and the requested line is in a transient state, so the request is bounced; (3) at a certain point the requested line moves in Not Present state: the LLC-0 informs the L1 to use the new view; (4) L1 receive the response from LLC-0 and retries its unicast request towards LLC-1.

second case the request, no matter the steady state in which the line is (SS, M, etc.), is served: this is due to the fact that soon or later that line in the bank will be dumped, and consequently will automatically be invalidated in the sharing L1s. Finally as explained in Section 3.2, when the memory controller sends a new configuration to the L1s, these have to drain and complete their pending multicast requests before answering with an ACK response. The reason is not immediate and it is explained by the Figure 3.7 . This picture represents the situation in which are present two L1s (L1-0 and L1-1, drawn on the outside) and two LLC banks (LLC-0 and LLC-1, drawn inside). At the beginning of the transaction only LLC-0 is powered on, while LLC-1 is switched off: this means that in the current view, L1-0 and L1-1 see in their last shared level cache only LLC-0 bank. First L1-0, then L1-1, receive a LOAD request for data A but since they have not the data, they trigger a multicast request towards all the active LLC banks, i.e

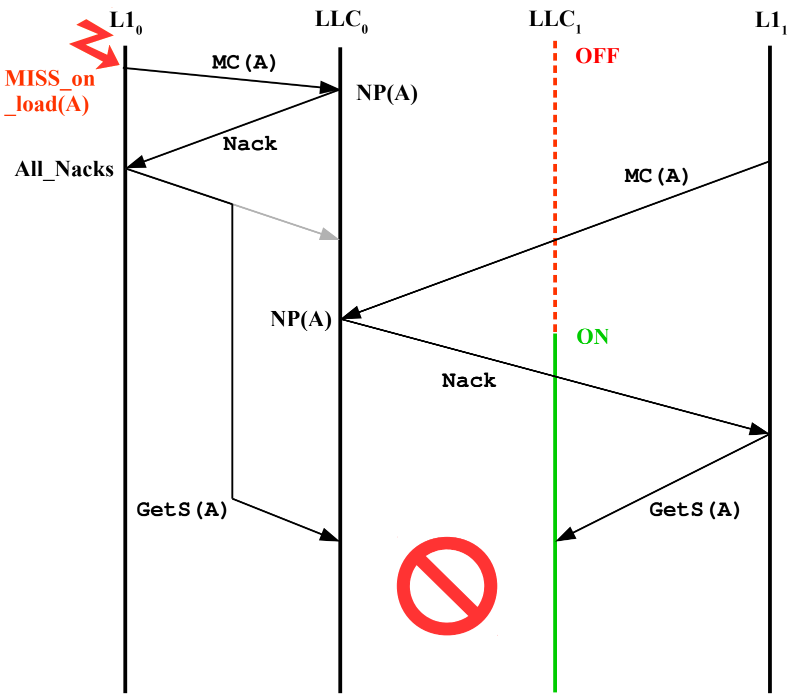


Figure 3.7: Possible deadlock in the handshake mechanism.

in our case only LLC-0. Since LLC-0 has not the data, in turn it answers NACK, first to L1-0 then to L1-1 request. In the meanwhile, the memory controller sends a Switch-On command to LLC-1 and at the same time a new configuration to all the L1s. In the figure is shown what would happen if multicast messages were allowed and triggered during a Switch-On/Switch-Off transition of a LLC bank. The most important point is the following: due to different reasons (the congestion of the network, the different position of the L1s in the network, etc.) the message containing the new configuration will not arrive at the same moment to the L1s. In our case this is translated in the following scenario: after receiving the NACK response from LLC-0, L1-0 has not received the new configuration yet and so it issues the unicast request according to its configuration, which contains only LLC-0. On the other side L1-1 receives the new view exactly during a multicast transaction and, according to this new configuration, it issues the unicast request for the same block asked by L1-0 to another LLC bank, in particular the switched on one. This would lead the last shared level of cache to have two copies of the same data placed in different positions, breaking the coherence and causing problems to the running application: this is not an allowed situation.

By stopping the new multicast request while draining the pending ones when a L1 receives the first message of the handshake protocol containing the new configuration, and by restarting them only when the memory controller send the message that inform explicitly the L1s to start using it, we forces the L1s to complete the requests that have been issued according to an older configuration and then to restart the new requests according to the same new one. By forcing the L1s to use the new configuration only when they receive the explicit message from the memory controller, independently from when they receive this message, we are sure that all the new requests will be issued according to the same new view of the LLC.

Chapter 4

Experimental results

In this chapter is presented the assessment of the proposed Switching On-Off LLC bank mechanism, from the performance and energy view point. It will be shown its behavior with a representative subset of the *Slash2x* benchmark suite, discussing for every application the total energy consumption, and detailing that of the three most important component of the system: cache, interconnect and memory. Also, it will be reported the $energy-delay\ product$ as a power-performance aggregate metric. For every aspect a comparison with a baseline MESI architecture will be shown, highlighting the differences. It will be shown how the overall energy of the system is dominated by the static energy of the cache banks and how the total consumption can be improved, by giving to every application a number of banks that fits its demand.

The rest of the chapter is organized as follows. Section 4.1 describes the simulation setup, the target architecture and the benchmarks used. Section 4.2 describes the executed tests for validating our mechanism. Section 4.3 describes the obtained results and the improvement on energy consumption. Finally Section 4.4 describes some design-space exploration, showing how the behavior of the examined applications change with different architectural parameters and motivating some aspects of our mechanism.

4.1 Simulation Setup

Our baseline architecture is a S-NUCA one with no possibilities of switching on/switching off slice of Last Level Cache (the L2 in our scenario): given a

Processor Core	1GHz, In-Order X86 Core, 1 cycle per execution phase
L1I Cache	32kB 8-way Set Associative
L1D Cache	32kB 8-way Set Associative
L2 cache	256kB per bank, 16-way Set Associative
Coherence Prot.	MESI (3 VNET protocols, 4 VC per VNET protocol)
Memory Type	DDR3_1600_x64
Topology	2D-mesh 4x4 at 16 Cores
Network frequency	2GHz
Channel width	128 bits
Flit size	16 byte
Technology	45nm at 1.0V
Real Traffic	Subset of *Splash2x* benchmarks.

