UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

# Study and performance analysis of cache-coherence protocols in shared-memory multiprocessors

Dissertation presented by
**Anthony GÉGO**

for obtaining the Master's degree in
**Electrical Engineering**

Supervisor(s)
**Jean-Didier LEGAT**

Reader(s)
**Olivier BONAVENTURE, Ludovic MOREAU, Guillaume MAUDOUX**

Academic year 2015-2016

**Abstract**

Cache coherence is one of the main challenges to tackle when designing a shared-memory multiprocessors system. Incoherence may happen when multiple actors in a system are working on the same pieces of data without any coordination. This coordination is brought by the coherence protocol : a set of finite states machines, managing the caches and memory and keeping the coherence invariants true.

This master's thesis aims at introducing cache coherence in details and providing a high-level performance analysis of some state-of-the art protocols. First, shared-memory multiprocessors are briefly introduced. Then, a substantial bibliographical summary of cache coherence protocol design is proposed. Afterwards, gem5, an architectural simulator, and the way coherence protocols are designed into it are introduced. A simulation framework adapted to the problematic is then designed to run on the simulator. Eventually, several coherence protocols and their associated memory hierarchies are simulated and analysed to highlight the performance impact of finer-designed protocols and their reaction faced to qualitative and quantitative changes into the hierarchy.

**Résumé**

La cohérence des caches est un des principaux défis auxquels il faut faire face lors de la conception d'un système multiprocesseur à mémoire partagée. Une incohérence peut se produire lorsque plusieurs acteurs manipulent le même jeu de données sans aucune coordination. Cette coordination est apportée par le protocole de cohérence : un ensemble de machines à états finis qui gèrent les caches et la mémoire et qui s'assurent de la validité des invariants garantissant la cohérence.

Ce mémoire a pour objectif de présenter la cohérence des caches de manière détaillée et de fournir une analyse des performances globale de plusieurs protocoles formant l'état de l'art. Tout d'abord, les systèmes multiprocesseurs à mémoire partagée sont brièvement présentés. Ensuite, un large résumé issu d'une recherche bibliographique sur le domaine de la cohérence des caches est proposé. Par après, gem5, un simulateur d'architectures informatiques, et la manière dont sont programmés les protocoles à l'intérieur de celui-ci sont présentés. Un environnement de simulation adapté au problème étudié est ensuite conçu pour son exécution à travers le simulateur. Enfin, plusieurs protocoles de cohérence ainsi que les hiérarchies de mémoires associées sont simulés et analysés afin de mettre en évidence l'impact en terme de performance d'une conception plus raffinée de ces protocoles ainsi que leur réaction face à des modifications qualititatives et quantitatives de la hiérarchie.

# Acknowledgements

I first would like to thank my supervisor, Pr. Jean-Didier Legat, for the time spent listening to my progress and my difficulties, his advices, his feedback and his encouragement.

I also would like to thank my friends Simon Stoppele, Guillaume Derval, François Michel, Maxime Piraux, Mathieu Jadin, Gautier Tihon for their encouragement during this year.

Eventually, I also would like to thank Pierre Reinbold and Nicolas Detienne for allowing me to run the simulations carried out for this master's thesis on the INGI infrastructure, saving a significant amount of time for obtaining results.

# List of abbreviations

**DMA** Direct Memory Access. 14

**DSL** Domain Specific Language. 2

**FSM** Finite States Machine. 15

**ISA** Instruction-Set Architecture. 51

**KVM** Kernel-based Virtual Machine. 2, 52

**L0** Level 0 cache. 57

**L1** Level 1 cache. 56

**L2** Level 2 cache. 56

**LLC** Last-Level cache. 13

**LRU** Least Recently Used. 5

**MIMD** Multiple Instruction-stream Multiple Data-stream. 8

**MMIO** Memory Mapped Input/Output. 54

**NUMA** Non-Uniform Memory Acess. 10

**PPA** Performance-Power-Area. 78

**ROI** Region of Interest. 59

**SMP** Symmetric Multiprocessor. 10

**SWMR** Single-Writer-Multiple-Readers. 14

**TLB** Translation Look-aside Buffer. 14

**TSO** Total Store Order. 12

**UART** Universal Asynchronous Receiver Transmitter. 53

**UMA** Uniform Memory Access. 10

# Contents


**1 Introduction** **1**

**2 Reminder on caches** **3**
2.1 Spatial and temporal locality . . . 3
2.2 Cache internal organization . . . 3
2.2.1 Direct mapped cache . . . 4
2.2.2 Fully associative cache . . . 4
2.2.3 N-way associative cache . . . 5
2.3 Replacement strategies . . . 5
2.4 Block size and spatial locality . . . 6
2.5 Reducing the miss rate . . . 6
2.6 Writing data to memory . . . 7

**3 Shared-memory multiprocessors** **8**
3.1 Interconnection networks . . . 8
3.1.1 Shared bus . . . 8
3.1.2 Crossbars . . . 9
3.1.3 Meshes . . . 9
3.2 Memory hierarchies . . . 10
3.3 Shared memory correctness . . . 11
3.3.1 Consistency . . . 11
3.3.2 Coherence . . . 12

**4 Cache coherence protocols** **13**
4.1 Definitions . . . 14
4.1.1 Coherence definition . . . 14
4.1.2 Coherence protocol . . . 15
4.2 Coherence protocol design space . . . 17
4.2.1 States . . . 17
4.2.2 Transactions . . . 19
4.2.3 Design options . . . 19
4.3 Snooping coherence protocols . . . 20
4.3.1 An MSI snooping protocol . . . 21
4.3.2 A MESI snooping protocol . . . 24
4.3.3 A MOSI snooping protocol . . . 24
4.3.4 A non-atomic MSI snooping protocol . . . 26
4.3.5 Interconnect for snooping protocols . . . 30
4.4 Directory coherence protocols . . . 30
4.4.1 An MSI directory protocol . . . 30
4.4.2 A MESI directory protocol . . . 34
4.4.3 A MOSI directory protocol . . . 35

4.4.4 Directory state and organization . . . 36
4.4.5 Distributed directories . . . 39
4.5 System model variations . . . 39
4.5.1 Instruction caches . . . 39
4.5.2 Translation lookaside buffers (TLBs) . . . 40
4.5.3 Write-through caches . . . 40
4.5.4 Coherent direct memory access (DMA) . . . 40
4.5.5 Multi-level caches and multiple multi-core processors . . . 41

**5 Workload-driven evaluation 42**
5.1 The gem5 architectural simulator . . . 42
5.1.1 CPU, system and memory models . . . 42
5.1.2 Ruby memory model . . . 43
5.1.3 SLICC specification language . . . 44
5.2 The SPLASH2 and PARSEC3 workloads . . . 49
5.2.1 SPLASH2 benchmark collection . . . 49
5.2.2 PARSEC3 benchmark collection . . . 50
5.3 Choosing the simulated Instruction-Set Architecture (ISA) . . . 51
5.4 Making the simulation framework . . . 52
5.4.1 Cross-compilation versus virtual machine . . . 52
5.4.2 Configuring and compiling a gem5-friendly Linux kernel . . . 53
5.4.3 Configuring a gem5-friendly Linux distribution with PARSEC . . . 53
5.4.4 Integrating the gem5 MMIO into PARSEC for communication . . . 54

**6 Analysis of common coherence protocols and hierarchies 55**
6.1 Simulation environment . . . 55
6.2 Proposed hierarchies and protocols . . . 56
6.2.1 One-level MI . . . 56
6.2.2 Two-Level MESI . . . 56
6.2.3 Three-Level MESI . . . 57
6.2.4 Two-Level MOESI . . . 58
6.2.5 AMD MOESI (MESIF) Hammer . . . 58
6.3 Overall analysis . . . 59
6.3.1 Execution time . . . 59
6.3.2 Memory accesses . . . 60
6.3.3 Network traffic . . . 61
6.3.4 Quantitative hierarchy variations . . . 61
6.4 Detailed protocol analysis . . . 64
6.4.1 One-level MI . . . 64
6.4.2 Two-level MESI . . . 66
6.4.3 Three-level MESI . . . 68
6.4.4 Two-level MOESI . . . 71
6.4.5 AMD MOESI (MESIF) Hammer . . . 73
6.5 Concluding remarks . . . 75

**7 Conclusion 77**

**A Workbench configuration 81**

**B Simulated protocols tables 85**

**C Simulation distribution and analysis 86**

# Chapter 1

# Introduction

In the mid-1980s, the conventional DRAM interface started to become a performance bottleneck in high-performance as well as desktop systems [22]. The speed and performance improvements of microprocessors were significantly outpacing the DRAM speed and performance improvements. The first computer systems to employ a cache memory, made up of SRAM, that directly feeds the processor, were then introduced.

Because the cache can run at the speed of the processor, it acts as a high-speed buffer between the processor and the slower DRAM. The cache controller anticipates the processor memory needs and preloads the high-speed cache memory with data which can then be retrieved from the cache rather than the much slower main memory.

Nowadays, while significant speed and performance improvements have been brought to DRAM (reaching up to 4.2 billion transfers per second with DDR4 SDRAM [22]), this trend remains topical. Moreover, the need for faster and faster computer systems led to a technological bottleneck in the last decades. After being restricted to mainframes for almost two decades, multiprocessors were introduced in the desktop and more recently in embedded systems as chip multiprocessors.

For performance reasons, shared-memory multiprocessors ended up dominating the market. However, sharing memory across different processors that may operate on the same data introduces several design challenges, such as powerful and scalable interconnection networks, and memory correctness, especially when those processors have private caches.

Memory correctness defines what is correct for a processor to observe from the memory. When multiple processors manipulate the memory, all the instructions are interleaved from the shared memory point-of-view and defining correctness first consists in defining what kind of interleavings are permitted by the system. Moreover, most systems would ensure that each processor is able to access to an up-to-date version of each piece of data at any time. With the multiple private caches that are spread across the system, this is not an easy task. This last problem is referred to as cache coherence.

This master's thesis addresses a particular interest to this last problem, and proposes an in-depth study of cache coherence and cache coherence protocols, as well as the design and evaluation of these protocols in the gem5 simulator. In more details:

- Chapter 2 consists in a short reminder on caches. The basic concepts, cache organizations, replacement strategies and design options are reminded.
- Chapter 3 introduces shared memory multiprocessor systems and make an overview of their key design aspects : interconnection networks, memory hierarchies and memory correctness, and discuss the memory consistency and cache coherence problems.
- Chapter 4 proposes an in-depth study of the cache coherence mechanisms. Cache coherence is formally defined and the key structures to solve the problem are introduced. The

design space is presented and different kinds of protocols are discussed with concrete examples. Eventually, system model variations that must be taken into account during the design of a cache coherence protocol are discussed.

- Chapter 5 discusses the methodology used to assess the different protocol performances. The field of computer architecture is becoming increasingly quantitative and performance characteristics are isolated by using benchmarks run on simulators. The gem5 simulator and the PARSEC/SPLASH2 benchmarks are be introduced. Their characteristics and the limitations of possible simulations are then discussed. Eventually, the whole methodology used to run the PARSEC/SPLASH2 on the gem5 simulator and produce a disk image with appropriate communication with the simulator is explained.

- Chapter 6 describes some topical memory hierarchies and coherence protocol schemes and provides simulation results of the benchmarks. The dummy Valid/Invalid coherence scheme is simulated as a reference. Then, more complex protocols are simulated under different conditions. Both architectural and protocol trade-offs are discussed and their impact on the system performances (execution time, memory accesses, network traffic) are highlighted.

This work is aimed at providing an initiating material for people interested in the field of cache coherence, quickly starting designing their own protocols, and being able to evaluate their performance without diving into tricky software aspects. In more details, it brings the following contributions:

- A bibliographical research and a comprehensive summary about cache coherence, the design of protocols used to solve it, and some of their implementation aspects, only requiring basic knowledge of computer architecture.

- An introduction to gem5 and its coherence Domain Specific Language (DSL), to enable the retro-engineering and design of coherence protocols on the simulator.

- An updated gem5-compatible x86 Linux kernel binary integrating Kernel-based Virtual Machine (KVM) functionalities, and a ready-to-use image disk containing the whole PARSEC and SPLASH2 benchmark suites and the gem5 interoperability API for communication, built with a architecture independent methodology. Details for reproduction are available in Appendix A.

- An analysis tool, written in MATLAB, to parse gem5 simulation results and produce comparison graphs between numerous system variations, according to the specified simulation results to study, as well as a small parsing tool, written in Python, for converting gem5 cache coherence protocols documentation from HTML/Javascript into LaTeX format. Details are available in Appendices B and C.

- A performance analysis of some common state-of-the-art memory hierarchies and coherence protocols running the PARSEC/SPLASH2 parallel benchmarks, highlighting the impact of main architectural decisions.

# Chapter 2

# Reminder on caches

Computer processors and memories communicate via an interconnect, which is most of the time a bus in single-chip processor designs. In those designs, either the processor speed is a multiple of the bus speed, either the memory is slower than the processor, or both. To alleviate this performance flaw, computer designers have added caches to the processor.

These caches are made from faster but more expensive technology (SRAM) and hold used memory data. As the cache capacity is smaller than that of main memory, only a subset can be kept in the cache. When attempting to access data, the processor will therefore first checks the cache for data. If the cache hits, the data is available immediately. If it misses, the data must be fetched from the main memory and is placed in the cache for future use. When the cache is full, to accommodate new data, the old one must be replaced.

Managing the cache data is not done magically and is the role of the cache controller. In this chapter, the spatial and temporal locality of memory access, the different cache internal organizations and the replacement strategies will be reminded.

## 2.1 Spatial and temporal locality

A cache would ideally anticipate all the processor data needs and pre-fetch them in main memory such that it would have a zero miss rate. However, it is impossible to predict the future with perfect accuracy. The needed data can be guessed based on the past memory accesses, exploiting temporal and spatial locality, to achieve a low miss rate.

Temporal locality means that the processor is likely to access data that it has already accessed recently. A loop counter, for example, always increment the same piece of data. When the processor loads or stores data that is not in the cache, the data is copied from the main memory to the cache and subsequent requests for that data will hit in the cache.

Spatial locality means that when the processor accesses a certain piece of data, it is also likely to access data in the vicinity of the former. Arrays, for example, are contiguous data that are often processed sequentially.

Therefore, several adjacent memory words can be fetched along with a specific word access. This group of words is called a cache block, and its size is called the block size. A cache of capacity C contains $B = C/b$ blocks.

## 2.2 Cache internal organization

A cache is organized into $S$ sets, which can hold one or more blocks of data and each memory address maps to exactly one set in the cache. Some of the address bits are used to determine the set containing the data. If more than one block are contained in a set, the data may be kept

in any of the blocks in the set. Three categories of caches exist based on the number of blocks in a set.

### 2.2.1 Direct mapped cache

A direct mapped cache has one block in each set. The first $S$ memory addresses are directly mapped onto the $S$ cache blocks, and then the mapping wraps around, and the $S$ next memory addresses also maps onto the $S$ cache blocks.

As there are many memory addresses mapped to a same set, the actual address of the data contained in each set must be tracked. The least significant bits (LSB) of the address are used to specify the cache set that holds the data. The remaining bits, called the tag, are stored to indicate the actual memory address.

In a byte-addressable memory system with 32-bit addresses and 32-bit words and caches with an eight-word capacity and one-word block size, the two LSB of the address form the byte offset. The next three bits form the set bits and determine the sets holding the data. The remaining 27 bits form the tag and indicate the actual memory address of the held data. To indicate whether the data in a set is meaningful, a validity bit is used. If this bit is 0, the content has no significance. This happens, for example, at the computer startup, when no data has been loaded in the cache.

Figure 2.1 illustrates the address scheme, mapping, and validity bit as well as a possible hardware implementation that may be constructed as an eight-entry SRAM. When a load occurs, the cache checks the tag and validity bit of the decoded entry. If the tag matches the 27 adress MSB and the validity bit is 1, the cache hits and the data is returned to the processor. Otherwise, the cache misses and data must be fetched from the main memory.

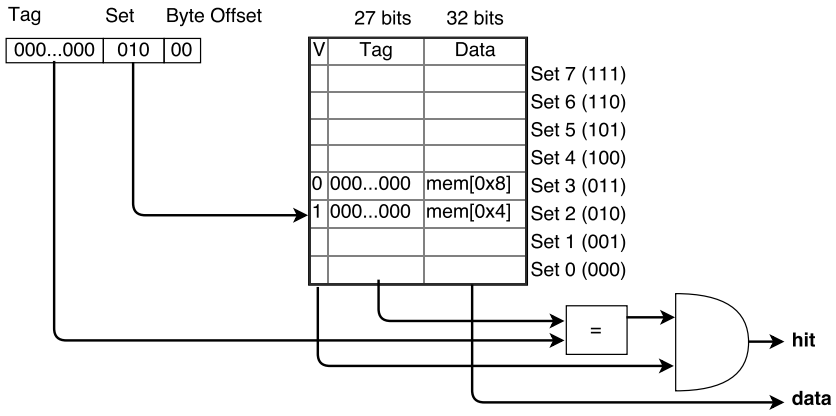



Figure 2.1 – Direct mapped cache with 8 sets

When two accessed addresses map on the same cache block, a conflict occurs, and the most recently accessed data replaces the older one. In direct mapped caches, as there are one block per set, a conflict happens every time the two accessed address map to the same set.

### 2.2.2 Fully associative cache

A fully associative cache is made of a single set. This set has $B$ ways, meaning that it contains B blocks of data. A memory address can then be mapped to a block in any of thee ways.

The cache array structure is illustrated in Figure 2.2. When a memory address is accessed, as may tag comparisons as the number of ways must be made, as data could be in any of the cache blocks. A multiplexer then chooses the appropriate data if the cache hits.

As all memory addresses map to the same set, fully associative caches have the fewest conflict misses for a given cache capacity, but however need more hardware for tag comparison,

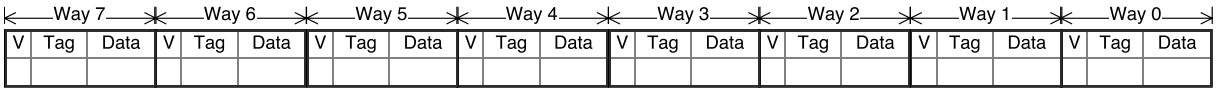



Figure 2.2 – Content of a fully associative cache with 8 blocks

making them bet suited for small caches. They however address the problem of how the data must be replaced in case of a conflict. This is described in Section 2.3.

### 2.2.3 N-way associative cache

An N-way associative cache, where N is said to be the degree of associativity of the cache, is a mix of a direct mapped cache and fully associative cache. It provides N blocks in each of its sets. Therefore, memory addresses still map to specific sets according to their least significant bits, but then map to any of the N blocks in the set.

The structure of an 8-word cache with a degree of associativity of $N = 2$, and its possible implementation, are illustrated in Figure 2.3. In this case, the cache has $S = 4$ sets and two bits are used to select the set, while the tag is 28-bits long. When a memory address is accessed, the cache reads the blocks from all the ways of the selected set and checks the tags and validity bits. A multiplexer selects the appropriate data if a cache hit occurs.