Table 4.1: Experimental setup: processor and technology parameters common to the considered architectures.

certain size of the LLC Cache, every part will remain powered on, even if the application will not need it. This baseline architecture is a MESI-based for what concerns coherence of data. In the comparisons with our architecture, we will refer to as "MESI". In our architecture this baseline has been enriched with a multicast mechanism, then improved with a prefetch optimization, for supporting the Dynamic mapping of data (as explained in Section 3.1 and Section 3.3). Moreover, a Handshake mechanism for switching on-off portion of LLC Cache has been added after, with the memory controller provided with a policy controller for taking decisions (as explained in Section 3.2). This policy controller contains all the logic of our energy-saving oriented policy (Section 3.4.2). In the following pages, we will refer to it as "Flexible Cache" (FC). The static parameters embedded in our policy are summarized in Table 4.3. The first two values are the static energy consumed by a single LLC bank in powered on and powered off state respectively, while the third value is the latency for switching-on a single bank. They have been estimated looking at CACTI-P tool [20]. The remaining values have been estimated looking at the statistics obtained by the simulations and the configuration values of our architecture. Probe timeout is the time with

Splash2x Application	Input Set Size

Barnes	Simsmall Input Set Size;
	16K Particles;
	16 threads

Cholesky	Simsmall Input Set Size;
	13992x13992, NZ=316740;
	16 threads

Fft	Simsmall Input Set Size; 2^{20} total complex data points; 16 threads

Fmm	Simsmall Input Set Size;
	16K Particles;
	16 threads

Lu_cb	Simsmall Input Set Size;
	512x512 Matrix, Block = 16;
	16 threads

Lu_ncb	Simsmall Input Set Size;
	512x512 Matrix, Block = 16;
	16 threads

Ocean_cp	Simsmall Input Set Size;
	514x514 Grid;
	16 threads

Ocean_ncp	Simsmall Input Set Size;
	514x514 Grid;
	16 threads

Volrend	Simsmall Input Set Size;
	Head-Scaledown4;
	16 threads

Water_nsquared	Simsmall Input Set Size; 8^3 Molecules; 16 threads
Water_spatial	Simsmall Input Set Size; 15^3 Molecules; 16 threads

Table 4.2: The used benchmarks taken from the Splash2x suite.

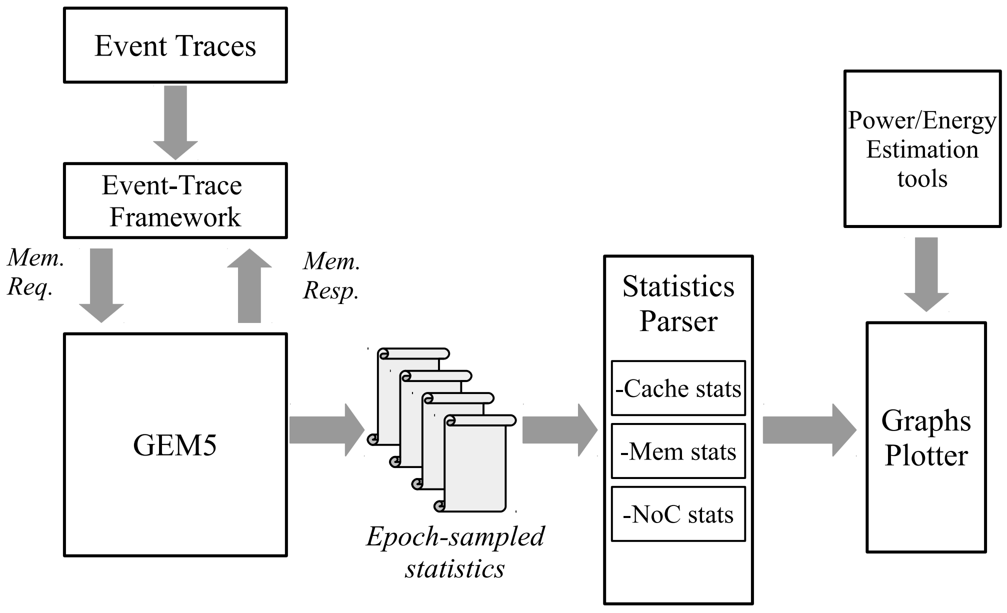


Figure 4.1: Workflow representation.

which LLC controller send probe messages. The cache configuration is a 32KB L1I caches, a 32KB L1D caches and a 256KB LLC caches per bank. The considered NoC topology is a 4x4 2D-mesh with 16 cores and NoC routers with a 4 VC. The adopted memory type is DDR3_1600_x64 with 10 ns of static frontend and backend latency [16]. Eleven application form the Splash2x benchmarks suite have been used for validating our architecture: they are listed in Table 4.2.

A representation of the workflow can be seen in Figure 4.1 The architecture have been integrated in the GEM5 cycle accurate simulator, in an enhanced version [33, 36, 34, 35, 32, 8]. It has been modified such that a periodic dumping of the statistics is possible: setting a simulation parameter, it is possible to decide how often during the simulation the statistic are sampled. In this way, it is possible to obtain $epoch - based$ statistics, with an epoch as long as the value set in the parameter. This value corresponds to Epoch time in Table 4.3 and for us is equal to 1ms. Applications of the Splash2x have been simulated in GEM5 through SynchroTrace, a scalable,

Parameter	Chosen Value
Energy on state LLC bank	110 μJ
Energy off state LLC bank	10nJ
Switching on LLC bank latency	10 cycles (10ns)
Memory read energy	16nJ
Memory write energy	13nJ
Router + Link energy traversal	430pJ
Flit for 1 packet	10
Average Hop Count for 1 flit	3.5
Epoch time duration	1ms
Probe timeout	1ms

Table 4.3: Energy parameters embedded in our policy.

flexible, and accurate trace-based multi-threaded simulation methodology [22]. Epoch-statistics have been processed and plotted with GNU Octave plotting tool, obtaining all the graphs contained in this chapter; energy and power value related to the Cache banks have been obtained from CACTI6.0 tool [21], while the NoC value related to energy/power of links and routers have been obtained from the DSENT tool [27].