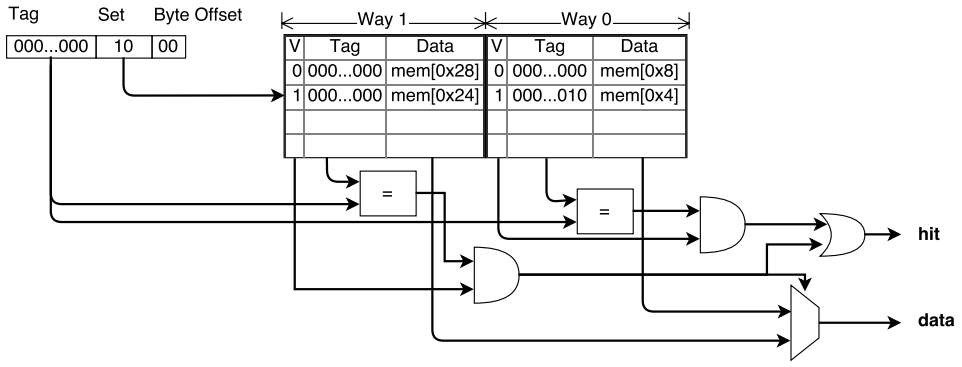



Figure 2.3 – 2-way associative cache with 8 blocks

N-way associative caches are a compromise between direct mapped caches and fully associative caches in terms of conflict. Indeed, the multiple ways allow reducing the number of conflicts while the different sets allow sparing hardware parts such as comparators. Indeed, in a N-way associative cache, only N comparators are needed.

## 2.3 Replacement strategies

When a conflict occurs in a direct mapped cache, as each address maps to a unique block and set, the only way to deal with a full cache set is to replace the block in the set with the new data.

In case of an N-way associative or fully associative cache, a block must be chosen for eviction when such a conflict occurs and the set is full. The temporal locality principle expressed earlier suggests than the most appropriate block to evict is the Least Recently Used (LRU) one, as it is the least likely to be used again in the near future.

This replacement scheme can be implemented by adding a use bit next to the validity bit and indicates if the block in the corresponding way was the most recently accessed. When one of the ways is used, this bit is updated, and the way having this bit high is the one replaced when needed. For a two-way cache, the least recently used block is actually replaced. In caches

with higher degree of associativity, a random block having the used bit low is replaced. Such a policy is called pseudo-LRU, and is often good enough in practice.

## 2.4 Block size and spatial locality

Using a one-word block only takes advantage of temporal locality, but not spatial locality. Indeed, only recently accessed memory locations are put into the cache, while in typical computer programs, memory locations in the neighbourhood of an accessed piece of data are also likely to be used in the near future. To exploit this, larger blocks can be used, to hold several consecutive memory words.

The main advantage is that, when a word is put in the cache, adjacent words are fetched in the same time and, therefore, subsequent accesses are more likely to hit. However, larger blocks means that a fixed-size cache will have fewer blocks, possibly leading to more conflicts. Moreover, the time required for fetching a missing block from the main memory will be longer as the adjacent words also have to be fetched.

A hardware implementation of a direct mapped cache with 2 sets and 4-word blocks in illustrated at Figure 2.4. In this case, the two least significant bits are still reserved for byte offset, the two next bits are used for identifying the block offset, the next bit for identifying the set and the remaining most significant bits form the tag. A multiplexer is needed to select the appropriate word within the block. As the words in the block are contiguous, only one tag is needed.

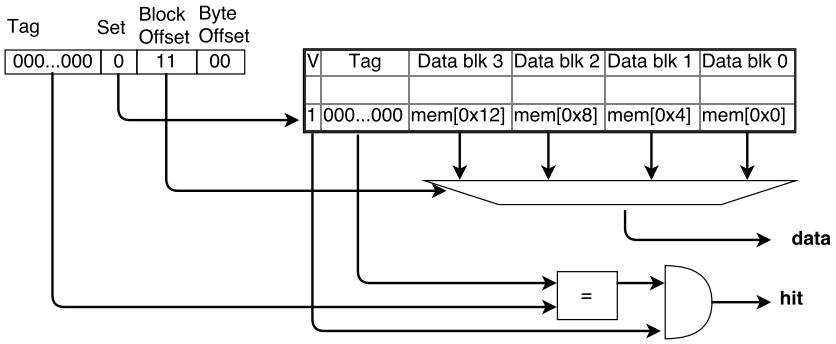



Figure 2.4 – Direct mapped cache with 2 sets and 4-word blocks

## 2.5 Reducing the miss rate

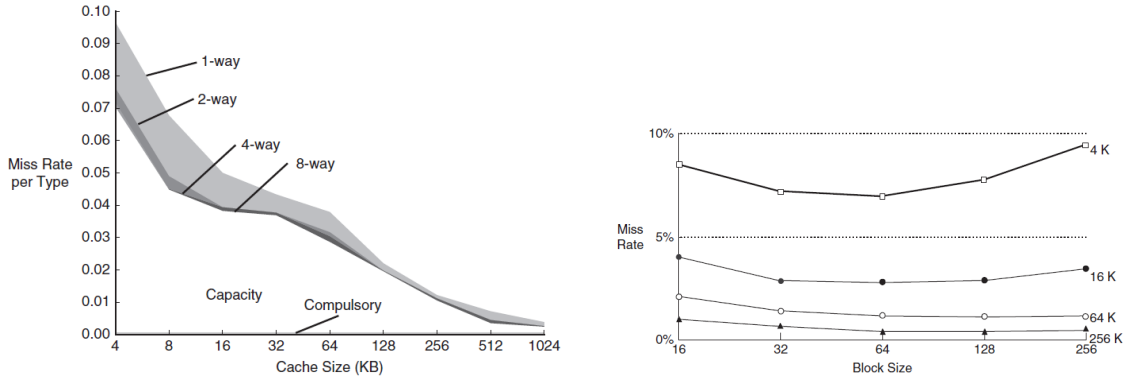
Cache miss rate can also be reduced by changing capacity, block size, and/or associativity. Those misses can be classified as :

- Cumpulsory miss: when the block has to be read from main memory whatever the cache design is.
- Capacity miss: when it is impossible to hold all the concurrently used data in the cache.
- Conflict miss: when multiple addresses map to the same set and evict used blocks.

Trade-offs can be made by changing the cache parameters. Increasing cache capacity, for instance, can reduce conflict and capacity misses, while increasing block size could reduce compulsory misses due to spatial locality and increase conflict misses (for a fixed-size cache). The best way to evaluate memory system performance, as it will be explained later, is by running benchmarks while varying the design parameters. This is illustrated in Figure 2.5a. The

dark region represents compulsory misses. Increasing the cache capacity reduces the capacity misses , while increasing the associativity decreases the conflict misses.

While using larger blocks takes advantage of spatial locality, they may increase the probability of conflicts. This is illustrated in Figure 2.5b. For small caches, increasing the block size beyond 64 byte increases the miss rate because of conflicts.




(a) SPEC2000 benchmark (b) SPEC92 benchmark

Figure 2.5 – Miss rate with varying parameters on SPEC benchmark. From [17]

Using larger caches allows achieving lower miss rate, but are more expensive and slower than small caches. Another solution is to use several levels of caches. Modern processors generally have two to three cache levels. The first level is designed to be small and fast, providing data within one or two cycles. The lower level caches are still built in the same technology but are larger and thus slower. In this design, the processor first looks for data in the first level. If the first level cache miss, the processor looks in the second level cache, and so fourth until a hit occurs or the data is fetched from the main memory.

## 2.6 Writing data to memory

When the data has to be stored, the processor first checks the cache. If the cache hit, the new data is simply written to the cache. Otherwise, the corresponding block is fetched from the main memory and the new data is written to the cache.

Caches can be classified according to their write behaviours :

- Write-through caches, in which data is simultaneously updated in the main memory.
- Write-back caches, in which a block is updated in the main memory after eviction if its dirty bit, indicating is data was modified, is high.

Modern caches are usually write-back because of the long main memory access time. In shared-memory multiprocessors, write-back caches significantly affect the way coherence is ensured, as the following chapter will show.

# Chapter 3

# Shared-memory multiprocessors

Technological barriers and advances urged processor designers to put multiple processor cores on the same single chip in the last decade. While multiprocessors systems were reserved for some high-end systems in the past, most modern publicly available computers now integrate chip multiprocessors.

In the Flynn taxonomy, parallel processors belongs to the category of Multiple Instruction-stream Multiple Data-stream (MIMD) computers, meaning that they process both multiple instruction streams as well as multiple data streams in parallel. In this design, several processors with their cache hierarchy asynchronously operate in parallel.

These processors can either share a common global memory (GM-MIMD) and communicate via loads and stores, or simply use a local memory and communicate via message passing (LM-MIMD). In this chapter, only GM-MIMD machines are considered. The different processors in those specific kind of machines must be connected together by means of an interconnection network, and can use different memory hierarchies. Moreover, sharing a common memory between the processor require to define and ensure the correctness of such a memory system. All those aspects are part of the design space of those machines.

In this chapter, interconnection networks and memory hierarchies in GM-MIMD machines are introduced. Then, the shared memory correctness problem is discussed and the consistency and cache coherence sub-problems are introduced.

## 3.1 Interconnection networks

Many kinds of interconnection networks exist and were used in many multiprocessor architectures. In this section, three common kind of interconnection networks are introduced : shared bus and crossbars-based networks, which are direct networks and meshes, which are indirect networks.

### 3.1.1 Shared bus

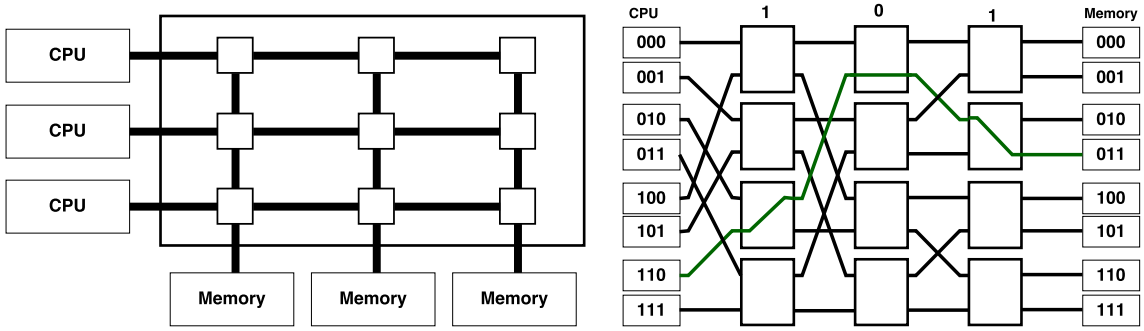
Shared bus is the simplest form of interconnection, and is illustrated in Figure 3.1. It has many advantages such as cost and ease of use but has severe limitations. It does not scale due to the physical and latency constraints limiting the number of devices connected together. Indeed, contention quickly arise as the probability that the bus is busy significantly increase with the number of processors.

While the speed of processors in the past enabled the use up to 20 processors on the same bus [1], modern chip multiprocessors, with their higher bandwidth, do not generally come with more than eight cores on a shared bus.

Figure 3.1 – Shared bus architecture

### 3.1.2 Crossbars

Crossbars are structures that can connect p processors to m memories (or other devices) with a concurrency equal to $\min(p, m)$. Figure 3.2a illustrates a typical crossbar. Each square cell is a controlled switch that either forms two individual up-down and left-right links or a left-down link. Crossbars are extensively used in network processors, for instance, where they provide the switch fabric for incoming and outgoing packets on different lines. There are also used in chip multiprocessors, in the IBM Power4 and Power5 for instance [1], but limit the area devoted to the interconnection network due to the large number of switches required.




(a) A crossbar network

(b) A Banyan network. Adapted from [2]

Figure 3.2 – Crossbar-based topologies

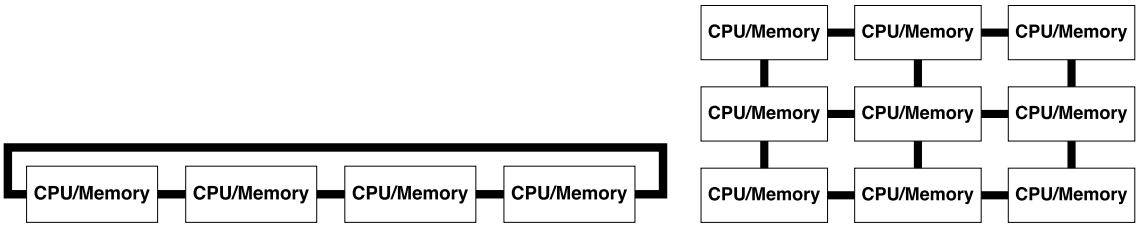
Crossbars can also be used to build complex direct interconnection networks such as the Banyan network [2] illustrated in Figure 3.2b. This multistage network consist of $\log n$ stages of $n/2$ switches each. These switches are 2x2 crossbars which will either directly link or cross-link the inputs and outputs if their control signal is 0 or 1, respectively. A example control signal is illustrated on the Figure. Compared to full crossbars, whose cost grows as $m \times n$, the cost of such a network grows as $n \log n$, with the drawback a connection takes $\log n$ time.

### 3.1.3 Meshes

Meshes are indirect interconnection networks, meaning that elements are connected to other elements at various distances. They are often differentiated by their dimensions.

- A ring, illustrated at Figure 3.3a, is a unidirectional network.
- A nearest-neighbour network, illustrated at Figure 3.3b, is a two-dimensional mesh where an element which is not at the edge is connected to four neighbours by a same-length link.
- A 2-D torus is a nearest-neighbour network with links wrapping around the edges. It also is a two-dimensional network
- An m-dimensional hypercube, where elements are connected by a same-length link to its of its m neighbours, is an m-dimensional network.

Message routing can be done in several ways. A common pattern is the higher dimension first, meaning that in a 2D mesh, the message first travels vertically. For chip multiprocessors,




(a) One-dimensional (ring) (b) Two-dimensional

Figure 3.3 – Mesh topologies

the flow control can be done by simple acknowledgements and retries. However, no commercial chip multiprocessors uses such indirect networks at this point [1], and only rare research machines for grid computing use 2D mesh.

## 3.2 Memory hierarchies

The parallel processors in a shared-memory system share a common address space and data exchange between the CPUs is achieved via traditional loads and stores. The main memory can either be situated at the same distance of all processors, called Uniform Memory Access (UMA), or be distributed among the processors, called Non-Uniform Memory Acess (NUMA).

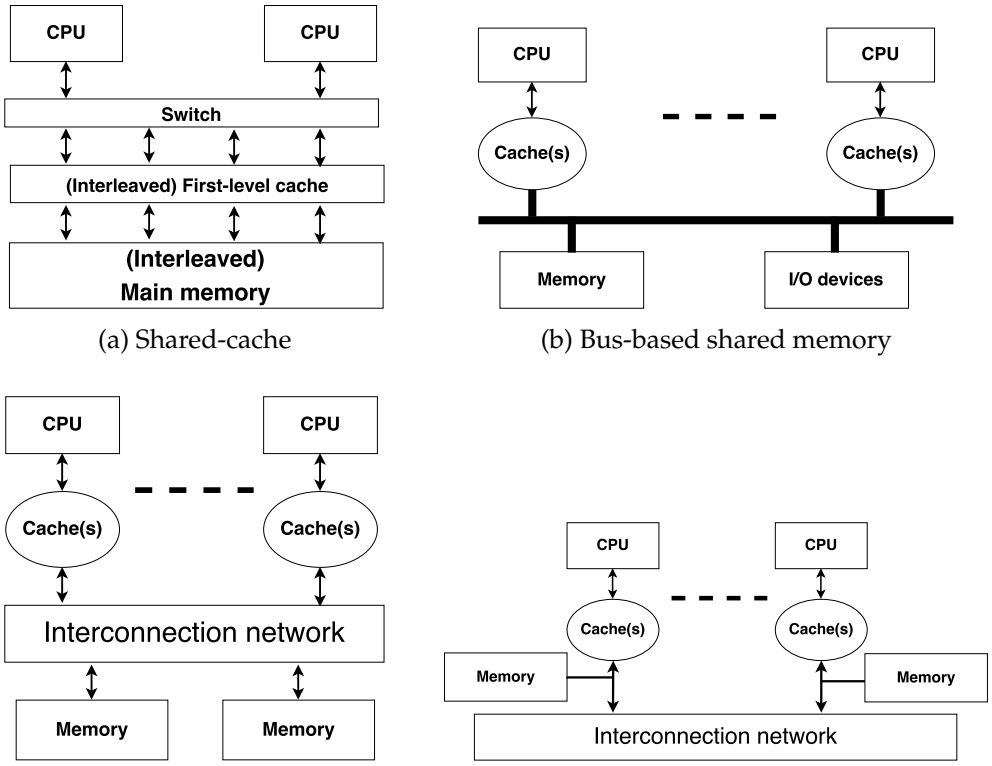
Early UMA designs consisted in a shared cache system [11], and is illustrated in Figure 3.4a. In this system, the interconnection network is located between the processor and the first level cache, which could be interleaved with main memory to increase the bandwidth. This design has poor performances as the complex interconnect is on the critical path determining the latency of cache access and the unique shared cache must deliver a tremendous bandwidth to the multiple processors accessing it.

A UMA system with a shared bus interconnection is illustrated in Figure 3.4b. In some implementations, the different actors can hare an additional bus for control purposes, synchronization, or non-atomic cache coherence, for instance. The processors in the Figure can also be a multiprocessor itself. This kind of organization is referred as a Symmetric Multiprocessor (SMP) system, although the do not need to be homogeneous, as long as they do not share the same ISA.

As it will be further discussed, shared bus topologies have some disadvantages coming from electrical constraints on the bus load and contention for accessing it, making it not scalable. A second common UMA architecture is illustrated in Figure 3.4c and is referred to as a dance-hall architecture. In this topology, the interconnect network can be of various type such as a crossbar or an indirect network.

The two previous architecture require putting the memory components far from the processors, increasing the non-cached access time even for data that are exclusively used by one processor. An alternative is to use distributed memory, as illustrated in Figure 3.4d, also called NUMA architectures. In this kind of architecture, each processor owns a part of the global memory. On a load or store, the local memory directly responds if it holds the requested data, otherwise, the request is sent trough the interconnection to the appropriate local memory of another processor.

Chip multiprocessors generally follows a shared-bus or more typically a dance-hall architecture, as the local memory is incorporated into chips, where their non-private caches are generally assimilated to the memory modules.




(a) Shared-cache

(b) Bus-based shared memory

(c) Dancehall

(d) Distributed memory

Figure 3.4 – Common shared-memory hierarchies. Adapted from [11]

## 3.3 Shared memory correctness

Shared memory systems provide various goodness properties such as high performance, low power and low cost. However, memory correctness must be first provided, and well mastered for their hardware implementation, where bug fixes are expensive. The problem of providing memory correctness can be divided into two sub-problems : consistency and coherence.