4.2 Regression Tests

To avoid deadlock in the coherence protocol modification, several tests have been executed. Each step and addition w.r.t the baseline architecture has been followed by a validation session, to verify the absence of deadlocks and the strength of the protocol w.r.t to introduced changes. For Multicast mechanism, the *ruby_mem_test* module has been executed exactly for this purpose. The MemTest class, implemented in our used GEM5 simulator, tests a cache coherent memory system by generating false sharing and verifying the read data against a reference updated on the completion of writes. Each tester reads and writes a specific byte in a cache line, as determined by its unique id. Thus, all requests issued by the MemTest instance are a single byte and a specific address is only ever touched by a single tester. In addition to verifying the data, the tester also has timeouts for both requests and responses, thus checking that the memory-system is making progress.

When this timeout exceeds a certain threshold (set in the Controller of the MemTest), an exception is raised signaling that a possible deadlock has been detected.

The Handshake mechanism has been validated by embedding in the Policy Controller first a *Static* then a *Random* decision maker. The *Static* policy embeds in its logic a predefined number of choices that have to be executed. This policy has been used to verify the most simple, basic and short sequences of choices, so that in case of a deadlock it would have been easily reproducible. The *Random* one instead validates the system by generating a sequence of casual choices randomly generated. It has been used to verify more complex sequence of choices, together with the durability and the strength of the architecture. Moreover, it has suddenly been slightly improved with the possibility of setting the probability with which a decision type is taken (for example for every time a decision has to be taken, the probability of switching-on can be set equal to 60% while switching-off equal to 40%). This policy has also two variants: one that take another decision as soon as it ends the last choice, another that can wait some time between every two sequential decisions. Both the policies (*Static* and *Random*), independently of the decision taken, can deal with only one bank at a time. Given this two implemented policies, a set of tests has been executed and successfully completed while running the *ruby_mem_test* module explained above. The set of tests has been generated by exploring different scenarios and changing some architectural parameters, started from a situation in which all LLC banks are powered on. In detail, the tests are the followings:

Type	**Topology**	**Test**	**Time**	**Mem size**
Static	4x4 2D-Mesh	Single Switch-Off of a single LLC bank then stop; made for LLC_x with x $\in$ 0..15	0ms	500MB
Static	4x4 2D-Mesh	Continuous Switch On-Off of a single LLC bank; made for LLC_x with x $\in$ 0..15	0ms	500MB

Static	2x4 2D-Mesh	Sequential Switch-Off from LLC_0 to LLC_6, then Sequential Switch-On from from LLC_0 to LLC_6	0ms	500MB
Static	4x4 2D-Mesh	Sequential Switch-Off from LLC_0 to LLC_6, then Sequential Switch-On from from LLC_0 to LLC_6	0ms	500MB
Static	4x4 2D-Mesh	Sequential Switch-Off from LLC_0 to LLC_{14}, then Sequential Switch-On from from LLC_0 to LLC_{14}	0ms	500MB
Static	1x2 2D-Mesh	Continuous Switch On-Off of LLC_0, L1=2kB, LLC=4kB	0ms	64kB
Static	1x2 2D-Mesh	Continuous Switch On-Off of LLC_1, L1=2kB, LLC=4kB	0ms	64kB
Static	1x2 2D-Mesh	Continuous Switch On-Off of LLC_0, L1=2kB, LLC=4kB	1ms	64kB
Static	1x2 2D-Mesh	Continuous Switch On-Off of LLC_1, L1=2kB, LLC=4kB	1ms	64kB
Static	2x2 2D-Mesh	Continuous alternate Switch On-Off of LLC_0 and LLC_1, L1=2kB, LLC=4kB	1ms	64kB
Static	2x2 2D-Mesh	Continuous alternate Switch On-Off of LLC_0 and LLC_1, L1=2kB, LLC=4kB	5ms	64kB

Static	2x4 2D-Mesh	Continuous Switch On-Off of LLC_0	0ms	64kB
Random	2x4 2D-Mesh	Switch On-Off random LLC bank with different probabilities (Switch Off 60%, Switch On 40%)	0ms	500MB
Random	4x4 2D-Mesh	Switch On-Off random LLC bank with different probabilities (Switch Off 60%, Switch On 40%)	0ms	500MB
Random	4x4 2D-Mesh	Switch On-Off random LLC bank, taking a minimum number of Switched-On banks equal to 8	0ms	500MB
Random	4x4 2D-Mesh	Switch On-Off random LLC bank, taking a minimum number of Switched-On banks equal to 1	0ms	500MB
Random	4x4 2D-Mesh	Switch On-Off random LLC bank running all the Splash2x applications cited in Table 4.2 to completion	0ms	8GB

The Prefetch improvement for Multicast mechanism has also been validated with two scenarios:

Type	**Topology**	**Test**	**Time**	**Mem size**
Static	1x2 2D-Mesh	Continuous Switch On-Off of LLC_0	5ms	500MB
Random	4x4 2D-Mesh	Switch On-Off random LLC bank, taking a minimum number of Switched-On banks equal to 1	1ms	500MB

4.3 Energy and Performance Analysis

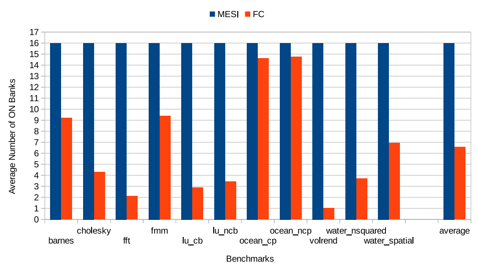
Flexicache reduces the average number of active LLC banks (see Figure 4.2(a)) by adapting the LLC size to the requirements of the executing application. Figure 4.2(a) highlights different class of applications with respect to the memory requirements. Few benchmarks, e.g., *ocean_cp* and *ocean_ncp* highlight a strong memory-bound behavior, with an average number of active LLC banks of 14 and 15, respectively. Conversely, FlexiCache can greatly reduce the number of LLC banks for the benchmarks with smaller memory requirements. For example, a single LLC banks is active on average for *volrend* and two banks are used on average for *fft*.