### 3.3.1 Consistency

The consistency memory model provides rules about loads (memory reads) and stores (memory writes) and how to act upon memory in order to ensure correctness. In a single-core processor, the correctness criterion partitions behaviour between one correct result and many incorrect alternatives, because the processor architecture mandates that the execution of a thread transforms a given input state into a single output state, even on an out-of-order core. In shared memory multi-core systems, due to the multiple threads running concurrently, this partitions behaviour between many correct executions and many more incorrect ones. Indeed, the threads may generate many correct interleavings of instructions.

The need for consistency models can be illustrated by the following real-world example. Consider the situation in which a professor wants to change his course project guidelines and ask his assistant to post the new guidelines on Moodle. While he's giving the lecture, the professor informs his students the new project guidelines are online. However, at that moment, the teaching assistant has not posted the updated document yet because he's still busy and some diligent students may download the old guidelines, even though the updates were made in the correct order. A consistency model defines if that kind of behaviour is correct or not.

Similar behaviours can happen in shared memory hardware with out-of-order processors, write buffers, prefetching, and multiple cache banks. Correctness must then be defined to

specify the allowed behaviour of multithreaded programs executing with shared memory, so that programmers know what to expect and system designers know the limits to what they can provide.

Several consistency models, cited from stronger to weaker, are found on the market:

- Sequential consistency, which states that a multi threaded execution should look like an interleaving of the sequential executions of the different threads, as if they were multiplexed on a single core processor. This is used for the MIPS R10000 for example.
- Total Store Order (TSO), which is designed for the use of FIFO write buffers holding the results of committed stores before the effective write to the caches. In this optimization, stores are just guaranteed to be executed in order, loads are not. This is implemented, for instance, on x86 and SPARC processors.
- Relaxed consistency, which relies on the observation that most memory orderings in stronger models are not necessary. Indeed, if a thread update a dozen data items and then set a synchronization flag, the programmer does not care if data were updated in the specified order in their code. Specific instructions are therefore needed to define memory load/store barriers and flush write buffers.

### 3.3.2 Coherence

Coherence is not visible to software and is, actually, not required. However, a vast majority of shared memory systems implement a coherence protocol that provides coherence. Coherence problems can happen when multiple processor cores have access to the same piece of data at the same time and that at least one of these cores is overwriting this piece of data.

Considering the previous example, imagine one of the students did not attend the class. As the professor only told that the updated guidelines were online during the class, this student now has a stale copy of the guidelines, and an incoherence occurs. Another examples of incoherence that take place in the computing world are stale web caches and programmers using non-updated code repositories.

The use of a coherence protocol prevents accessing such a stale piece of data. It consists in a set of rules implemented by the different actors within the system. The goal of coherence is actually to make the caches of a shared memory system as functionally invisible as the caches in a single-core system. That means that a programmer could not determine whether and where a system has caches by analysing the results of loads and stores, because coherence ensures the caches never enables new or different functional behaviour.

# Chapter 4

# Cache coherence protocols

Modern chip multiprocessors are based on the dance-hall shared memory architecture introduced in the last chapter. For this reason, this will be considered as the reference system model for the next sections, illustrated in Figure 4.1. The chip is composed of multiple single-threaded processors or processor cores having their own private write-back data caches and a Last-Level cache (LLC) shared by all the processors. The processors and the LLC communicate with each other via the interconnection network. In the next sections, the term "cache" will refer to the processors' private caches, as LLC is a memory-side cache and does not introduce another level of coherence issues. This LLC also serves as an on-chip memory controller.

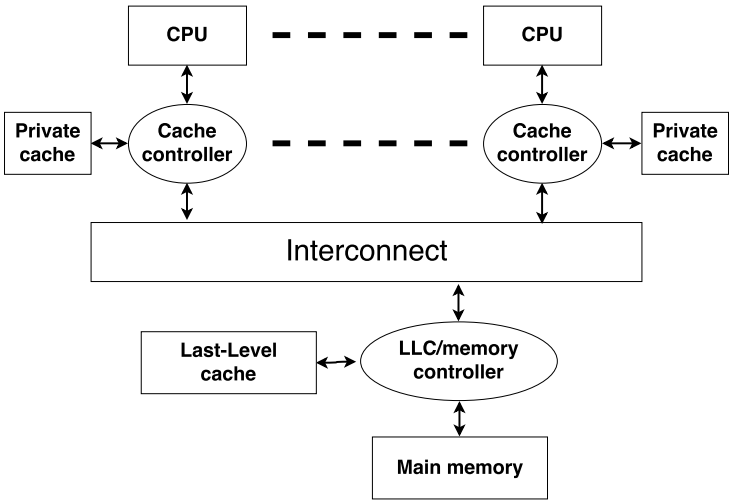



Figure 4.1 – Reference system model used

In this chapter, cache coherence and coherence protocols are defined. Then, the protocol design space is discussed and two protocol families are presented : snooping and directory-based protocols. Eventually, system model variations are discussed to handle commercial system cases. Indeed, the reference model does not consider, for instance, instructions caches, multiple-level caches or coherent direct memory access. A large part of the material in this Chapter comes from [11] and [23], and the protocol examples discussed throughout have been designed by Sorin et al. [23].

## 4.1 Definitions

### 4.1.1 Coherence definition

Incoherence happens only because of one issue : several actors have access to caches and memory. These actors are typically the multiple processors, Direct Memory Access (DMA) engines, and external devices that write/read to caches and memory. Figure 4.2 illustrates an example of incoherence. Two processors, having their own private caches, load memory address A and a copy of data at address A is thus stored in their respective caches. Then, CPU2 updates the value at address A and writes the value on its cache, making CPU1's value of A in its cache incoherent.

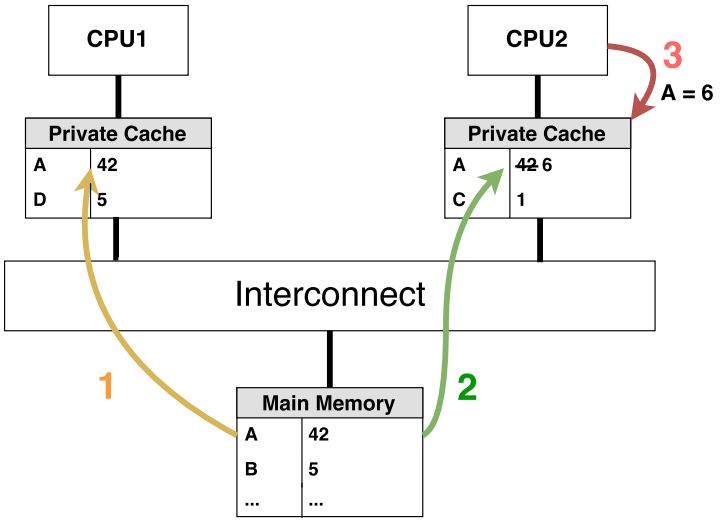



Figure 4.2 – Example of incoherence in a multiprocessor system

This can be intuitively qualified as "incorrect". To provide a formal definition of coherence, two invariants may be used, based on what is observed as incorrect:

- A Single-Writer-Multiple-Readers (SWMR) invariant which states that, at any moment in time, for a given memory location, there is either a single processor that may write or read it, or one or several processors that may read it. This invariant only need to be maintained in logical time and the memory location's lifetime can be seen as divided into epochs.
- A data invariant which states that the value of a memory location at the start of an epoch is the same as the value of that same memory location at the end of its last write and read epoch. This ensures that the value of a given memory location is correctly propagated, and that during read epochs, all processors will read the same value for that memory location.

The majority of coherence protocols, called "invalidate protocols" are designed explicitly to maintain these invariants and will be investigated in the next sections.

While processors perform reads and writes with various sizes, ranging from 1 to 64 bytes, in practice, coherence is commonly maintained at the granularity of cache blocks. Therefore, by applying the first invariant, in typical systems, it is not possible for a processor to write the part of a block while another processor is writing another part of that block.

Cache coherence also has a specific scope. All storage components that hold blocks from the shared address space are concerned. These include the different level caches, the main memory, the instruction caches and the Translation Look-aside Buffer (TLB). However, it does

not pertain to the architecture. The consistency model does not place any explicit constraint on coherence or the coherence protocol. Consistency models, on the contrary, may rely on some coherence properties for correctness.

### 4.1.2 Coherence protocol

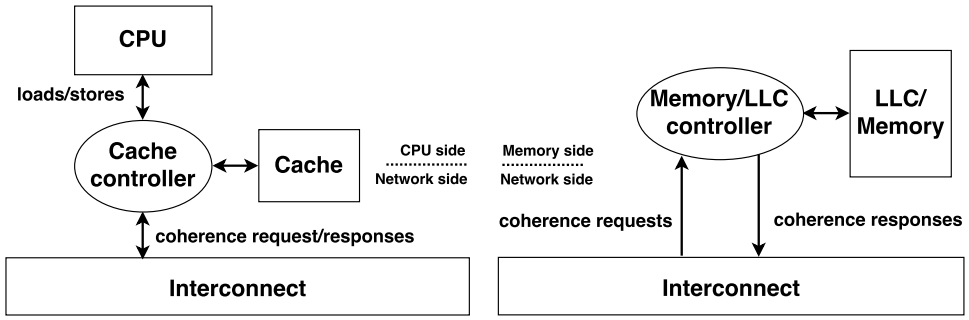
The two coherence invariants are implemented by associating with each storage component a Finite States Machine (FSM) called the coherence controller. All these components constitute a distributed system exchanging messages to ensure both the invariants are maintained at all times for each block.

The coherence protocol specifies the interactions between the finite state machines. The coherence controllers have several responsibilities. The cache controller of Figure 4.3a processes requests from two sources:

- The processor, from which it accepts loads/stores and returns load values to. When the cache misses, the controller starts a coherence transaction by issuing a coherence request (for read permission for instance) for the desired block to one more coherence controllers.
- The interconnection network (other controllers), from which it receives coherence requests and coherence responses that it must process.

A transaction consists of a request and all the messages needed to satisfy the request. The type of transactions and messages depends on the protocol specifications.

The memory controller of Figure 4.3b usually only processes requests from the interconnection network. It therefore does not issue requests nor receive coherence responses. Other components may act as a cache controller or a memory controller depending on their requirements.




(a) Cache controller (b) LLC/Memory controller

Figure 4.3 – Coherence controllers

A coherence controller is a finite state machine which receives and processes events depending on the block state. For an event of type E to block B, the controller takes actions that are function of E and of B state and may change the state of B.

These machines are specified by means of a graph, or a table, for convenience, in which rows correspond to block states and columns correspond to events. A state/event entry is called a transition and consists of the actions taken when the event occurs and a new state for the block. The transitions are expressed in the format "action/next state". The specifications of a coherence protocol can thus be described by providing such a table for each of the cache or memory controllers present in the system, and is illustrated below.

The most stupid coherence protocol that can be designed keeps the cache and the LLC/memory blocks in two possible stable states : Invalid (I) and Valid (V) as it was introduced

for the single-actor computer in Chapter 2. At the memory side, the I state means that all caches holds the block in the I state, and the V state means that one of the cache holds the block in V state. In this example, it is considered the caches are write-back and the interconnection network is a simple shared bus: all the processors and the LLC are connected together and messages are observed by everyone.

At start-up, all cache and memory blocks are in state I. The processors can then issues reads and writes to their cache controllers and the latter will generate an *Evict Block* event when room must be made. Cache misses will initiate coherence transactions in order to obtain a valid copy of the cache block.

Two types of transactions are implemented with three messages : *Get* for requesting a block, *DataResp* for transferring the block data and *Put* for writing back data to the memory. The *Get* transaction, initiated by a cache miss, is atomic.It waits for a *DataResp* message after sending a *Get* message. No other transaction can happen meanwhile. On an eviction, the cache controller sends a *Put* message with the entire cache block to the memory controller.

Transitions between the states are illustrated in Figure 4.4. Prefixes *Own* and *Other* indicates whether the request comes from the cache controller itself or another. If a cache controller receives a *Get* message from another cache controller for a block it holds in V state, it must send the block data to the requester and change the block state to I.

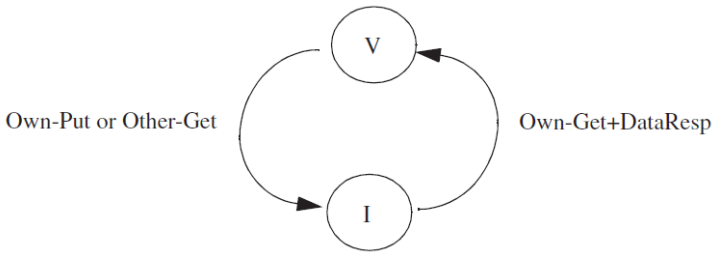



Figure 4.4 – Transitions between stable states at cache controller. From [23]

Tables 4.1 and 4.2 describes the complete protocol specifications. The shaded and blank entries respectively represent impossible transitions and ignored events.

| S/E | Load or Store | Evict Block | Own-Get | DataResp for Own-Get | Own-Put | Other-Get | DataResp for Other-Get | Other-Put |
|---|---|---|---|---|---|---|---|---|
| I | issue Get /$IV^D$ | | | | | | | |
| $IV^D$ | stall Load or Store | stall Evict | | copy data into cache, perform Load or Store /V | | | | |
| V | perform Load or Store | Issue Put (with data) /I | | | | Send DataResp /I | | |

Table 4.1 – Cache controller specifications. Adapted from [23]

| S/E | Get | Put |
|---|---|---|
| I | send data block in DataResp message to requester/V | |
| V | | Update data block in memory/I |

Table 4.2 – Memory controller specifications. Adapted from [23]

The state $IV^D$ is called a transient state and corresponds to a block in I state waiting for

data via a *DataResp* message before transitioning to V state. Transient states appear when the transitions between two stable states are not atomic. Indeed, the *Get* transaction requires more than one message to complete.

## 4.2 Coherence protocol design space

A cache coherence protocol designer must choose the states, transactions, events and transitions for each of the coherence controller in the system. While the stable states and transactions are independent of a specific protocol, the events, transitions and transient states are highly dependent on the protocol.

### 4.2.1 States

Chapter 2 described caches in a single-actor environment. While the Invalid and Valid states may be sufficient in multiple actors systems, different kinds of valid states can be distinguished. Cache blocks have four characteristics it is worth encoding in their states:

- Validity : A valid block has an up-to-date value. It may be read, but may only be written if it is also exclusive.
- Dirtiness : A dirty block has an up-to-date value which differs from the value in memory. The cache controller is responsible for eventually updating the value in the memory. A block that is not dirty is called a clean block.
- Exclusivity : A block is exclusive when it is the only **privately** cached copy of that block in the system.
- Ownership : A block is owned by a cache controller if it is responsible for responding to coherence requests for that block. This block cannot be evicted without giving the ownership to another coherence controller.

#### Stable cache states

In 1986, Sweazy and Smith [24] introduced a five-state MOESI model on which many coherence protocols nowadays still rely on. These states are formed from combinations of the previously defined characteristics. The three first main states are:

- M(odified) : The block is valid, exclusive, owned, may be dirty, and may be written or read. The cache has the only valid copy of the block and it is potentially stale at the memory. The cache is responsible for requests for the block.
- S(hared): The block is valid but not exclusive, not dirty, not owned and is read-only. The other caches may hold valid, read-only copies of the block.
- I(nvalid): The block is invalid. Either the cache does not hold the block or it holds a stale copy that it may not read or write.

In addition to this three first states, the MOESI set specifies an O state and a E state, which are used to optimize certain situations:

- O(wned): The block is valid, owned, and potentially dirty, but not exclusive, and read-only. It is potentially stale at the memory. The cache is responsible for requests for the block. The other caches may have a read-only copy but are not owners.

- E(xclusive): The block is valid, owned, exclusive, clean, and read-only. It is up-to-date at the memory. No other cache has a valid copy of the block.

For convenience, a Venn diagram of the MOESI states is illustrated in Figure 4.5. All states except I are valid states. M, O and E are owned states. M and E are exclusive states. M and O are potentially dirty states. This diagram also shows that the example IV protocol condensed the MOES states into the V state.

The MOESI states are quite common and may be called differently. However, they are not an exhaustive set. For instance, Intel is known to use a MESIF set, in which the F(orward) state is similar to the O state except that it is clean. Therefore, the memory has an up-to-date copy of the block.

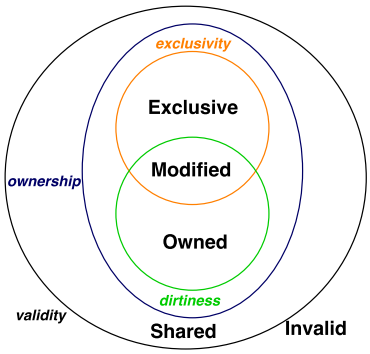



Figure 4.5 – Venn diagram of the MOESI states

Stables states are generally maintained in the caches by adding a few bits to the cache entry, as it was done in Chapter 2 for the single-actor system.

#### Transient cache states

The stable states occurs when there is no current coherence activity for a block, and are typically used for referring to a protocol. However, transient states may occur during the transition from a stable state to another. They can be noted $XY^Z$, which means that the block is transitioning from stable state X to stable state Y, but is waiting for an event Z to complete the transition.

As transient states are numerous and do not concern all blocks but only those that have pending transactions, they are maintained with specific hardware structures, miss status handling registers (MSHRs), used to track pending transactions.

#### LLC/Memory states

Block states in the LLC and memory can be named by two general approach, whose choice does not affect functionality or performance:

- Cache-centric: The block state in the LLC/memory is an aggregation of the block states in the caches. If a block is in all caches in I, then it is in memory in I. In the next sections, only cache-centric states will be considered.
- Memory-centric: The block state corresponds to the memory controller permissions on this block. If a block is in all caches in I, then it is in memory in O, because memory acts like an owner for the block.

As it is not possible to attach a memory status to all blocks in the system, many multiprocessor systems use an inclusive LLC, meaning that it maintains a copy of every block cached anywhere is the system. Thus, memory does not need to maintain states, and he state of a block in memory is the same as in LLC.

### 4.2.2 Transactions

As the basic goals of coherence controllers are similar, most protocols have a similar set of transactions. A set of common transactions and the goal of their requester are available in Table 4.3. They are all initiated by cache controllers.

| Transaction | Goal of requester |
|---|---|
| GetShared (GetS) | Obtain a block in shared (read-only) state |
| GetModified (GetM) | Obtain a block in Modified (read-write) state |
| Upgrade (Upg) | Upgrade a block from read-only (Shared/Owned) to read-write (Modified) state<br>Upg (unlike GetM) does not require data to be sent to requester |
| PutShared (PutS) | Evict block in Shared state (not if silent evicts) |
| PutExclusive (PutE) | Evict block in Exclusive state (not if silent evicts) |
| PutOwned (PutO) | Evict block in Owned state |
| PutModified (PutM) | Evict a block in Modified state |

Table 4.3 – Common transactions. Adapted from [23]

Table 4.4, lists requests that the processors can make to the cache controller and how these lead to the initiation of coherence transactions.

| Event | Response of cache controller |
|---|---|
| load | if cache hit, respond with data from cache; else initiate GetS transaction |
| store | if cache hit in state E or M, write data into cache; else initiate GetM or Upg transaction |
| atomic R/M/W | if cache hit in state E or M, atomically execute R/M/W semantics; else initiate<br>GetM or Upg transaction |
| instruction fetch | if cache hit (inclusive-caches), respond with instruction from cache; else initiate GetS transaction |
| read-only prefetch | if cache hit, ignore; else optionally initiate GetS transaction |
| read-write prefetch | if cache hit in state M, ignore; else optionally initiate GetM or Upg transaction |
| replacement | depending on block state, initiate PutS, PutE, PutO or PutM transaction |

Table 4.4 – Common processor requests to cache controller. Adapted from [23]

Although protocols use a similar set of transactions, they differ by the way coherence controllers interact to perform them. For example, some family of protocols can broadcast requests while another family can unicast requests to a specific controller.

### 4.2.3 Design options

Many different protocols can be designed from the same set of states and transactions as it is not possible to provide an exhaustive list of possible events and transitions for each coherence controller. However, two main decisions have a major impact on the rest of the protocol. For both these design decisions, making hybrids are possible.

#### Snooping and directory protocols

Two main classes of coherence protocols are commonly in use nowadays:

- Snooping protocols: In these protocols, cache controllers initiate a block request by broadcasting it to all other coherence controllers which will process it and reply, if needed, with data for instance. They rely on the interconnection network to deliver the messages in a consistent order, and most of them typically assume a total order obtained via a shared bus. Relaxed buses however exist.
- Directory protocols: In these protocols, cache controllers initiate a block request by unicasting it to the block home memory controller which looks into its directory which caches

are the current owner or sharers for that block. If the LLC/memory is the owner, it terminates the transaction by sending data to the requester. Otherwise, it forwards the request to the owner cache, which completes the transaction.

Snooping protocols are simple, but they do not scale to large numbers of processors as broadcasting does not scale. Directory protocols are scalable because they unicast, but many transactions take more time because they require an extra message to be sent when the home controller is not the owner. In addition, the choice of protocol affects the interconnection network as, for instance, classical snooping protocols require total order.

#### Invalidate and update protocols

Protocols can also be classified by the action performed when a processor writes to a block:

- Invalidate protocol: In these protocols, copies of a block are first invalidated before proceeding to a write operation. Therefore, no other processor can read the block while the first one is writing new value, and this other processor has to initiate a new coherence transaction to obtain that new value.
- Update protocol: In these protocols, the processor initiates the update of all the copies of the block after writing it, in order to reflect the new value of the block.

As the processor does not have to initiate and wait for a *Get* transaction to complete, update protocols reduce the latency to read a newly written block . However, they typically consume more bandwidth because update messages, containing data, are larger than invalidate messages, and they complicate the implementation of many consistency models. For instance, preserving write atomicity is much more difficult when multiple caches must apply multiple updates to multiple copies of a block. Because of that, they are rarely implemented.

## 4.3 Snooping coherence protocols

Snooping protocols were the first widely deployed class of coherence protocols and offer several attractive features such as low-latency transactions and a simple design.

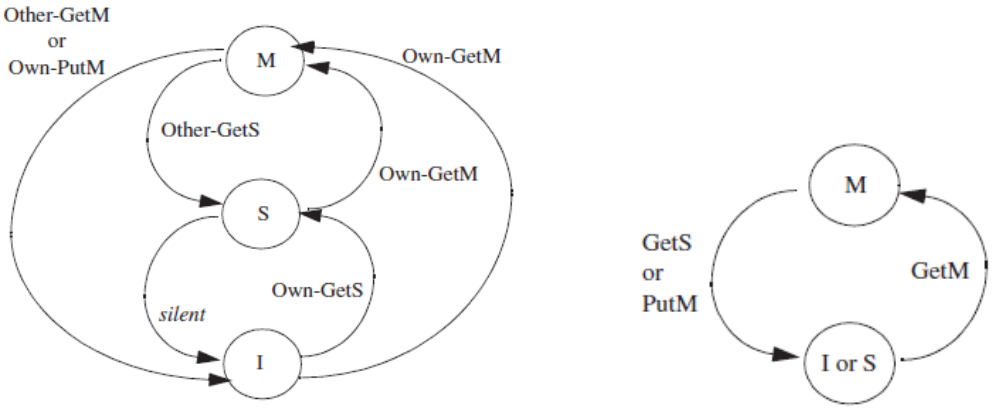
In such protocols, all the coherence controllers snoop the requests and process them in the same order. Thus, they all observe the same scenario and they can update correctly their finite state machines. A totally ordered broadcast network, such as a bus, is thus required to ensure this property. These protocols practically create a total order of coherence requests across all blocks, even though only a per-block order is required. However, total order makes the implementation of some consistency models simpler.

Requiring total order has implications on the interconnection network as it must determine the order of requests. This is done by the serialization point which broadcast the messages to all controllers. The issuing controller then learns by snooping where its request has been ordered, maybe several cycles after issuing the request. In the case of a shared bus, an arbitration logic can be used to ensure that only a single request is issued at a time, which acts as a serialization point.

The key aspects of snooping protocols revolve around the requests. Response messages can travel on a separate network that does not need broadcast nor ordering capabilities. As they carry data, they are longer than requests and there are benefits to sending them on a simpler, lower-cost network. The time interval between the request appears on the bus and the response reception does not affect the serialization of the transaction.

### 4.3.1 An MSI snooping protocol

In this section, an unoptimized MSI snooping protocol is introduced. The interconnection network is supposed to be a shared bus, caches are supposed to be write-back and a cache-centric notation is used. Figures 4.6a and 4.6b respectively show the transitions between the stable states in the cache and the memory controller. For simplicity, they only show requests observed on the bus and other events like loads, stores and responses are omitted. Prefixes Own- and Other- denote whether the controller observing the request is the initiator.




(a) Cache controller (b) Memory controller

Figure 4.6 – MSI snooping protocol transitions. From [23]

**First assumption: atomic requests and atomic transactions**

Two atomicity properties are first assume to specify a first version of the protocol : atomic requests and atomic transactions. The first one states the the request is ordered in the same cycle that it is issued, preventing a block from changing its state in between. The second states that a subsequent request for a same block cannot appear on the bus until the first transaction has completed. This is not a problem for different blocks, as coherence involves operations on a single block. A similar protocol was implemented in the SGI Challenge.

The detailed specifications are available in Tables 4.5 and 4.6.Two transients states are added to the cache controller and one in the memory controller. This small amount of transient states is due to the atomicity constraints limiting the number of possible interleavings.

Shaded entries represent impossible transition and blank entries legal transitions that require no action. Entries labelled "(A)" are impossible due to the transaction atomicity. The event of observing the data for another processor's transaction is omitted as no response is needed from the cache controller.

In a MSI protocol, processor loads hit when the block is in state S or M, while processor writes hit only when the block is in state M. On misses, the cache controller initiate a transaction by sending a *GetS* or *GetM* request. Transients states $IS^D$, $IM^D$ and $SM^D$ indicate the controller is waiting for a data response. Because the requests have already been ordered, the blocks are logically in states S, M or M for those transient states, but loads and stores must wait for the data to arrive.

The atomicity properties simplify the protocol in two ways. The atomic requests ensures that a cache controller can issue a request without another request from another processor for the same block being ordered before, and it can directly waits for data response. The atomic transactions eliminates the need to handle requests from other processors while being in a transient state.

| S/E | Load | Store | Replace | Own-GetS | Own-GetM | Own-PutM | Data | Other-GetS | Other-GetM | Other-PutM |
|---|---|---|---|---|---|---|---|---|---|---|
| I | issue GetS /IS$^D$ | issue GetM /IM$^D$ | | | | | | | | |
| IS$^D$ | stall Load | stall Store | stall Evict | | | | copy data into cache, load hit, /S | (A) | (A) | (A) |
| IM$^D$ | stall Load | stall Store | stall Evict | | | | copy data into cache store hit, /M | (A) | (A) | (A) |
| S | load hit | issue GetM /SM$^D$ | -/I | | | | | | -/I | |
| SM$^D$ | load hit | stall Store | stall Evict | | | | copy data into cache store hit, /M | (A) | (A) | (A) |
| M | load hit | store hit | issue PutM send Data to memory /I | | | | | send Data to req and memory /S | send Data to req /I | |

Table 4.5 – MSI snooping (atomic req./trans.) cache controller. Adapted from [23]

| S/E | GetS | GetM | PutM | Data from Owner |
|---|---|---|---|---|
| IorS | send data block in Data message to requester/IorS | send data block in Data message to requester/M | | |
| IorS$^D$ | (A) | (A) | | update data block in memory/IorS |
| M | -/IorS | | -/IorS$^D$ | |

Table 4.6 – MSI snooping (atomic req./trans.) memory controller. Adapted from [23]

Data responses come from the memory controller or another cache only if that cache has the block in state M. Cache controllers thus ignore *GetS* messages for blocks in S state, but respond to *GetS* and *GetM* messages for blocks in M state and transition to state S or I, respectively.

The LLC/memory has two stable state, M and IorS, and one transient state IorS$^D$. In state IorS, the memory controller responds to both *GetS* and *GetM* request. In state M, it does not respond with data as it is the owner cache controller which is responsible for the block. However, when observing the *GetS* request in M state, meaning the owner cache controller transitions to state S, it must update data and thus transitions to state IorS$^D$.

At the moment of an eviction, the S-to-I state transition at the cache controller is performed silently, as all other coherence controllers remain unchanged. However, the M-to-I transition requires the cache controller to issue a *PutM* request and send data back to the memory controller, which has transitioned to IorS$^D$ after observing the request. Atomic requests prevents an intervening state downgrade issued by another processor before the *PutM* gets ordered on the bus. Atomic transactions prevents other requests for the block until the transaction is complete.

#### Second assumption: non-atomic requests and atomic transactions

The optimizations made by inserting a message buffer between the cache controller and the bus, allowing controllers not to wait until it is available gave rise to non-atomic requests. Separating the moments the requests are issued and ordered opens a window of vulnerabilities the coherence protocol must address. In this design, atomic transactions are still considered.

Tables 4.7 and 4.8 present the specifications details of an updated protocol. More transient states have been introduced due to the relaxed property. Indeed, cache controllers can now observe requests from other controllers between the issue and the observation of its coherence requests.

| S/E | load | store | replace | Own-GetS | Own-GetM | Own-PutM | Other-GetS | Other-GetM | Other-PutM | Own Data Response |
|---|---|---|---|---|---|---|---|---|---|---|
| I | issue GetS/IS$^{AD}$ | issue GetM/IM$^{AD}$ | | | | | - | - | - | |
| IS$^{AD}$ | stall | stall | stall | -/IS$^{D}$ | | | - | - | - | |
| IS$^{D}$ | stall | stall | stall | | | | (A) | (A) | | -/S |
| IM$^{AD}$ | stall | stall | stall | | -/IM$^{D}$ | | - | - | - | |
| IM$^{D}$ | stall | stall | stall | | | | (A) | (A) | | -/M |
| S | hit | issue GetM/SM$^{AD}$ | -/I | | | | - | -/I | - | |
| SM$^{AD}$ | hit | stall | stall | | -/SM$^{D}$ | | - | -/IM$^{AD}$ | - | |
| SM$^{D}$ | hit | stall | stall | | | | (A) | (A) | | -/M |
| M | hit | hit | issue PutM/MI$^{A}$ | | | | send data to requester and to memory/S | send data to requester/I | - | |
| MI$^{A}$ | hit | hit | stall | | | send data to memory/I | send data to requester and to memory/II$^{A}$ | send data to requester/II$^{A}$ | | |
| II$^{A}$ | stall | stall | stall | | | send NoData to memory/I | - | - | | |

Table 4.7 – MSI snooping (non-atomic requests) cache controller. Adapted from [23]

| S/E | GetS | GetM | PutM | Data from owner | NoData |
|---|---|---|---|---|---|
| IorS | send data to requester | send data to requester/M | -/IorS$^{D}$ | | |
| IorS$^{D}$ | (A) | (A) | | write data to LLC/IorS | -/IorS |
| M | -/IorS$^{D}$ | | -/M$^{D}$ | | |
| M$^{D}$ | (A) | (A) | | write data to LLC/IorS | -/M |

Table 4.8 – MSI snooping (non-atomic requests) memory controller. Adapted from [23]

For the I-to-S transitions, the cache controller issues a *GetS* and transitions to the state IS$^{AD}$, where A stands for appearance. In this transient state, the block state is effectively I, and the controller remains in this state until it observes its request. After that, the new transient state is IS$^{D}$ and the block state is logically S, but loads cannot be performed as the controller waits for data, which is in the next message, as transactions are still atomic. The I-to-M transition behaves in a similar way.

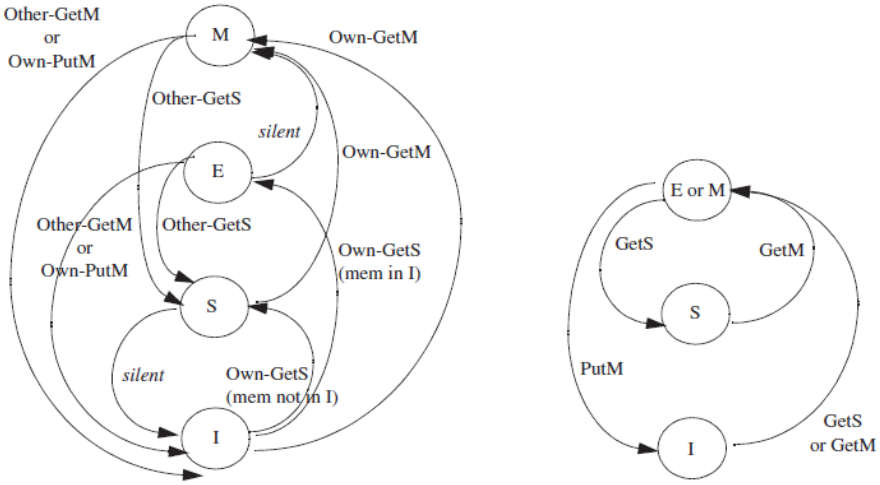
The S-to-M transition is affected by the window of vulnerability. When a cache controller issues a *GetM* request for a block in S, it transitions to SM$^{AD}$ and the block remains effectively in S. However if another controller *GetM* request is ordered first, the former must transition to IM$^{AD}$ to actually invalidate its copy.

The M-to-I transition is also affected. When a cache controller issues a *PutM* request for a block in state M, it transitions to MI$^{A}$. Until observing its own request, the block is effectively in state M. If an intervening *GetS* or *GetM* request from another controller is ordered before, the former controller must respond as the block is effectively in M state and transition to II$^{A}$ state to actually invalidate its copy. This particular transient state is introduced to prevent the memory controller from being stuck, as it will observe the *PutM* request. Therefore, the cache controller in state II$^{A}$ will send a special no data-message to the memory controller.

This case is managed at the memory controller by adding an M$^{D}$ state, indicating that memory controller should revert to state M when it receives a no-data message.

### 4.3.2 A MESI snooping protocol

Many computer programs contain sequence in which blocks may be read and then subsequently written. In an MSI protocol, a cache controller would first need to initiate a *GetS* transaction to obtain read permission, and then, a *GetM* transaction to obtain write permission. A MESI protocol offers a significant advantage in these situations. In no other cache has access to the block, the cache controller can obtain the block in state E when issuing a *GetS* request, and then silently upgrade the block state from E to M, reducing per two the amount of transactions needed in this scenario.




(a) Cache controller (b) Memory controller

Figure 4.7 – MESI snooping protocol transitions. From [23]

For this to be possible, the cache controller must determines that there are no other sharers. This can be done, for instance, by maintaining a "conservative" S state at the memory controller, meaning that there are zero-or-more caches in state S. The cache controller silently evicts blocks in state S, and the blocks in LLC stays in S even after the last sharer has evicted its copy. If a block in state M is written back, its state in the LLC block becomes I.

Figure 4.7a and 4.7b illustrate the transitions between the stables states in a MESI protocol. At the cache controller side, the *GetS* request make the block state transition to S or E depending on the its state in the LLC. Data is marked as exclusive in this case. A *PutM* message is used to evict blocks in state E, as both the state are merged at the LLC. The LLC has one more stable state to distinguish between blocks shared by zero or more sharers (S) and blocks that are not shared at all (I).

Tables 4.9 and 4.10 details the specifications of the MESI protocol. Atomic transactions are still considered. The differences compared to the MSI protocol are highlighted in bold. The cache controller is augmented with stable E and transient $EI^A$ states, while the memory controller is augmented with several more states.

### 4.3.3 A MOSI snooping protocol

In an MSI or MESI protocol, when a cache holds a block in state M or E and receives a *GetS* request, it must transitions to state S and responds to both the requester and the memory controller as the memory controller becomes the new responsible for that block as it is in S state. Adding the O state optimizes this situation by eliminating the extra data message sent to the memory controller, as well as unnecessary writes to the memory when a block has to be written by several processors subsequently.

| S/E | load | store | replace | Own-GetS | Own-GetM | Own-PutM | Other-GetS | Other-GetM | Other PutM | Own Data resp. | Own Data resp. (excl.) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| I | issue GetS/IS$^{AD}$ | issue GetM/IM$^{AD}$ | | | | | - | - | - | | |
| IS$^{AD}$ | stall | stall | stall | -/IS$^{D}$ | | | - | - | - | | |
| IS$^{D}$ | stall | stall | stall | | | | (A) | (A) | (A) | -/S | **-/E** |
| IM$^{AD}$ | stall | stall | stall | | -/IM$^{D}$ | | - | - | - | | |
| IM$^{D}$ | stall | stall | stall | | | | (A) | (A) | (A) | -/M | |
| S | hit | issue GetM/SM$^{AD}$ | -/I | | | | - | -/I | - | | |
| SM$^{AD}$ | hit | stall | stall | | -/SM$^{D}$ | | - | -/IM$^{AD}$ | - | | |
| SM$^{D}$ | hit | stall | stall | | | | (A) | (A) | (A) | -/M | |
| **E** | **hit** | **hit/M** | **issue PutM/EI$^{A}$** | | | | **send data to requester and to memory/S** | **send data to requester/I** | - | | |
| M | hit | hit | issue PutM/MI$^{A}$ | | | | send data to requester and to memory/S | send data to requester/I | - | | |
| MI$^{A}$ | hit | hit | stall | | | send data to memory/I | send data to requester and to memory/II$^{A}$ | send data to requester/II$^{A}$ | - | | |
| **EI$^{A}$** | **hit** | **stall** | **stall** | | | **send NoData-E to memory/I** | **send data to requester and to memory/II$^{A}$** | **send data to requester/II$^{A}$** | - | | |
| II$^{A}$ | stall | stall | stall | | | send NoData to memory/I | - | - | - | | |

Table 4.9 – MESI snooping cache controller. Adapted from [23]

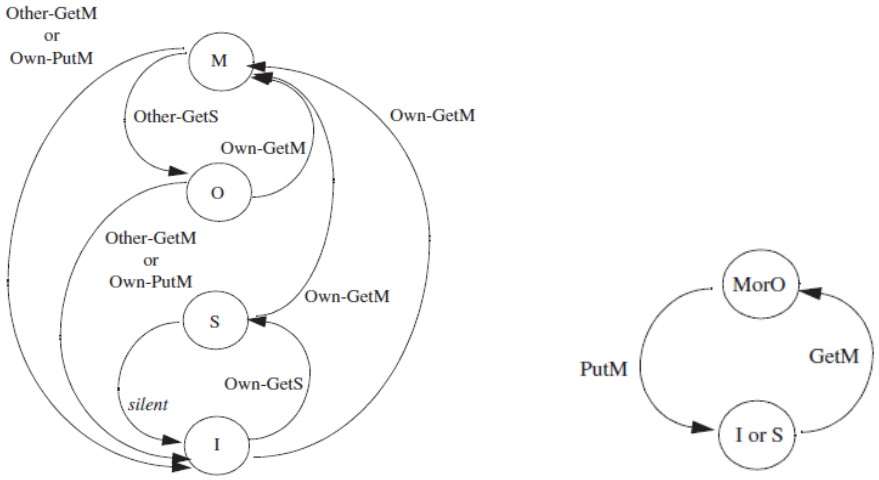
| S/E | GetS | GetM | PutM | Data | NoData | NoData-E |
|---|---|---|---|---|---|---|
| **I** | **send data to requester/EorM** | **send data to requester/EorM** | **-/I$^{D}$** | | | |
| **S** | **send data to requester** | **send data to requester/EorM** | **-/S$^{D}$** | | | |
| **EorM** | -/S$^{D}$ | - | **-/EorM$^{D}$** | | | |
| **I$^{D}$** | (A) | (A) | (A) | **write data to memory/I** | **-/I** | **-/I** |
| **S$^{D}$** | (A) | (A) | (A) | **write data to memory/S** | **-/S** | **-/S** |
| **EorM$^{D}$** | (A) | (A) | (A) | **write data to memory/I** | **-/EorM** | **-/I** |

Table 4.10 – MESI snooping memory controller. Adapted from [23]

Historically, the LLC was much slower than the private caches and might not be included in the processor chip itself. Therefore, the O state brought significant latency improvements by reducing transfers from and to the LLC.