Figure 4.2(c) shows the positive impact on the energy consumption due to the reduced number of active LLC banks. In particular, the energy saving is proportional to the average number of the switched off LLC banks. For example, *volrend* is executed with a single, active LLC bank with a net energy saving greater than 60%. It is worth noticing that the energy saving is kept proportional to the number of switched off banks until the performance metric, i.e., execution time, is not affected. In particular, two different aspects negatively impact the performance metric: the multicast mechanism and the limited LLC size.

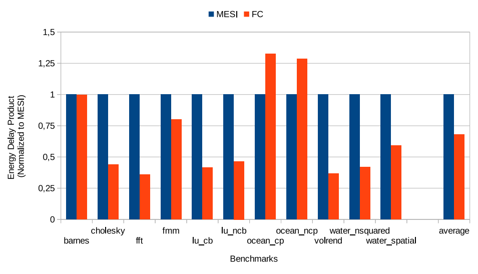
First, *FlexiCache* can decide to switch off just one bank more when the application is only asking for slightly more LLC space. However, the weight of the additional switched off bank is bigger than the performance reduction from the energy-delay viewpoint. Thus, the reduced LLC size increases the LLC misses and consequently the memory accesses that impact the application execution time.

Second, the multicast mechanism can indirectly impact the execution time by affecting both the network and the queuing latency. However, the overall system performance, i.e., execution time, are not affected until the network is close to saturation. In this scenario, a memory-bound application can severely contribute to the network saturation by enforcing an high number of concurrently active multicast transactions. Moreover, a memory-bound application is expected to prevent the LLC bank switch-off, thus each multicast transaction will generate an high traffic volume due to the need of addressing all the active LLC banks.

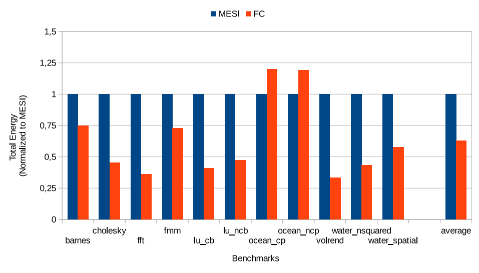
However, Figure 4.2(d) shows a performance overhead lower than 5% for almost all the considered benchmarks, with two notable exceptions, i.e.,



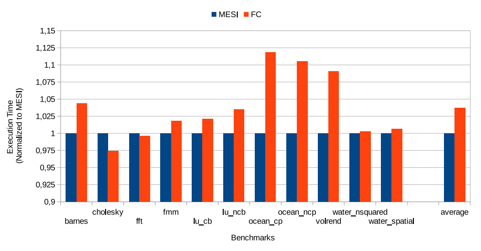
(a) Average Number of ON Banks



(b) Energy-Delay Product



(c) Total Energy Consumption



(d) Execution Time

Figure 4.2: Behavior of all benchmarks for what concerns the Average Number of ON LLC Banks, Total Energy Consumption and Execution Time with Prefetch mechanism and 256kB bank size.

ocean_cp and *ocean_ncp*. An in-depth analysis discussed in Section 4.3.1 highlights the multicast as the main responsible for the performance overhead, since it goes really near to saturate the interconnect resources because of the high number of LLC bank requested. Moreover, the prefetch scheme presented in Section 3.3 allows to timely fetch data from memory during the multicast instead of waiting for a possible, subsequent unicast *get* request, thus shadowing part of the memory latency. Last, the considered interconnect model does not support the broadcast/multicast mechanism, thus each multicast request is injected into the interconnect as a chain of unicast requests.

Figure 4.2(b) depicts the Energy-Delay-Product (EDP) normalized to the baseline MESI architecture for all the considered benchmarks. *FlexiCache* improves by 25% on average the EDP while for 6 out of 11 benchmarks the improvement is greater than 50%. Once more, a net performance and energy overhead is reported for both *ocean_cp* and *ocean_ncp* due to the impossibility to save energy by switching off LLC banks and the net performance overhead imposed by the overkilling *multicast mechanism* that can not benefit from any interconnect multicast support.

4.3.1 Time-based Detailed Analysis

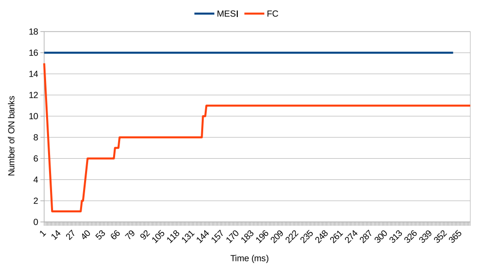
The results discussed in Section 4.3 overview the benefit of the proposed methodology without providing any details on the adaptivity of the policy as well as the contributions in terms of energy due to the different subsystems. This section presents those details by splitting the energy consumption in three terms to expose the contribution of the memory, the caches and the interconnect. Moreover, the presented results show the time-based evolution of those information side by side with the actions taken by the policy to dynamically optimize the LLC size.