Figure 4.8a and 4.8b illustrate te transitions between the stables states in a MOSI protocol. The main difference comes from the fact that, when a cache controller having a block in state M receives a *GetS* messages, it transitions to state O to keep ownership without having to update the LLC/memory.

Tables 4.11 and 4.12 details the specifications of the MOSI protocol. Atomic transactions are still considered. The differences compared to the MSI protocol are highlighted in bold. Two transient states are added to the cache controller, the OI$^{A}$ state and the OM$^{A}$ state, because these transitions are not silent. The memory controller has no additional state but the M state is renamed MorO as these states do not need to e distinguished at this level, and thus only a *PutM* transaction is needed for both the states.




(a) Cache controller (b) Memory controller

Figure 4.8 – MOSI snooping protocol transitions. From [23]

| S/E | load | store | replace | Own-GetS | Own-GetM | Own-PutM | Other-GetS | Other-GetM | Other PutM | Own Data resp. |
|---|---|---|---|---|---|---|---|---|---|---|
| I | issue GetS/$IS^{AD}$ | issue GetM/$IM^{AD}$ | | | | | - | - | - | |
| $IS^{AD}$ | stall | stall | stall | -/$IS^{D}$ | | | - | - | - | |
| $IS^{D}$ | stall | stall | stall | | | | (A) | (A) | (A) | -/S |
| $IM^{AD}$ | stall | tall | stall | | -/$IM^{D}$ | | - | - | - | |
| $IM^{D}$ | stall | stall | stall | | | | (A) | (A) | (A) | -/M |
| S | hit | issue GetM/$SM^{AD}$ | -/I | | | | - | -/I | - | |
| $SM^{AD}$ | hit | stall | stall | | -$SM^{D}$ | | - | -/$IM^{AD}$ | - | |
| $SM^{D}$ | hit | stall | stall | | | | (A) | (A) | (A) | -/M |
| **O** | **hit** | **issue GetM /$OM^{A}$** | **issue PutM/$OI^{A}$** | | | | **send data to requester** | **send data to requester/I** | - | |
| **$OM^{A}$** | **hit** | **stall** | **stall** | | **-/M** | | **send data to requester** | **send data to requester /$IM^{AD}$** | - | |
| M | hit | hit | issue PutM/$MI^{A}$ | | | | **send data to requester/O** | send data to requester/I | - | |
| $MI^{A}$ | hit | hit | stall | | | send data to memory/I | **send data to requester/$OI^{A}$** | send data to requester/$II^{A}$ | - | |
| **$OI^{A}$** | **hit** | **stall** | **stall** | | | **send data to memory/I** | **send data to requester** | **send data to requester/$II^{A}$** | - | |
| $II^{A}$ | stall | stall | stall | | | send NoData to memory/I | - | - | - | |

Table 4.11 – MOSI snooping cache controller. Adapted from [23]

| S/E | GetS | GetM | PutM | Data from Owner | NoData |
|---|---|---|---|---|---|
| IorS | send data to requester | send data to requester/MorO | -/$IorS^{D}$ | | |
| $IorS^{D}$ | (A) | (A) | | write data to memory /IorS | -/IorS |
| **MorO** | - | - | **-/$MorO^{D}$** | | |
| **$MorO^{D}$** | (A) | (A) | | **write data to memory/IorS** | **-/MorO** |

Table 4.12 – MOSI snooping memory controller. Adapted from [23]

### 4.3.4 A non-atomic MSI snooping protocol

Up until now, all bus transactions in the presented protocols consisted of indivisible request-response pair. Having such an atomic bus is similar to having an un-pipelined processor as

there is no way to overlap activities. A separate bus is assumed for the responses, as it was suggested in the beginning of the section.

In an atomic bus, illustrated at Figure 4.9, as a transaction occupies the bus until it completes, the bus throughput is limited by the sum of the transactions latencies, including any wait cycle. A pipelined bus, illustrated at Figure 4.10, is not atomic. It does not need to wait for a response before a subsequent request can be serialized, and responses are provided in the same order. This is still unoptimized in case of a long latency response for a prior request. To prevent this, a split transaction bus, illustrated in Figure 4.11 can be used. In this case, the data responses must carry the identity of the request or the requester.

To design a snooping protocol on a split transaction bus, two separate buses are needed for the requests and responses. FIFO queues connect the controllers to both buses in both directions, except for the memory controller which does not need to send requests.

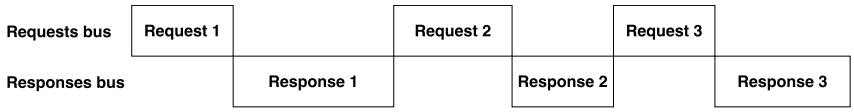



Figure 4.9 – An atomic bus

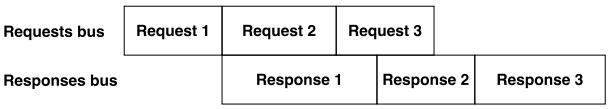



Figure 4.10 – A pipelined bus

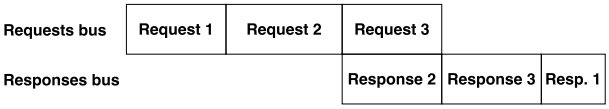



Figure 4.11 – A split-transaction bus

**First assumption: relaxing the transaction atomicity**

Tables 4.13 and 4.14 details the updated MSI protocol specifications. Several new transitions are now possible, highlighted in bold, compared to the atomic transactions MSI protocol. Indeed, the cache controller can now observe requests for block it holds in transient states.

A cache controller holding a block in $IS^D$ can observe a *GetS* message for this block, for instance. In this case, as the block is logically in state S, no action has to be taken by this controller. Its transaction is effectively completed, it just wait for the data to arrive.

Other transitions are more complex to handle. For instance, a cache controller holding a block in $IM^D$ can also observe a *GetS* for that block. In this case, as it is logically in M, the cache controller must respond. The simplest solution is to stall the request until the data is received. The request is thus processed when the block is in M state. For all other similar transitions, requests are also stalled until data arrives.

Making stalls sacrifies performance and raises two other issues. First, it raises potential for deadlocks. Indeed, the designer must ensures that awaited events will occur to prevent the controllers being block in a stall. In this protocol, responses are guaranteed to be received, as the stalls only occur on the request bus.

Second, it enables a cache controller to observe the response of its request before actually observing its own request. Indeed, as FIFO buffers are put between the buses and the controllers, it is possible for a cache controller to treat the request of another cache controller, say, a *GetM* request, which has still not observed its own request as it is currently in stall. It may therefore observe the data on the response bus before the request. To handle this case, states

$IS^{AD}$, $IM^{AD}$ and $SM^{AD}$ can respectively transitions to new states $IS^{A}$, $IM^{A}$ and $SM^{A}$, meaning that data is received but the request is not yet observed.

Relaxing the transaction atomicity property also make the handling of *PutM* requests different. If the cache controller issuing the request observes a *GetS* or *GetM* request before its *PutM* request, it will transition from $MI^{A}$ to $II^{A}$ to actually invalidate its copy, and won't be able to respond to the LLC/memory. To handle this situation, the LLC can holds the identity of the current owner of the block, such that it can ignores the invalid *PutM* request. The cache controller in $II^{A}$ state will therefore transition to I state. This is simpler than sending no-data, because there might be a large number of no-data messages in a non-atomic protocol, which could, moreover, arrive before the PutM request, as explained above.

| S/E | load | store | replace | Own-GetS | Own-GetM | Own-PutM | Other-GetS | Other-GetM | Other-PutM | Own Data Resp. for own req. |
|---|---|---|---|---|---|---|---|---|---|---|
| I | issue GetS/$IS^{AD}$ | issue GetM/$IM^{AD}$ | | | | | - | - | - | |
| $IS^{AD}$ | stall | stall | stall | -/$IS^{D}$ | | | - | - | - | **-/$IS^{A}$** |
| $IS^{D}$ | stall | stall | stall | | | | - | **stall** | | load hit/S |
| **$IS^{A}$** | **stall** | **stall** | **stall** | **load hit/S** | | | - | - | | |
| $IM^{AD}$ | stall | stall | stall | | -/$IM^{D}$ | | - | - | - | **-/$IM^{A}$** |
| $IM^{D}$ | stall | stall | stall | | | | **stall** | **stall** | | store hit/M |
| **$IM^{A}$** | **stall** | **stall** | **stall** | | **store hit/M** | | - | - | | |
| S | hit | issue GetM/$SM^{AD}$ | -/I | | | | - | -/I | | |
| $SM^{AD}$ | hit | stall | stall | | -/$SM^{D}$ | | - | -/$IM^{AD}$ | | **-/$SM^{A}$** |
| $SM^{D}$ | hit | stall | stall | | | | **stall** | **stall** | | store hit/M |
| **$SM^{A}$** | **hit** | **stall** | **stall** | | **store hit/M** | | - | **-/$IM^{A}$** | | |
| M | hit | hit | issue PutM/$MI^{A}$ | | | | send data to requester and to memory/S | send data to requester/I | | |
| $MI^{A}$ | hit | hit | stall | | | send data to requester/I | send data to requester and to memory/$II^{A}$ | send data to requester/$II^{A}$ | | |
| $II^{A}$ | stall | stall | stall | | | -/I | - | - | - | |

Table 4.13 – Split transaction bus MSI snooping cache controller. Adapted from [23]

| S/E | GetS | GetM | PutM from Owner | Data PutM from non-owner | Data |
|---|---|---|---|---|---|
| IorS | send data to requester, set | send data to requester, set Owner to requester/M | | - | |
| M | clear Owner/$IorS^{D}$ | set Owner to requester | clear Owner/$IorS^{D}$ | - | **write data to memory/$IorS^{A}$** |
| $IorS^{D}$ | **stall** | **stall** | **stall** | - | write data to memory/IorS |
| $IorS^{A}$ | **clear Owner/IorS** | - | **clear Owner/IorS** | - | |

Table 4.14 – Split transaction bus MSI memory controller. Adapted from [23]

**Second assumption: non-stalling protocol**

Stalling sacrifies performance as requests behind the one being stalled may be processed directly. Ideally, a coherence controller might process those requests. However, snooping protocols require total order, and controllers have to observe and process the request in the same order.

To deal with this problem, transient states can be added to reflect messages that the coherence controller has received but must complete after a later event. Tables 4.15 and 4.16 details the updated protocol specifications, with main modifications highlighted in bold. Many transient states are added.

For instance, if a cache holding a block in $IS^{D}$ receives a *GetM* request for that block, it

| S/E | load | store | replace | Own-GetS | Own-GetM | Own-PutM | Other-GetS | Other-GetM | Other-PutM | Own Data Resp. for own req. |
|---|---|---|---|---|---|---|---|---|---|---|
| I | issue GetS/IS$^{AD}$ | issue GetM/IM$^{AD}$ | | | | | - | - | - | |
| IS$^{AD}$ | stall | stall | stall | -/IS$^{D}$ | | | - | - | - | **-/IS$^{A}$** |
| IS$^{D}$ | stall | stall | stall | | | | - | **-/IS$^{D}$I** | | load hit/S |
| **IS$^{A}$** | **stall** | **stall** | **stall** | **load hit/S** | | | - | - | | |
| IM$^{AD}$ | stall | stall | stall | | -/IM$^{D}$ | | - | - | - | **-/IM$^{A}$** |
| IM$^{D}$ | stall | stall | stall | | | | **-/IM$^{D}$S** | **-/IM$^{D}$I** | | store hit/M |
| **IM$^{A}$** | **stall** | **stall** | **stall** | | **store hit/M** | | - | - | | |
| **IM$^{D}$I** | **stall** | **stall** | **stall** | | | | - | - | | **store hit, send data to GetM requester/I** |
| **IM$^{D}$S** | **stall** | **stall** | **stall** | | | | - | **-/IM$^{D}$SI** | | **store hit, send data to GetS requester and memory/S** |
| **IM$^{D}$SI** | **stall** | **stall** | **stall** | | | | - | - | | **store hit, send data to GetS requester and memory/I** |
| S | hit | issue GetM/SM$^{AD}$ | -/I | | | | - | -/I | | |
| SM$^{AD}$ | hit | stall | stall | | -/SM$^{D}$ | | - | -/IM$^{AD}$ | | **-/SM$^{A}$** |
| SM$^{D}$ | hit | stall | stall | | | | **-/SM$^{D}$S** | **-/SM$^{D}$I** | | store hit/M |
| **SM$^{A}$** | **hit** | **stall** | **stall** | | **store hit/M** | | - | **-/IM$^{A}$** | | |
| **SM$^{D}$I** | **hit** | **stall** | **stall** | | | | - | - | | **store hit, send data to GetM requester/I** |
| **SM$^{D}$S** | **hit** | **stall** | **stall** | | | | - | **-/SM$^{D}$SI** | | **store hit, send data to GetS requester and memory/S** |
| **SM$^{D}$SI** | **hit** | **stall** | **stall** | | | | - | - | | **store hit, send data to GetS requester and memory/I** |
| M | hit | hit | issue PutM/MI$^{A}$ | | | | send data to requester and to memory/S | send data to requester/I | | |
| MI$^{A}$ | hit | hit | stall | | | send data to requester/I | send data to requester and to memory/II$^{A}$ | send data to requester/II$^{A}$ | | |
| II$^{A}$ | stall | stall | stall | | | -/I | - | - | - | |

Table 4.15 – Non-stalling split transaction MSI snooping cache controller. Adapted from [23]

| S/E | GetS | GetM | PutM from Owner | Data PutM from non-owner | Data |
|---|---|---|---|---|---|
| IorS | send data to requester, set | send data to requester, set Owner to requester/M | | - | |
| M | clear Owner/IorS$^{D}$ | set Owner to requester | clear Owner/IorS$^{D}$ | - | **write data to memory/IorS$^{A}$** |
| IorS$^{D}$ | **stall** | **stall** | **stall** | - | write data to memory/IorS |
| IorS$^{A}$ | **clear Owner/IorS** | - | **clear Owner/IorS** | - | |

Table 4.16 – Non-stalling split transaction MSI memory controller. Adapted from [23]

transitions to IS$^{D}$I. That new nomenclature means that after receiving the data, it will transition to state I. If the block is in IM$^{D}$ and the controller observes a *GetS* request for the block, it will transition to I$^{D}$S, send the data to the requester it has remembered and transition to S upon its arrival.

Livelock problems may arise in such a non-stalling protocol. Indeed, when the cache controller with a block in IM$^{D}$S receives the data, it sends it to the requester and loose its requested write permission. If the processor reissues the *GetM* message, the same situation may happen again and again. To prevent this, the cache controllers in states IS$^{D}$I, IM$^{D}$I, IM$^{D}$S or IM$^{D}$SI are required to perform a load or store at the data arrival, *if and only if* it was the oldest load or store in program order when the request was first issued.

The stalls from the memory controller were not removed as this is not feasible. Indeed, the possible request for an awaited data would require the introduction of more transient states, requiring paying the same attention. There is no way to bound the number of transient states needed at the LLC/memory to a number smaller than the number of processors.

### 4.3.5 Interconnect for snooping protocols

All the protocols presented in this section assumed transferring requests and data on shared buses, because a total order broadcast was needed by the request. However, these properties can be obtained without using a physical bus. For instance, a tree with coherence controllers as leaves. If requests are unicast to the root and the root broadcast down the tree, then a total order is obtained. The root act as the serialization point. Such a topology is used in the Sun Starfire processor [9]. A logical total order can also be established on any network by broadcasting the timestamp at which the request must be treated with the message [20].

While a total order broadcast is required by the requests, responses can travel on a separate network and do not need to be broadcast or ordered. Such networks include crossbars, meshes, torus, butterflies,... Using a separate non-bus network for the responses may ease the implementation as high-speed shared buses are difficult to implement, increase the throughput as a bus provides only one response at a time and decrease the latency as an arbitration mechanism is needed for a bus.

## 4.4 Directory coherence protocols

Snooping protocols are simple and offers good performance because each transaction can be completed with two messages. However, totally ordered broadcast networks are costly, and hardly scale to a large number of processors. To address this lack of scalability, directory protocols were introduced. These ones avoid both the totally ordered broadcast network and having each coherence controller process every request.

Directory protocols use a directory which maintains the list of which caches hold each block and in what states. A cache controller issues its requests to the directory which determine what action to take according to the block state. It either directly responds or forward the request to a cache controller. Transactions typically involve two or three steps : a unicast request, a response or a forwarded request, and the response for the forwarded request. Sometimes a fourth step is needed because the responding cache controller indirect by the directory to notify it of some event. In contrast, snooping protocols do not centralize the information and require requests to be broadcast but, however, are able to complete a transaction in two steps.

Transactions are generally serialized at the directory. If two requests race the dictionary, the interconnection network chooses which request the directory will process first. In this case, the second request might either be processed directly after the first one or held in the directory awaiting the first request to complete. In some design, negative acknowledgements can be directly sent.

As there is no total order, all the controllers do not observe the same things and the requests must be individually serialized with respect to all the caches concerned by them. The requester is thus notified when the cache controller has serialized its request. For instance, on a *GetM* request, all caches controllers holding the block in state S have to acknowledge the request.

### 4.4.1 An MSI directory protocol

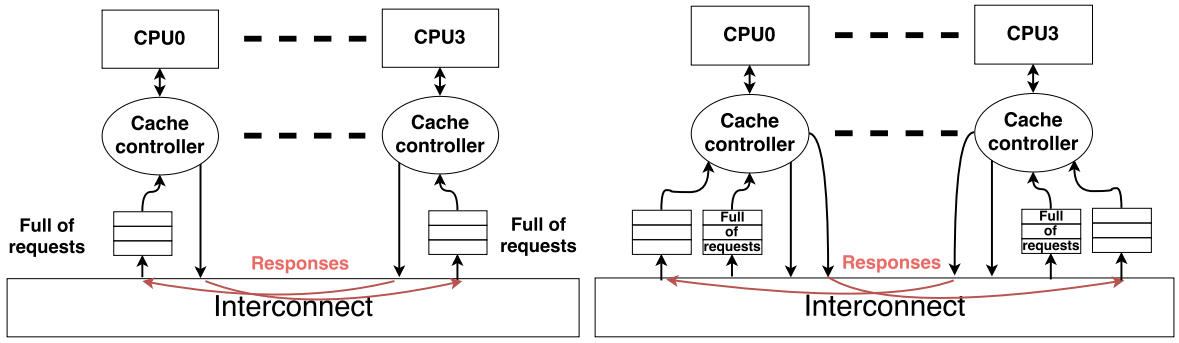
In this section, an unoptimized MSI directory protocol is introduced. The interconnection network is supposed to enforce point-to-point ordering, to reduce complexity. It can be a mesh, a torus, or any other topology. A directory is added next to the LLC and the LLC controller also is the directory controller. A block is owned by the directory controller unless a cache holds it in M state. For each block in memory, there is a corresponding directory entry, illustrated at Figure 4.12. It includes the stable coherence state, the identity of the owner and the identities of the sharers in a one-hot bit vector.

Figure 4.14 illustrates the transitions between stable states in the protocol, described in details below. The notations *Req* and *Dir* respectively identifies the requester and the directory.

Figure 4.12 – Directory entry for a block in a N-nodes system

Cache-centric notation is used for the directory states.

In this protocol, if messages use the same network links or buffer, the system may potentially deadlock. Indeed, as illustrated in Figure 4.13a, if CPU1 and CPU2 are responding to each other's request but their buffer are already full, the responses cannot be processed and both controller stall. A solution is to separate the networks for each class of messages, as illustrated in Figure 4.13b, to enable responses to be eventually consumed. As three classes of messages are used in this protocol, requests, forwarded messages and responses, it will use three separate networks. In practice, only the forwarded messages network require point-to-point ordering.




(a) Deadlock example (b) Separate networks

Figure 4.13 – Avoiding deadlocks in directory protocols

Tables 4.17 and 4.18 details the protocol specifications and introduce some transient states, because controllers must manage the states of the blocks that are in the midst of transactions. These are represented in the form $XY^{AD}$ where A denotes waiting for acknowledgement and D waiting for data.

### The I-to-S transition

The I-to-S transition can be divided in three cases when a cache controller issues a *GetS* request to the directory and changes its state to $IS^{D}$.

In the first case, no cache controller holds the block in state M and the directory is the owner. It respond with a Data message, changes the block state to S if not done already and adds the requester into the sharer list. The transaction completes when the data arrives at the cache controller, which changes the block state to S.

In the second case, the directory is not the owner, forwards the request to the owner and changes the block state to $S^{D}$. The owner responds to the *Fwd-GetS*, sends a Data message to the requester and changes the block state to S. It then sends the Data to the directory which must have an up-to-date copy. When the data arrives at both the cache and the directory, they both transition the block to state S and the transaction completes.

In the third case, the issuing cache controller receives an Invalidation message. This is indeed possible if another racing processor issues a *GetM* request that is serialized after the *GetS* request. In this case, the directory first sends Data message due to the former request and then an Invalidate message due to the later request. As Data and Invalidate messages are of different classes, they can arrive out of order at the coherence controller.

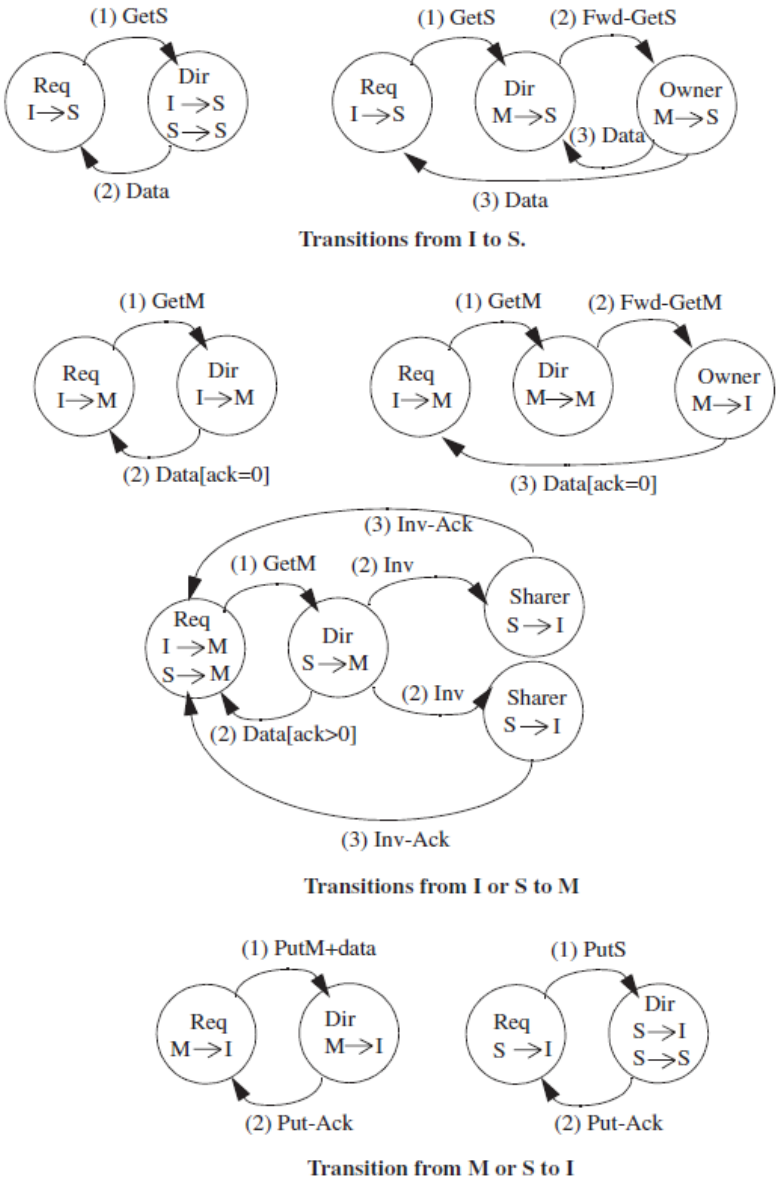



Figure 4.14 – MSI directory protocol transitions. From [23]

### The I- or S-to-M transition

When a cache controller issues a *GetM* request to the directory, it changes the block state to $IM^{AD}$. The directory responses can be divided in three scenarios.

First, if the directory is in state I, it sends a Data message with an *AckCount* of zero, transitions to state M and updates the block owner.

Second, if the directory is in state M, it has to forward the request *Fwd-GetM* to the owner, and update the block owner. The now-previous owner responds by sending a Data Message with an*AckCount* of zero.

Third, when the directory is in state S, it responds with a Data message and an *AckCount* equal to the number of sharers, and sends Invalidation messages to those sharers, which invalidate the copy and send *Inv-Acks*. When all the *Inv-Acks* are received by the requester, it transitions to state M. A special *Last-Inv-Ack* event is present in Table 4.17.

Races might happen when, for instance, a cache controller receives a *Fwd-GetS* message while in state $IM^{A}$, because the directory has already sent data to the controller, sent Invalidate

| S/E | load | store | replace | Fwd-GetS | Fwd-GetM | Inv | Put-Ack | Data from dir ack=0 | Data from dir ack>0 | Data from Owner | Inv-Ack | Last Inv-Ack |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I | send GetS to Dir/$IS^D$ | send GetM to Dir/$IM^{AD}$ | | | | | | | | | | |
| $IS^D$ | stall | stall | stall | | | stall | | -/S | | -/S | | |
| $IM^{AD}$ | stall | stall | stall | stall | stall | | | -/M | -$IM^A$ | -/M | ack- - | |
| $IM^A$ | stall | stall | stall | stall | stall | | | | | | ack- - | -/M |
| S | hit | send GetM to Dir/$SM^{AD}$ | send PutS to Dir/$SI^A$ | | | send Inv-Ack to Req/I | | | | | | |
| $SM^{AD}$ | hit | stall | stall | stall | stall | send Inv-Ack to Req/$IM^{AD}$ | | -/M | -/$SM^A$ | -/M | ack- - | |
| $SM^A$ | hit | stall | stall | stall | stall | | | | | | ack- - | -/M |
| M | hit | hit | send PutM+data to Dir/$MI^A$ | send data to Req and Dir/S | send data to Req/I | | | | | | | |
| $MI^A$ | stall | stall | stall | send data to Req and Dir/$SI^A$ | send data to Req/$II^A$ | | -/I | | | | | |
| $SI^A$ | stall | stall | stall | | | send Inv-Ack to Req/$II^A$ | -/I | | | | | |
| $II^A$ | stall | stall | stall | | | | -/I | | | | | |

Table 4.17 – MSI directory protocol cache controller. Adapted from [23]

| S/E | GetS | GetM | PutS-NotLast | PutS-Last | PutM+data from Owner | PutM+data from NonOwner | Data |
|---|---|---|---|---|---|---|---|
| I | send data to Req add Req to Sharers/S | send data to Req set Owner to Req/M | send Put-Ack to Req | send Put-Ack to Req | | send Put-Ack to Req | |
| S | send data to Req, add Req to Sharers | send data to Req, send Inv to Sharers, clear Sharers, set Owner to Req/M | remove Req from Sharers, send Put-Ack to Req | remove Req from Sharers send Put-Ack to Req/I | | remove Req from Sharers, send Put-Ack to Req | |
| M | Send Fwd-GetS to Owner, add Req and Owner to Sjarers, clear Owner/$S^D$ | Send Fwd-GetM to Owner, set Owner to Req | send Put-Ack to Req | end Put-Ack to Req | copy data to memory, clear Owner, send Put-Ack to Req/I | send Put-Ack to Req | |
| $S^D$ | stall | stall | remove Req from Sharers send Put-Ack to Req | remove Req from Sharers send Put-Ack to Req | | remove Req from Sharers, send Put-Ack to Req | copy data to memory/S |

Table 4.18 – MSI directory protocol cache controller. Adapted from [23]

to sharers and changed the block state to M. Thus, it simply forwards the *GetS* message and the *Fwd-GetS* message, travelling on a different network, can arrive before the *Inv-Acks*.

### The M-to-I transition

When a cache controller evicts a block in M, it sends a *PutM* request to the directory and changes its state to $MI^A$. The directory update the LLC, responds with a *Put-Ack* message, and transitions to I. Meanwhile, the block remains effectively in M at the cache, and if *Fwd-GetS* or *Fwd-GetM* messages are received, the controller responds and changes the block state to $SI^A$ or $II^A$, respectively, denoting that the block states is effectively S or I, but the controller must wait the *PutAck* to complete the transaction.

### The S-to-I transition

Compared to previously seen protocols, a shared block is no more silently evicted: the information is no more broadcast. When a cache controller evicts a block in S, it sends a *PutS* request to the directory and changes its state to $SI^A$. The directory update the LLC, responds with a *Put-Ack* message, and transitions to I when all the *PutS* messages from all the sharers are received. Meanwhile, the block remains effectively in S at the cache, and if Invalidation request is received, the controller changes the block state to $II^A$, denoting that the block state is effectively I, but the controller must wait the *PutAck* to complete the transaction.

### 4.4.2 A MESI directory protocol

The MESI protocol enables a processor issuing a *GetS* request to obtain a block in state E if no other cache holds the block. It allows the cache controller to silently upgrade the block state from E to M without issuing a coherence request.

As the cache holding the block in state E (or M) is the owner of the block, the directory will forward requests to it. Moreover, as for the M state in the previous MSI protocol, the eviction of a block in E must notify the directory with a *PutE* message to inform it that it is the new block owner.

Figure 4.15 illustrates the transitions between the stables states in this MESI protocol. Differences compared to the previous MSI protocol are highlighted in bold. The I-to-E and E-to-I transitions have been added. The *PutE* message does not need to carry data as the data is clean.

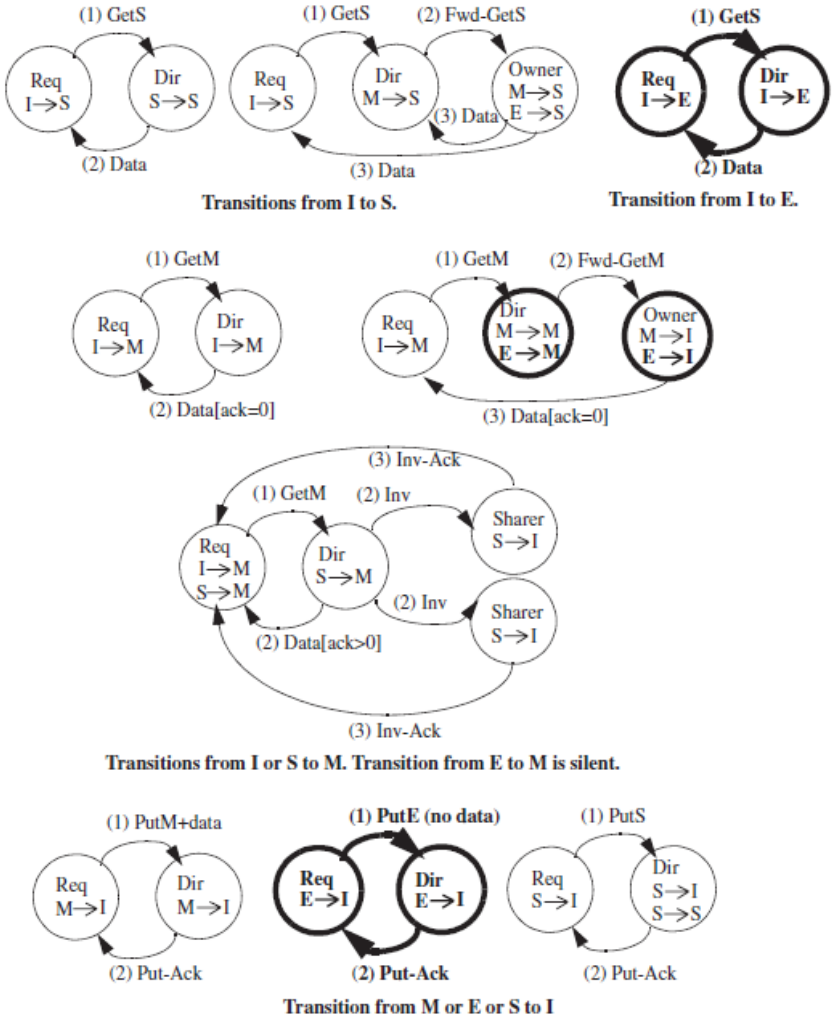



Figure 4.15 – MESI directory protocol transitions. From [23]

Tables 4.19 and 4.20 list the MESI protocol specification details, with difference compared to the previous MSI protocol highlighted in bold. The cache controller is augmented with the E state and a transient state $EI^A$ to handle the eviction. The $IS^D$ state can now transition to E if the data is marked as exclusive.

The directory controller is augmented with an E state is added and it must distinguish data coming from the cache owner in case of a *PutE* message.

| S/E | load | store | replace | Fwd-GetS | Fwd-GetM | Inv | Put-Ack | Excl. data from dir | Data from dir ack=0 | Data from dir ack>0 | Data from Owner | Inv-Ack | Last Inv-Ack |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I | send GetS to Dir/$IS^D$ | send GetM to Dir/$IM^{AD}$ | | | | | | | | | | | |
| $IS^D$ | stall | stall | stall | | | stall | | **-/E** | -/S | | -/S | | |
| $IM^{AD}$ | stall | stall | stall | stall | stall | | | | -/M | -$IM^A$ | -/M | ack- - | |
| $IM^A$ | stall | stall | stall | stall | stall | | | | | | | ack- - | -/M |
| S | hit | send GetM to Dir/$SM^{AD}$ | send PutS to Dir/$SI^A$ | | | send Inv-Ack to Req/I | | | | | | | |
| $SM^{AD}$ | hit | stall | stall | stall | stall | send Inv-Ack to Req/$IM^{AD}$ | | | -/M | -/$SM^A$ | -/M | ack- - | |
| $SM^A$ | hit | stall | stall | stall | stall | | | | | | | ack- - | -/M |
| M | hit | hit | send PutM+data to Dir/$MI^A$ | send data to Req and Dir/S | send data to Req/I | | | | | | | | |
| **E** | **hit** | **hit/M** | **send PutE (no data) to Dir/$EI^A$** | **send data to Req and Dir/S** | **send data Req/I** | | | | | | | | |
| $MI^A$ | stall | stall | stall | send data to Req and Dir/$SI^A$ | send data to Req/$II^A$ | | -/I | | | | | | |
| **$EI^A$** | **stall** | **stall** | **stall** | **send data to Req and Dir/$SI^A$** | **send data to Req/$II^A$** | | **-/I** | | | | | | |
| $SI^A$ | stall | stall | stall | | | send Inv-Ack to Req/$II^A$ | -/I | | | | | | |
| $II^A$ | stall | stall | stall | | | | -/I | | | | | | |

Table 4.19 – MESI directory protocol cache controller. Adapted from [23]

| S/E | GetS | GetM | PutS-NotLast | PutS-Last | PutM+data from Owner | PutM from NonOwner | PutE(no data) Owner | PutE from NonOwner | Data |
|---|---|---|---|---|---|---|---|---|---|
| I | send Exclusive data to Req, set Owner to Req/E | sed data to Req, set Owner to Req/M | send Put-Ack to Req | send Put-Ack to Req | | send Put-Ack to Req | | **send Put-Ack to Req** | |
| S | send data to Req add Req to Sharers | send data to Req, send Inv to Sharrs, set Owner to Req/M | remove Req from Sharers,send Put-Ack to Req | remove Req from Sharers, send Put-Ack | | remove Req from Sharers, send Put-Ack to Req | | **remove Req from Sharers, send Put-Ack to Req** | |
| **E** | **forward GetS to Owner, make Owner sharer, add Req to Sharers,clear Owner/$S^D$** | **forward GetM to Owner, set Owner to Req/M** | **send Put-Ack to Req** | **send Put-Ack to Req** | **copy data to mem, send Put-Ack to Req, clear Owner/I** | **send Put-Ack to Req** | **send Put-Ack to Req,clear Owner/I** | **send Put-Ack to Req** | |
| M | forward GetS to Owner, make Owner sharer, add Req to Sharers,clear Owner/$S^D$ | forward GetM to Owner, set Owner to Req | send Put-Ack to Req | send Put-Ack to Req | copy data to mem, send Put-Ack to Req, clear Owner/I | send Put-Ack to Req | | **send Put-Ack to Req** | |
| $S^D$ | stall | stall | remove Req from Sharers,send Put-Ack to Req | remove Req from Sharers, send Put-Ack to Req | | remove Req from Sharers send Put-Ack to Req | | **remove Req from Sharers send Put-Ack to Req** | copy data to LLC/mem/S |

Table 4.20 – MESI directory protocol cache controller. Adapted from [23]

### 4.4.3 A MOSI directory protocol

The MOSI protocol enables a cache controller with a block in state M to transition to state O when it observes a *Fwd-GetS* without copying the value back to the LLC. Therefore, more coherence requests are directly responded by the cache controllers, and, in the case of directory protocols, more three-step transactions instead of four-step ones.