Three representative benchmarks are discussed in the rest of the sections to summarize the main characteristics of all the analyzed applications even if each application presents its own specific behavior.

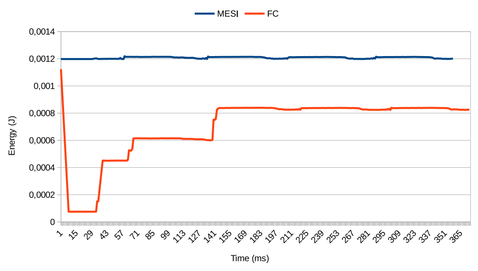
Figure 4.3 reports the collected results for the *barnes* application showing the time evolution of five different statistics. Figure 4.3(b), 4.3(c) and 4.3(d) depict the evolution over time of the energy consumed by the entire LLC, the memory and the interconnect, respectively. Moreover, Figure 4.3(a) reports the number of active LLC banks and the total energy is shown in Figure 4.3(e). Note that for each considered statistic both the MESI and the *FlexiCache* architecture have been simulated and reported side by side.

The application traverses different execution phases with a configuration stage at the beginning of the execution. The phases are periodic as demonstrated by the consumed network energy in Figure 4.3(d) using the *FlexiCache* architecture. Moreover, the energy due to the memory accesses spikes before the beginning of each phase. However, each spike consumes less energy than the previous one, since *FlexiCache* dynamically switches more LLC banks to face the increased memory requirements. In particular, the time periods where the NoC energy increases are correlated with the increase in the multicast actions due to the limited L1 size that enforce replacements of useful data. However, the L1 misses are correctly filtered by the LLC layer without resorting to too many memory data fetches.

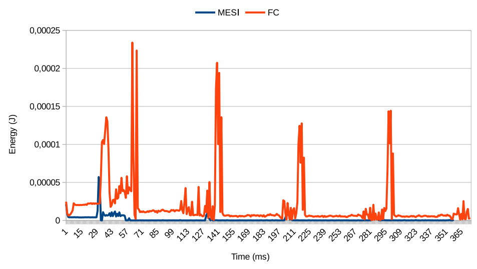
Figure 4.4 details the same information for the *cholesky* applications for both the MESI and the *FlexiCache* architectures. *Cholesky* shows a different behavior with respect to *barnes*, with a long phase that enables a single



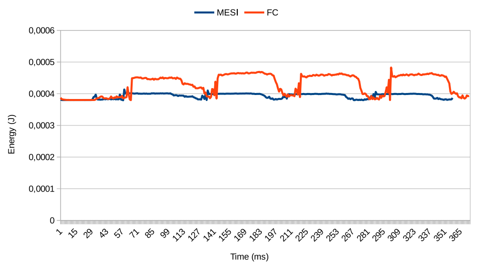
(a) BARNES On LLC Banks per epoch



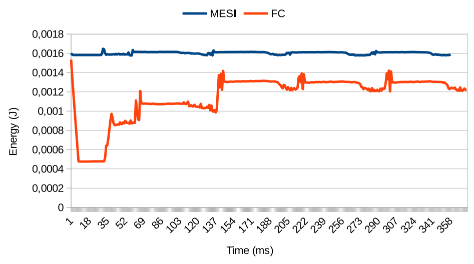
(b) BARNES Cache Energy Consumption per epoch



(c) BARNES Memory Energy Consumption per epoch

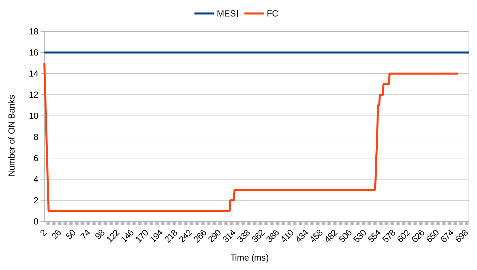


(d) BARNES NoC Energy Consumption per epoch

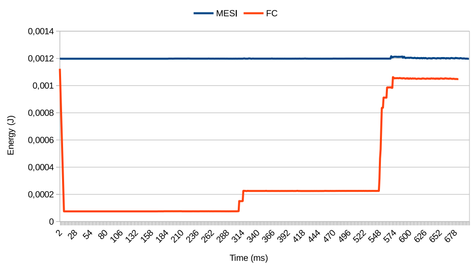


(e) BARNES Total Energy Consumption per epoch

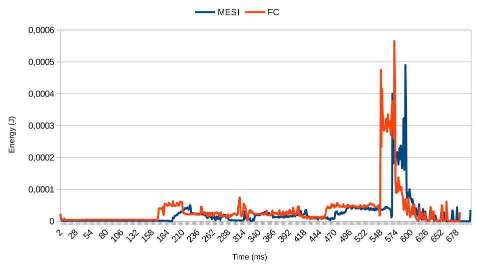
Figure 4.3: Behavior of Barnes benchmark for what concerns the Number of ON LLC Banks, Cache/Memory/NoC Energy and Total Energy Consumption with Prefetch mechanism and 256kB bank size.



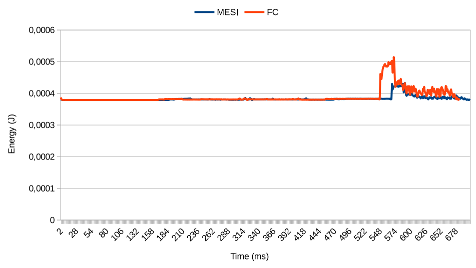
(a) CHOLESKY On LLC Banks per epoch



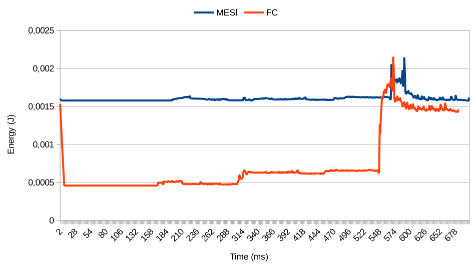
(b) CHOLESKY Cache Energy Consumption per epoch



(c) CHOLESKY Memory Energy Consumption per epoch



(d) CHOLESKY NoC Energy Consumption per epoch



(e) CHOLESKY Total Energy Consumption per epoch

Figure 4.4: Behavior of Cholesky benchmark for what concerns the Number of ON LLC Banks, Cache/Memory/NoC Energy and Total Energy Consumption with Prefetch mechanism and 256kB bank size.

master thread before spawning the working ones. In this scenarios, *Flexi-Cache* can take advantage of its power gating capabilities to switch off up to 15 LLC banks without affecting the execution time. Moreover, the policy is fast enough to track the evolution of the execution phase and consequently adapting the number of active LLC banks.