Figure 4.16 illustrates the transitions between the stables states in this MOSI protocol. Differences compared to the previous MSI protocol are highlighted in bold. When a cache controller with a block in S or I issues a *GetM* request to the directory in which block is in state O, the directory forwards the request to the owner with an *AckCount*. It should also send an Invalidate message to all the other sharers. The owner responds to the requester with a Data message and the *AckCount* and the requester waits for all the *Inv-Acks* before transitioning the block state to M. A *PutO* transaction is also introduced and is similar to the *PutM* transaction.

Tables 4.21 and 4.22 list the MOSI protocol specification details, with difference compared

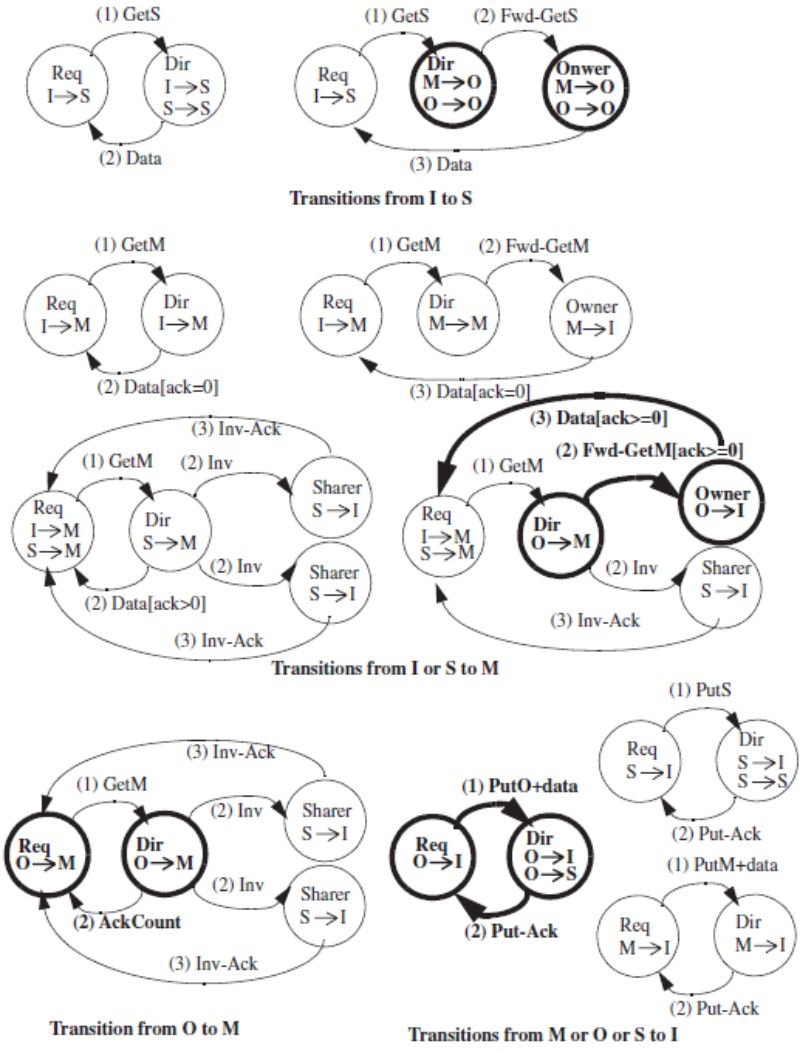



Figure 4.16 – MOSI directory protocol transitions. From [23]

to the previous MSI protocol highlighted in bold.

The cache controller is augmented with the O state and transient states $OM^{AC}$, $OM^{A}$ and $OI^{A}$. The $OM^{AC}$ means the controller is waiting for both *Inv-Acks* and an *AckCount* from the directory. The data is state O is already valid.

When a cache controller has a block in state $OM^{AC}$ or in $SM^{AD}$ and receives a *Fwd-GetM* or an Invalidation message from another processor for the same data, this means that the later processor's *GetM* has been ordered first by the directory, and the former cache controller must change the block state to $IM^{AD}$ to effectively invalidates its copy. Indeed, the directory considered the cache had its copy invalidated as the first *GetM* it processed induced that change in the directory.

### 4.4.4 Directory state and organization

In practical systems, directories generally do not maintain the complete states for each block and the full list of sharers, because this is not scalable for a system with a large number of caches and the main motivation for designing directory protocols was the lack of scalability of snooping protocols. Two important techniques can be used to reduce the amount of information the directory has to maintain for each block : coarse directories and limited pointers.

In coarse directories, a coarse list, a superset of the actual set of sharers, is maintained.

| S/E | load | store | replace | Fwd-GetS | Fwd-GetM | Inv | Put-Ack | Data from dir ack=0 | Data from dir ack>0 | Data from Owner | Ack Count from dir | Inv-Ack | Last Inv-Ack |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I | send GetS to Dir/IS$^D$ | send GetM to Dir/IM$^{AD}$ | | | | | | | | | | | |
| IS$^D$ | stall | stall | stall | | | stall | | -/S | | -/S | | | |
| IM$^{AD}$ | stall | stall | stall | stall | stall | | | -/M | -IM$^A$ | -/M | | ack- - | |
| IM$^A$ | stall | stall | stall | stall | stall | | | | | | | ack- - | -/M |
| S | hit | send GetM to Dir/SM$^{AD}$ | send PutS to Dir/SI$^A$ | | | send Inv-Ack to Req/I | | | | | | | |
| SM$^{AD}$ | hit | stall | stall | stall | stall | send Inv-Ack to Req/IM$^{AD}$ | | -/M | -/SM$^A$ | -/M | | ack- - | |
| SM$^A$ | hit | stall | stall | stall | stall | | | | | | | ack- - | -/M |
| M | hit | hit | send PutM+data to Dir/MI$^A$ | **send data to Req/O** | send data to Req/I | | | | | | | | |
| MI$^A$ | stall | stall | stall | **send data to Req /OI$^A$** | send data to Req/II$^A$ | | -/I | | | | | | |
| **O** | **hit** | **send GetM to Dir/OM$^{AC}$** | **send PutO+data to Dir/OI$^A$** | **send data to Req** | **send data to Req/I** | | | | | | | | |
| **OM$^{AC}$** | **hit** | **stall** | **stall** | **send data to Req** | **send data to Req/IM$^{AD}$** | | | | | | **-/OM$^A$** | **ack- -** | |
| **OM$^A$** | **hit** | **stall** | **stall** | **send data to Req** | **stall** | | | | | | | **ack- -** | **-/M** |
| **OI$^A$** | **stall** | **stall** | **stall** | **send data to Req** | **send data to Req/II$^A$** | | **-/I** | | | | | | |
| SI$^A$ | stall | stall | stall | | | send Inv-Ack to Req/II$^A$ | -/I | | | | | | |
| II$^A$ | stall | stall | stall | | | | -/I | | | | | | |

Table 4.21 – MOSI directory protocol cache controller. Adapted from [23]

| S/E | GetS | GetM from Owner | GetM from NonOwner | PutS-NotLast | PutS-Last | PutM+data from Owner | PutM +data from NonOwner | PutO+data Owner | PutO +data from NonOwner |
|---|---|---|---|---|---|---|---|---|---|
| I | send Data to Req, add Req to Sharers/S | | send Data to Req, set Owner to Req/M | send Put-Ack to Req | send Put-Ack to Req | | send Put-Ack to Req | | send Put-Ack to Req |
| S | send Data to Req, add Req to Sharers | | send Data to Req, send Inv to Sharers,set Owner to Req, clear Sharers/M | remove Req from Sharers, send Put-Ack to Req | remove Req from Sharers, send Put-Ack to Req/I | | remove Req from Sharers, send Put-Ack to Req | | remove Req from Sharers, send Put-Ack to Req |
| O | forward GetS to Owner, add Req to Sharers | send Ack-Count to Req, send Inv to Sharers, clear Sharers/M | forward GetM to Owner, send Inv to Sharers, set Owner to Req, clear Sharers, send AckCount to Req/M | remove Req from Sharers, send Put-Ack to Req | remove Req from Sharers, send Put-Ack to Req | remove Req from Sharers, copy data to mem, send Put-Ack to Req, clear Owner/S | remove Req from Sharers, send Put-Ack to Req | copy data to memory, send Put-Ack to Req, clear Owner/S | remove Req from Sharers, send Put-Ack to Req |
| M | forward GetS to Owner, add Req to Sharers/O | | forward GetM to Owner, set Owner to Req | send Put-Ack to Req | send Put-Ack to Req | copy data to mem, send Put-Ack to Req, clear Owner/I | send Put-Ack to Req | | send Put-Ack to Req |

Table 4.22 – MOSI directory protocol cache controller. Adapted from [23]

Therefore, each bit in this list corresponds to K caches, and if one of these K caches holds the block in S state, then the bit for those cache is set in the list. However, an Invalidation is thus sent to all the K caches in this case, increasing the network bandwidth and the cache controller bandwidth to process these extra messages.

In limited pointer directories, the sharer list is replaced by a pointer list to caches sharing the block. Indeed, studies have shown that many blocks have zero or one sharer. Therefore, the list is limited to $i < N$ entries and its length is $i \log_2 N$. However, when the system wish to add an $i + 1^{th}$ sharer, several solutions can be used:

- Broadcast : In this case, the directory set the block state such that a subsequent *GetM* request requires the invalidation of all the caches, sending some unnecessary messages.

A common use case is when $i = 0$. Two bits can then be used to save the MSI state of the block and a special state when it is the single sharer, to avoid a broadcast when a cache wants to upgrade a block from state S to M.

- No broadcast : In this case, the directory asks one of the sharers to invalidate its copy. This can lead to significant performance problem for widely-shared blocks, due to the time spent invalidating the sharers.
- Software : In this case, the system makes a software interrupt. It offers great flexibility, but as using software means significant performance costs and implementation complexities, this has very rare commercial applications.

Those two design options, alongside with the complete entry introduced in the beginning of the section, are illustrated in Figure 4.17.

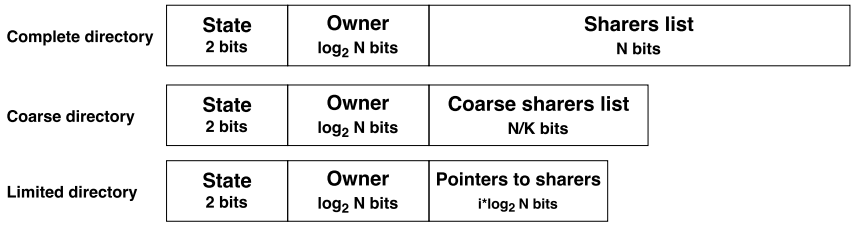



Figure 4.17 – Possible directory representations

Directories also have to logically contain an entry for every single block in the memory. Old implementations, such as the SGI Origin, maintained the complete directory state for each block in DRAM chips. Nowadays, directories are implemented on-chip as caches, to improve latency and power performance. However, these directory caches only contain a subset of the complete directory entries. Different design options can be taken to handle directory cache misses:

- Directory cache-backed by DRAM: In this design, the complete directory is stored in DRAM and the directory cache is used to reduce the access latency. This design suffers from important drawbacks: it requires a significant amount of DRAM to hold the directory, and it is possible to hit in the LLC and miss in the directory cache, leading to a DRAM access even if the data is available locally. Moreover, directory cache replacements must write back to DRAM.
- Inclusive directory cache: In this design, the directory cache holds the entries for a superset of all the blocks cached on the processor chip. There is therefore no misses, as a miss in such a directory cache means that the block is in state I. Two designs are possible.
  - Embedded in an inclusive LLC: As explained previously, an inclusive LLC contains an entry for all blocks cached in a lower-level cache. In this design, the coherence state is embedded in the LLC by adding some bits to each block. In case of a miss, the block is not cached on-chip, and its state is I in all lower-level caches. It however has some drawbacks. When a block is replaced in the LLC, the lower-level caches must invalidate their copies and a *Recall* request is needed for that. Moreover, inclusion requires to maintain a redundant copy of all the blocks in intermediate-level caches, whose collective capacity represents a significant franction of the LLC capacity.
  - Standalone inclusive directory: In this design, the directory cache is a standalone structure associated with the directory controller. For this cache to be inclusive, it must contain entries for the blocks cached in the lower-level caches, as a block in

LLC but not in a lower-level cache is in state I. It therefore consists of duplicates copies of the lower-level caches tags, which has a storage cost, but is more flexible. Inclusive directory caches also requires a high degree of associativity. For C processors associated to a K-way associative cache, the directory cache must be $CK$-way associative, to hold all the cache tags.

The directory cache associativity can be limited to $A < CK$, and not permitted to allow the worst-case situation ($CK$-way associativity). When a new block has to be added in a full directory cache, the directory controller evicts one of the block in the corresponding set from all the caches by issuing a *Recall* request to the sharers and waiting for *Inv-Acks*. However, this can lead to poor performance if the cache is too small and *Recalls* are too frequent.

- Null directory cache: In this design, no directory cache is used. The directory state helps prune the set of controllers to which forward a request. If this pruning is incomplete, the protocol still works, but less efficiently. A protocol using a limited pointer directory with broadcast can therefore be implemented without a directory, and thus without a directory cache. This design is popular for small-case systems, because it incurs no storage cost. The directory controller is still needed for the LLC and as an ordering point.

### 4.4.5 Distributed directories

Using a unique directory can lead to a performance bottleneck in large systems, and can be distributed in N nodes, as illustrated in Figure 4.18. Blocks are still assigned to one unique directory, and each node holds 1/N of the directory state.

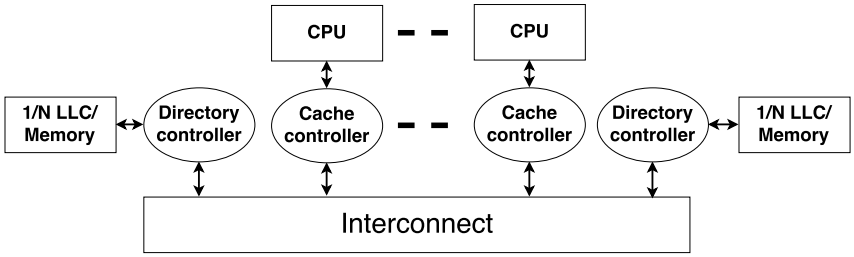



Figure 4.18 – System model with distributed directories

The directory responsible for a block B can be computed with simple arithmetic, B modulo N. Using multiple directories increases the bandwidth of coherence transactions and has no impact on the coherence protocol. In multicore processors, the nodes are the cores and the LLC and the directory cache are banked.

## 4.5 System model variations

The system model introduced at the beginning of this chapter omits several features available in most commercial systems.

### 4.5.1 Instruction caches

Modern systems all have at least one level of instruction cache. These caches are also concerned by cache coherence. For instance, by loading programs, the operating system can modify a cached memory blocked, and, just-in-time compilers generate code.

As the processor modifies the code via the data cache and these instructions are read-only, instruction cache controllers only need stable states I or S. They only take actions whe observing a *GetM* request by invalidating the block.

Because some instructions may remain buffered in the processor pipeline, the software modifying code needs to know if it affects the fetched instructions stream. This can be, for instance, implemented with a structure tracking the instruction in the pipelined and flushing it when it detects a change, or with a specific instruction to invalidate an instruction cache entry.

### 4.5.2 Translation lookaside buffers (TLBs)

The transaltion lookaside buffer is a cache for the page table that allows translations from virtual to physical addresses and must be kept coherent. However, it generally does not take part of the same hardware than the discussed coherence protocols as it is not addressed the same way data caches are.

The classical approach is a software-managed coherence scheme called TLB shootdown. When a processor invalidates a translation entry, it sends an interrupt to all processors. These processors traps to a software handler which either invalidates the entry from their TLBs or flushes all the entries from their TLBs, depending on the platform. Each processor then flushes the pipeline to ensure no instruction uses that stale translation. Eventually, the processors sends an acknowledgement interrupt to the initiating processor which can modify the entry. Some implementations replace software interrupt by specific TLB invalidating instructions.

### 4.5.3 Write-through caches

During all this chapter, write-back caches were assumed in all the described protocols. Write-through caches have several advantages. Their coherence can be carried out with a simple VI protocol. In this case, stores write through the LLC and invalidate the copies in other caches. Moreover, a cache eviction does not requires any action as the LLC always has the up-to-date data. Eventually, when a cache observes another write for a block, it only needs to invalid its copy.

However, they also have significant disadvantages such as the greater bandwidth and power requirement to write data to the next level, limiting their usage to L1 caches in modern systems. Moreover, in a multi-threaded processors system with a shared L1 cache and TSO consistency, a store to block A by one thread must prevent the other thread from accessing the new value until the copies in other caches have been invalidated.

### 4.5.4 Coherent direct memory access (DMA)

DMA are devices that reads and writes memory under software control, for making copies for instance, typically at the page granularity. Such a device should be able to read an up-to-date version of a block, even if one cache holds the block in M or O state, and to invalidate all stale copies.

The most simple solution to provide coherent DMA is by adding a coherent cache to the device and make it participate in the coherence protocol as other caches controllers. However, this is quite undesirable for two reasons. First, DMA controllers do not share the same locality patterns than a processor. For instance, they typically do not reuse the same blocks. Thus, they have no need for a cache larger than a single block. Second, they have no need in the data received by the issue of a *GetM* request as they will overwrite it completely. Many protocols optimize this case by providing a special *GetM-NoData* request to only obtain permission to write, while others provide a special *PutNewData*, updating memory and invalidating other copies.

Another solution is to operate via software in a similar way as TLB Shoot-down, flushing the caches before a DMA operation. However, this is inefficient, as the operating system must conservatively force the cache to flush a page even if none of its block are cached, and thus this is only implemented on some embedded systems.