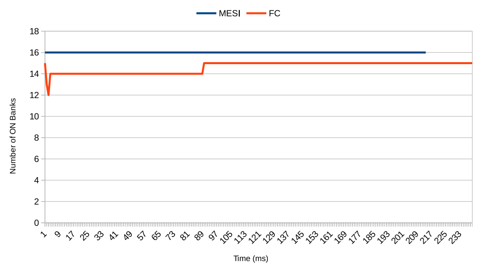
Ocean_cp represents the third type of application considered in this section and the obtained results are reported in Figure 4.5. The high memory requirements does not allow to effectively switch-off but few LLC banks for the entire execution, thus a modest energy saving is obtained. Besides the small L1 size triggers several multicast transactions that impose broadcast-equivalent NoC traffic. Moreover, the energy due to the memory accesses for MESI and *FlexiCache* are similar, since the application memory footprint does not fit in the entire LLC. As a final remark, Figure 4.5(e) highlights similar energy behavior for the two architectures while the excessive multicast actions without a proper broadcast support at interconnect level severely affect the NoC latencies thus degrading the overall execution time.

4.4 Design Space Exploration (DSE)

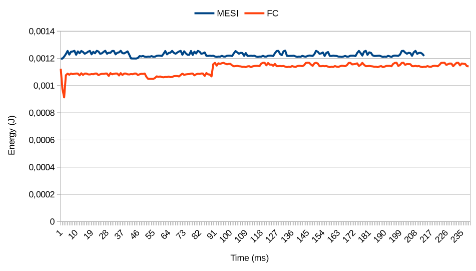
This section explores few design parameters to further strengthen the assessment of the proposed *FlexiCache*. Results considering a bigger LLC are presented Section 4.4.1, while Section 4.4.2 highlights the benefit of the *prefetcher* hardware module presented in Section 3.3.

4.4.1 Increasing the Size of LLC Cache

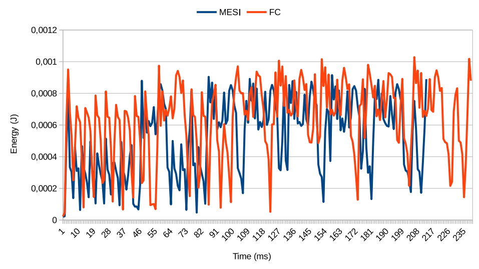
FlexiCache allows to adaptively switch-off and on the LLC banks to save energy without affecting the execution time, thus the use of bigger LLC banks allows to increase the saved energy. Figure 4.6 reports four statistics for all the considered benchmarks where each LLC bank is 1 MB: the average number of active LLC banks, the energy-delay-product (EDP), the total energy and the execution time. The comparison of the reported data with the results in Figure 4.2 that are extracted considering an LLC bank size of 256kB highlights three considerations. First, the average number of active banks lowers with the LLC bank size. For example *barnes* requires 9 and 5 LLC banks on average using an LLC bank size of 256kB and 1MB, respectively. However, few benchmarks, e.g. *ocean_cp* and *ocean_ncp*, does not benefit



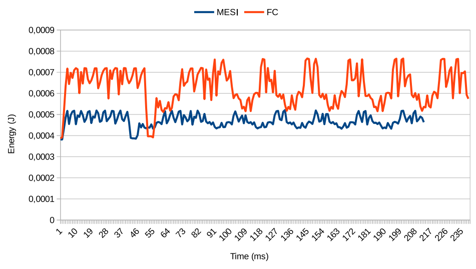
(a) OCEAN_CP On LLC Banks per epoch



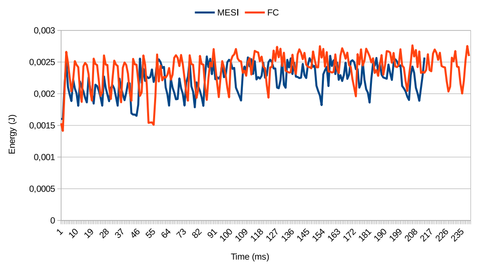
(b) OCEAN_CP Cache Energy Consumption per epoch



(c) OCEAN_CP Memory Energy Consumption per epoch

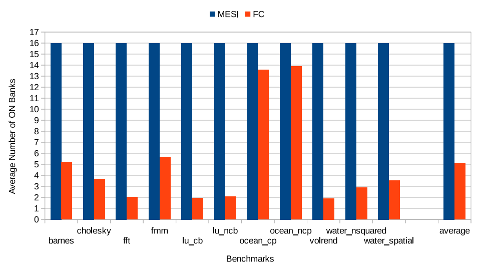


(d) OCEAN_CP NoC Energy Consumption per epoch

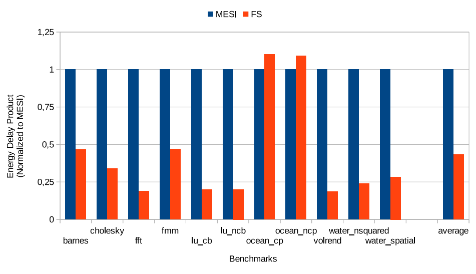


(e) OCEAN_CP Total Energy Consumption per epoch

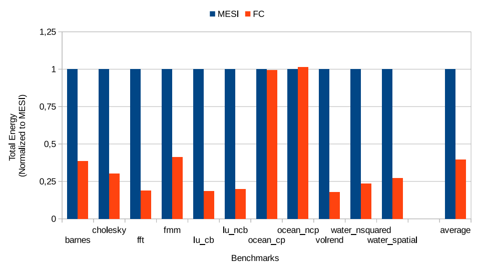
Figure 4.5: Behavior of Ocean_Cp benchmark for what concerns the Number of ON LLC Banks, Cache/Memory/NoC Energy and Total Energy Consumption with Prefetch mechanism and 256kB bank size.



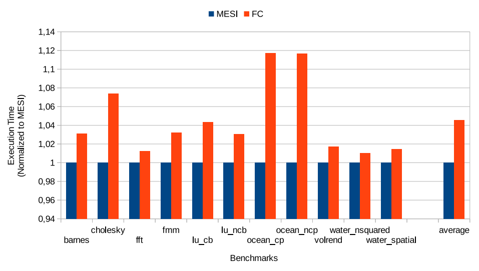
(a) Average Number of ON Banks



(b) Energy-Delay Product



(c) Total Energy Consumption



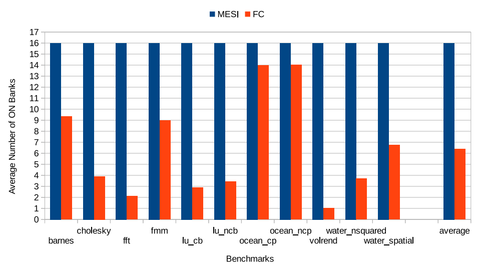
(d) Execution Time

Figure 4.6: Behavior of all benchmarks for what concerns the Average Number of ON LLC Banks, Total Energy Consumption and Execution Time with Prefetch and 1MB bank size.

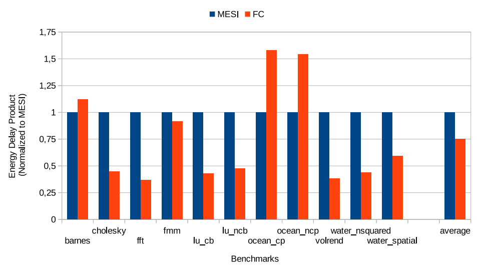
from the increased LLC size since their required memory is still bigger than it. Second, both the EDP and the total energy consumption is lower when a bigger LLC bank size is implemented. In particular, the EDP improves for *ocean_cp* and *ocean_ncp* move from 1.3 and 1.25 to 1.1 for both, thus assessing the benefit of a bigger LLC. Third, it can't be noticed significant differences from the execution time view point. Only for *volrend* the execution time really improves, passing from 1.08 to 1.02. For the remainings benchmarks a slight degrade can be noticed. The performance degradation with a bigger LLC could due to the higher contention to the cache controllers, as to the limitations of our policy.

4.4.2 The Impact of LLC Prefetch Scheme

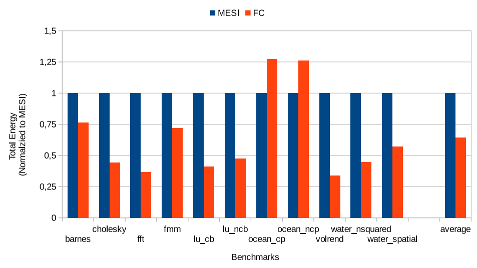
FlexiCache exploits a multicast mechanism to retrieve LLC data on an L1 miss since the possible switch on and off of different LLC banks make the behaving as a DNUCA. However, the multicast actually probes the active LLC banks to obtain the required cache line without enforcing any memory



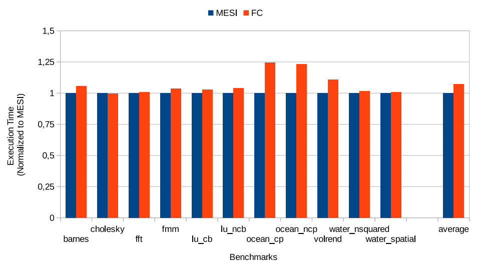
(a) Average Number of ON Banks



(b) Energy-Delay Product



(c) Total Energy Consumption



(d) Execution Time

Figure 4.7: Behavior of all benchmarks for what concerns the Average Number of ON LLC Banks, Total Energy Consumption and Execution Time WITHOUT Prefetch, 256kB bank size.

fetch transaction. In particular, each multicast request blocks at the LLC receiving banks that responds to the requestor with an ack/nack message eventually adding the requested data if present.

The memory fetch is only triggered by a unicast request that is issued once the original requestor has collected all the multicast nack messages, thus when it is safe to retrieve a new copy of the requested data from the main memory without duplicating it in the cache hierarchy. Such a behavior can strongly delay the actual memory fetch thus potentially impact the execution time. To this extent, the *LLC Data Prefetching* scheme has been described in Section 3.3 as a technique to allow the prefetching of the required data from memory at the time of the multicast, to partially shadow the memory latency.

In order to show the benefit of the proposed *LLC Data Prefetching* technique, this section discusses the simulation results of the *FlexiCache* architecture that does not implement the *LLC Data Prefetching* scheme as reported in Figure 4.7. The prefetching scheme strongly impacts all the considered metrics while not implementing it has a detrimental effect on the entire system. In particular, the execution time is negatively affected due to the increased latency before actually issue the memory fetch on an LLC miss. This effect is magnified for those benchmarks that report an high number of LLC misses, i.e., *ocean_cp* and *ocean_ncp* with a EDP degradation of 52% and 51% respectively when the prefetcher is not implemented. This is due to the increasing in the Total Execution Time, which increases from 11% and 10% to 25% for both. A slightly downgrade can be seen also for *barnes* and *fmm*.

Chapter 5

Conclusions and Future Works

The behavior of the applications influences in a critical way the utilization of the Last Level Cache, depending on their working set size. For that applications that fit entirely in the LLC and do not need all the available space, adapting the cache to their requirement can be a chance to save energy without affecting the performance.

This work proposes FlexiCache, a novel cache architecture that allows to dynamically resize the LLC in a multi-core with physically split LLC banks that share a single address space. The LLC resize is achieved by powering Off and On the LLC banks.

A novel coherence protocol is introduced to correctly manage the LLC resize stage as well as the DNUCA architecture due to the dynamic change of the LLC configuration. The novel coherence protocol is complemented by an energy-aware policy and a complete signalling infrastructure to collect valuable information to compute the power commands.

The policy exploits a utility metric to compute the power command, i.e., ON or OFF, to be sent to the LLC banks. The utility metric sits on the hypothesis that the application stays stable within the observation time window. Thus, the number of replacements in the LLC represent the quantity that influences the energy of the interconnect, the memory and the cache hierarchy. In particular the number of replacements in the LLC is influenced by switching ON or OFF the LLC banks.

It is worth noticing that even if the application changes the phase of execution this model is still valid, since the dynamic of the application is always orders of magnitude higher than the dynamic of the actuation system, i.e., switch ON or OFF an LLC bank.

The assessment is achieved considering a 4x4 NoC-based architecture using the Splash2x benchmarks from the parsec3.0 suite [31]. In particular, three metrics are considered: the energy, the performance, i.e., execution time to completion, and the energy-delay-product. Compared to the MESI state of the art architecture, the results show an average use of 6-7 LLC banks for the considered applications. It can be seen an improvement in the Energy Consumption of 30% on average, with some applications reaching 50% of energy savings. Performance are slightly affected, with degradation of less than 5% on average. These two aspects translate in an improvement on the Energy Delay product on average of 30%, reaching for some benchmarks an improvement of 70%.

A Design Space Exploration is discussed, which shown an even better improvement in the Energy Delay Product of 50% on average with greater LLC banks. Architecture without the Prefetch improvement for Multicast mechanism has been explored, showing that by anticipating the fetch data request from memory the execution time can be reduced, consequently improving the performance. This can be clearly seen in the memory-bound applications, in which the execution time worsens from 11% to 25%, impacting negatively on the Energy Delay Product which reaches a 50% degradation w.r.t the 25% of the architecture whit Prefetch.

5.1 Future Works

The present work proposed a new mechanism which allows to resize the LLC at a bank-level granularity. The multicast mechanism can be improved, by adding cooperation between cache levels.

The policy architecture enables further optimizations that are left as future work. For example, the performance metric is not directly accounted in the optimization. However, the presented results demonstrated the low performance overhead of FlexiCache compared to the baseline MESI architecture.

The exploration of different algorithms to select the target LLC bank represents an additional research extension that is also left as future work. Last, the CPU energy should be included in the optimization to provide a complete system-wide optimization that accounts not only for the uncore but for the entire system.

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, 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.

[4] Alessandro Bardine, Manuel Comparetti, Pierfrancesco Foglia, Giacomo Gabrielli, and Cosimo Antonio Prete. Way adaptable d&#45;nuca caches. *Int. J. High Perform. Syst. Archit.*, 2(3/4):215–228, August 2010.

[5] Alessandro Bardine, Pierfrancesco Foglia, Giacomo Gabrielli, and Cosimo Antonio Prete. Analysis of static and dynamic energy consumption in nuca caches: Initial results. In *Proceedings of the 2007 Workshop on MEmory Performance: DEaling with Applications, Systems and Architecture*, MEDEA '07, pages 105–112, New York, NY, USA, 2007. ACM.

[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] S. Chakraborty, S. Das, and H. K. Kapoor. Performance constrained static energy reduction using way-sharing target-banks. In *2015 IEEE*

International Parallel and Distributed Processing Symposium Workshop, pages 444–453, May 2015.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] B. Fitzgerald, S. Lopez, and J. Sahuquillo. Drowsy cache partitioning for reduced static and dynamic energy in the cache hierarchy. In *2013 International Green Computing Conference Proceedings*, pages 1–6, June 2013.

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

[16] A. Hansson, N. Agarwal, A. Kolli, T. Wenisch, and A. N. Udipi. Simulating dram controllers for future system architecture exploration. In

2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 201–210, March 2014.

[17] 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.

[18] S. Kaxiras, Zhigang Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *Proceedings 28th Annual International Symposium on Computer Architecture*, pages 240–251, 2001.

[19] 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.

[20] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi. Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 694–701, Nov 2011.

[21] Naveen Muralimanohar and Rajeev Balasubramonian. Cacti 6.0: A tool to understand large caches.

[22] S. Nilakantan, K. Sangaiah, A. More, G. Salvadory, B. Taskin, and M. Hempstead. Synchrotrace: synchronization-aware architecture-agnostic traces for light-weight multicore simulation. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 278–287, March 2015.

[23] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 423–432, Dec 2006.

[24] M. Sato, R. Egawa, H. Takizawa, and H. Kobayashi. A voting-based working set assessment scheme for dynamic cache resizing mechanisms. In *2010 IEEE International Conference on Computer Design*, pages 98–105, Oct 2010.

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

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

[27] C. Sun, C. H. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L. S. Peh, and V. Stojanovic. Dsent - a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*, pages 201–210, May 2012.

[28] K. T. Sundararajan, V. Porpodas, T. M. Jones, N. P. Topham, and B. Franke. Cooperative partitioning: Energy-efficient cache partitioning for high-performance cmps. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12, Feb 2012.

[29] 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.

[30] 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.

[31] Xusheng Zhan, Yungang Bao, Christian Bienia, and Kai Li. Parsec3.0: A multicore benchmark suite with network stacks and splash-2x. *SIGARCH Comput. Archit. News*, 44(5):1–16, February 2017.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.