### 4.5.5 Multi-level caches and multiple multi-core processors

In multi-level caches systems, the most straightforward way to ensure coherence is to treat each cache independently and make L1, L2 and LLC process all coherence requests. However, by making the L2 cache inclusive, processing messages to L1 caches can be avoided when the requested block is not in L2 cache, as, in this case, it is not in L1 caches neither. Requests can thus only be sent to L1 caches if the L2 cache holds the requested block. However, inclusion still wastes space for redundant storage of blocks and requires invalidating L1 copies of a block when it is evicted from L2 cache.

In a multi-chip systems, illustrated in Figure 4.19, the cores LLC can be viewed as another level of the memory hierarchy, and used either as a memory-side cache that holds blocks requested from the main memory by all the chips, or as a core-side cache that holds blocks requested by the cores on the chip. In the later case, the coherence protocol should also operate among the LLCs and memories.

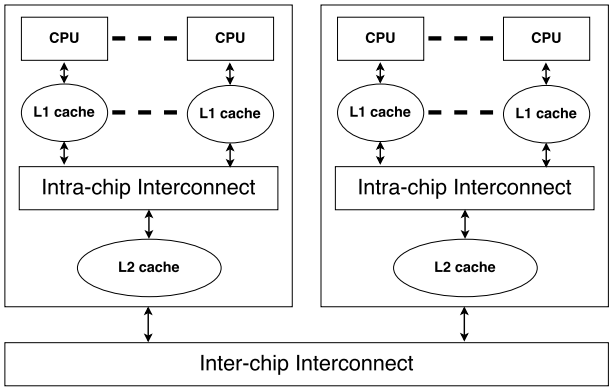



Figure 4.19 – Multiple multi-core chips system

Multi-level cache hierarchies introduces the possible need for hierarchical coherence protocols. For instance, in a multi-chip system, there can be an intra-chip coherence protocol and an inter-chip protocols, which do not interact together. The choice of the protocol at one level is independent from the protocol at another level and, for instance, the intra-chip protocol can be a snooping protocol while the inter-chip protocol can be a directory protocol. Indeed, a scalable protocol may not be useful in a chip with a limited number of cores.

Up until now, symmetrical processors and processor cores were assumed. However, there is a growing interest for heterogeneous computing that enables the use of multiple kind of cores relying on a different ISA on the same system. Many challenge arises from the fact that this kind of systems are not symmetric anymore. Indeed, all these actors no more have the same communication interface, or data format. In May 2016, AMD, ARM, Huawei, IBM, Mellanox, Qualcomm and Xilinx jointly announced that they would collaborate to build a cache-coherent fabric to interconnect their CPUs, accelerators and networks, via the CCIX consortium [10]. The solution that this new consortium will hopefully bring in the next years should make the gap more abstract between symmetric multiprocessing and heterogeneous computing.

# Chapter 5

# Workload-driven evaluation

The field of computer architecture becomes more and more quantitative, and design features are adopted after detailed trade-off evaluations. Based on the measurements provided by benchmarks, small programs stressing some of the machine features, assessments of emerging technology and new requirements for applications, designers propose new alternatives, which are evaluated through simulation. The benchmark workloads are thus run on a simulator integrating the new design in order to characterize its performance impact on the specific features stressed by the workload.

In this chapter, the gem5 simulator is introduced as well as the SPLASH2 and PARSEC benchmark suites. Then, the main chosen system parameters not depending on the cache coherence protocol are discussed, and the methodology employed to get the benchmark suites run on gem5 is detailed.

## 5.1 The gem5 architectural simulator

The gem5 simulator [7] is a free and open-source software providing a highly configurable simulation framework which can evaluate a large range of systems with different CPU models, system execution modes, and memory models, forming the result of a merge between the M5 [8] and GEMS [21] simulators. Most commercial ISA are supported (ARM, ALPHA, MIPS, Power, SPARC and x86) and is able to boot a Unix-like operating system such as BSD or Linux.

### 5.1.1 CPU, system and memory models

According to the need of accuracy for a specific simulation, the gem5 simulator offers three main degrees of flexibility, whose trade offs are shown in Figure 5.1.

- CPU model: Five CPUs models are provided. The *AtomicSimple* is a minimal model completing memory accesses immediately, *TimingSimple* simulates the timing of memory references, *InOrder* is a pipelined, in-order CPU, *O3* is a pipelined out-of-order CPU supporting hardware threads. An additional CPU model, KVM, provides, on supported architectures, a way to avoid binary translation.
- System model: Two modes are provided. The *System-call Emulation* (SE) mode avoids the need to model devices or the operating system, but does not integrate a thread scheduler, requiring mapping threads to cores statically for multi-threaded applications, and a *Full-System* (FS) mode models a complete system with OS and devices.
- Memory system: Two models are provided. The *Classic* model provides a fast memory system while the *Ruby* model provides an infrastructure capable of simulating varieties of cache coherent systems thanks to a domain-specific language.

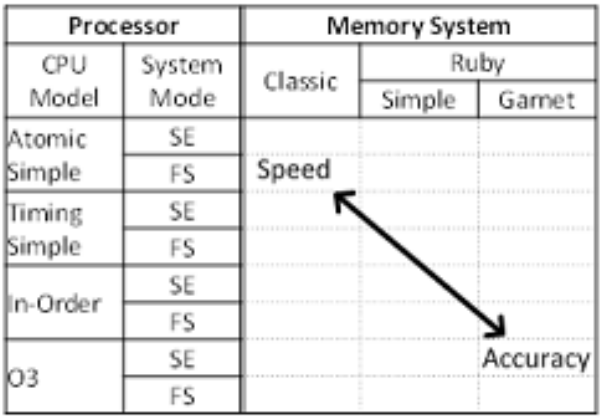



Figure 5.1 – Speed/accuracy trade offs in gem5. From [7]

The gem5 simulator is written in both C++, for describing the components behaviour, and in Python, for describing the systems. This provides the flexibility of easily simulating systems of any kinds as Python is an very high-level interpreted object-oriented language, while not loosing performance in the simulations as C++ is compiled to native machine code.

The need for simulating various cache coherent protocols prevents using the classic fast memory model of gem5,a although it remains possible to replace the embedded MOESI snooping protocol of this model as the code is open-source. However, the Ruby model provides a domain-specific language to design cache coherence protocols in a very convenient way : SLICC.

### 5.1.2 Ruby memory model

The Ruby memory model, whose high-level view is illustrated in Figure 5.2 allows building modular systems from independent components including inclusive and exclusive cache hierarchies, coherence protocols implementation, interconnection networks, DMA and memory controllers and sequencers that initiate memory requests and handle responses.

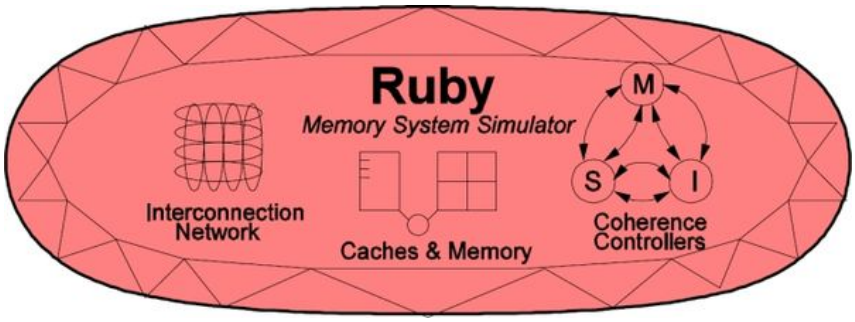



Figure 5.2 – Ruby memory model. From [13]

In more details, the Ruby system models the following components:

- Sequencers: It is responsible for feeding the whole memory system, including caches and off-chip memory with memory requests from the processor or DMA, and returning responses to the processor or DMA. Each processor core or DMA has its own sequencer
- Cache memories: Caches are set-associative with parametrizable size, associativity and replacement policies. Replacement policies, LRU or pseudo-LRU, and coherence protocols, are implemented separately from the caches.

- Memory controller: It simulates and services the requests that miss on the caches and models DRAM ban contention and refresh, as well as close-page policy of DRAM buffer.

Several interconnection network topologies are available in the Ruby memory model : crossbar, mesh, torus and point-to-point, with mesh and torus networks requiring the same number of directories as CPUs. Their definition is done in Python and are located in the `configs/topologies` folder. The default interconnection network is a crossbar. The advanced Garnet model also models detailed routing and control flow.

### 5.1.3 SLICC specification language

SLICC is a domain specific language and stands for *Specification Language for Implementing Cache Coherence*. As seen in Chapter 4, coherence protocols are implemented by means of finite state machines. SLICC is used for specifying their behaviour, as well as some constraints to be as close as possible to the hardware, such as the number of transitions that can tale place in a single cycle. The FSM takes its input from the input ports of the interconnection network and queues the output at the output ports of the network, as illustrated in Figure 5.3.

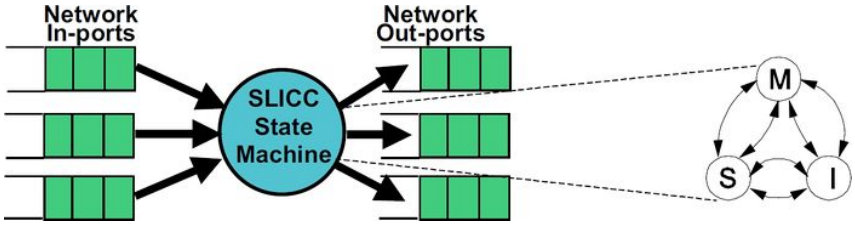



Figure 5.3 – Ruby memory model. From [13]

The SLICC compiler takes files that specify the controllers involved in the protocol and generates C++ code. Coherence protocols are defined using a main `.slicc` files including `.sm` files describing each of the state machines. In gem5, these files are typically found in the `/src/mem/protocol` folder and Ruby systems are typically found in the `configs/ruby` folder.

In the following sections, an MI directory protocol implementation will be used to illustrate the way coherence protocols are implemented using SLICC.

#### Finite state machines

In SLICC, each FSM is described with the `machine` datatype, and has several types of member, according to the type of controller : cache, or directory. An example of definition is given in Listing 5.1. This code defines a sequencer, the cache memory elements and the latency in number of cycles.

```
machine(L1Cache, "MI Example L1 Cache")
  : Sequencer * sequencer,
    CacheMemory * cacheMemory,
    Cycles cache_response_latency := 12;
    Cycles issue_latency := 2;
    {
      // Enumerations, and behaviour description
    }
```

Listing 5.1 – Machine definition in SLICC

In addition, message buffers have to be specified, as Listing 5.2. This code defines two input and two output network message buffers with the `network` attribute of different type of

data, with the `vnet_type` attribute, and on different virtual networks, according to the kind of transfer, with the `virtual_network` attribute. The last message buffer is the buffer from Ruby: the processor core, or the main memory, according to the defined controller.

```
MessageBuffer * requestFromCache, network="To", virtual_network="2",
  vnet_type="request";
MessageBuffer * responseFromCache, network="To", virtual_network="4",
  vnet_type="response";
MessageBuffer * forwardToCache, network="From", virtual_network="3",
  vnet_type="forward";
MessageBuffer * responseToCache, network="From", virtual_network="4",
  vnet_type="response";
MessageBuffer * mandatoryQueue;
```

Listing 5.2 – Message buffers definition

Into its description, the machine must include a descriptions of all its stable and transient states, with the `state_declaration` keyword, as illustrated in Listing 5.3 with an example of a MI protocol. The `AccessPermission` attribute defines the operations possible with data in these states. Four possibilities exist : `Invalid`, `Read_Only`, `Read_Write` and `Busy`, indicating a stalled request. A description of the state, for automatic documentation, can also be specified.

```
state_declaration(State, desc="Cache states") {
  I, AccessPermission:Invalid, desc="Not Present/Invalid";
  II, AccessPermission:Busy, desc="Not Present/Invalid, issued PUT";
  M, AccessPermission:Read_Write, desc="Modified";
  MI, AccessPermission:Busy, desc="Modified, issued PUT";
  MII, AccessPermission:Busy, desc="Modified, issued PUTX, received nack";
  IS, AccessPermission:Busy, desc="Issued request for LOAD/IFETCH";
  IM, AccessPermission:Busy, desc="Issued request for STORE/ATOMIC";
}
```

Listing 5.3 – Stable and transient states declaration

The different events involved in the coherence protocols requiring action to be taken by the controller can then be specified with an enumeration, as showed in Listing 5.4. As for states, description attributes can be used for documentation.

```
enumeration(Event, desc="Cache events") {
  Load,           desc="Load request from processor";
  Ifetch,         desc="Ifetch request from processor";
  Store,          desc="Store request from processor";
  Data,           desc="Data from network";
  Fwd_GETX,       desc="Forward from network";
  Inv,            desc="Invalidate request from dir";
  Replacement,    desc="Replace a block";
  Writeback_Ack,  desc="Ack from the directory for a writeback";
  Writeback_Nack, desc="Nack from the directory for a writeback";
}
```

Listing 5.4 – Events declaration

#### Data structures and messages

Structures can be defined in SLICC to represent entities in the system such as directory entries, as shown in Listing 5.5. Getter and setter for attributes are automatically generated. Many types are already declared in the `/src/mem/protocol/RubySlicc_*.sm` files. To inherit from a native SLICC/C++ interface class, the `interface` attribute can be used. Structures can also integrate C++ functions

```
structure(Entry, desc="...", interface="AbstractEntry") {
  State DirectoryState, desc="Directory state";
  NetDest Sharers,      desc="Sharers for this block";
  NetDest Owner,        desc="Owner of this block";
}
```

Listing 5.5 – Example of structure for a directory entry

Coherence messages that controllers use to communicate are also defined using structures in SLICC, as shown in Listing 5.6. The *addr* field stores a gem5-context address, the `Type` field is the coherence request or response, which is also defined with an enumeration, the `Requestor` field stores the ID of the requesting controller, the `Destination` field holds a bit array for the routing operated by the switches in the network, the `DataBlk` field contains the data and the *MessageSize* contains the message size for the network. The `Dirty` field specifies if the data block is dirty or not. The response message can also contain an integer `AckCount` attribute.

```
structure(RequestMsg, desc="...", interface="Message") {
  Addr addr,                  desc="Physical address for this request";
  CoherenceRequestType Type,  desc="Type of request (GetS, GetX, PutX, etc)";
  MachineID Requestor,        desc="Node who initiated the request";
  NetDest Destination,        desc="Multicast destination mask";
  DataBlock DataBlk,          desc="data for the cache line";
  MessageSizeType MessageSize,desc="size category of the message";
}
structure(ResponseMsg, desc="...", interface="Message") {
  Addr addr,                  desc="Physical address for this request";
  CoherenceResponseType Type, desc="Type of response (Ack, Data, etc)";
  MachineID Sender,           desc="Node who sent the data";
  NetDest Destination,        desc="Node to whom the data is sent";
  DataBlock DataBlk,          desc="data for the cache line";
  bool Dirty,                 desc="Is the data dirty (different than memory)?";
  MessageSizeType MessageSize,desc="size category of the message";
}
```

Listing 5.6 – Coherence messages definition

Miss status handling registers, also called transaction buffer entries and introduced in Section 4.2.1, are also implemented with structures, as shown in Listing 5.7. As a reminder, those MSHR keep track of blocks in transient states. It stores the gem-5 address for the block, its state, the block data and whether it is dirty, as well as, in this example, the number of pending acks. These structures are hold in a TBE Table, a lookup structure indexed by block addresses.

#### Input and output ports

Declaring an input port defines the behaviour of the controller when a message is received on that port. It is done with the `in_port` keyword and is showed in Listing 5.8. The first argument is the port identifier, the second is the type of message the input port receives and

```
structure(TBE, desc="TBE entry") {
  Address Address,              desc="Physical address for this TBE";
  State TBEState,               desc="Transient state";
  DataBlock DataBlk,            desc="Buffer for the data block";
  bool Dirty, default="false",  desc="data is dirty";
  int pendingAcks, default="0", desc="pending ack number";
  MachineID requestor,          desc="requestor";
}
```

Listing 5.7 – MSHR entry definition

the third corresponds to the message buffer to read from. The `RubyRequest` type are the requests that comes from Ruby : the processor, or the main memory. They can be user-defined as described above.

```
in_port(mandatoryQueue_in, RubyRequest, mandatoryQueue, desc="...") {
  if (mandatoryQueue_in.isReady(clockEdge())) {
    peek(mandatoryQueue_in, RubyRequest, block_on="LineAddress") {
      Entry cache_entry := getCacheEntry(in_msg.LineAddress);
      if (is_invalid(cache_entry) &&
          cacheMemory.cacheAvail(in_msg.LineAddress) == false ) {
        // make room for the block
        trigger(Event:Replacement, cacheMemory.cacheProbe(in_msg.LineAddress),
                getCacheEntry(cacheMemory.cacheProbe(in_msg.LineAddress)),
                TBEs[cacheMemory.cacheProbe(in_msg.LineAddress)]);
      }
      else {
        trigger(mandatory_request_type_to_event(in_msg.Type), in_msg.LineAddress,
                cache_entry, TBEs[in_msg.LineAddress]);
      }
    }
  }
}

out_port(responseNetwork_out, ResponseMsg, responseFromCache);
```

Listing 5.8 – Declaration of an input and output port in SLICC

The `peek` keyword is used to peek a message from the buffer and implicitly declares an `in_msg` which points at the head of the queue that can be used to access the message fields. Once the request is processed, an transition can occur. This is done by triggering an event using the `trigger` keyword. The number of arguments can vary and are generally the event to be triggered, the block address concerned by the message, the cache entry and the MSHR for that cache entry. The `trigger` also increments the counter that checks the number of transitions performed per cycle.

The output port is more trivial and is defined as shown at the end of Listing 5.8. The first argument is the port identifier, the second the message type to be transmitted and the third the message buffer on which put the message.

#### Actions

Actions are called when transitions, which are described further, are provoked by triggers and defines the behaviour of the controller. They are defined with the keyword `action`, as shown in the Listing 5.9. The first argument is the action identifier, the second an abbreviation for the documentation and the last a description, also used for documentation.

```
action(d_sendData, "d", desc="Send data to requestor") {
  peek(memQueue_in, MemoryMsg) {
    enqueue(responseNetwork_out, ResponseMsg, 1) {
      out_msg.addr := address;
      out_msg.Type := CoherenceResponseType:DATA;
      out_msg.Sender := machineID;
      out_msg.Destination.add(in_msg.OriginalRequestorMachId);
      out_msg.DataBlk := in_msg.DataBlk;
      out_msg.MessageSize := MessageSizeType:Response_Data;
    }
  }
}
```

Listing 5.9 – Definition of an action sending back data to the owner in SLICC

As the were specified via the `trigger` keyword, SLICC is aware of the block address for which the action is taken, its cache entry and its MSHR. The code in Listing 5.9 also illustrate the way a message is sent on its output port, via the `enqueue` keyword. As for the `peek` keyword, it implicitly declares a `out_msg` variable. The arguments are the output port name, the message type and the latency, in cycles, after which the message can be dequeued.

#### Transitions

A transition function maps the cross product of a set of current states and a set of events to a set of new states, and defines the SLICC actions to be taken when the transition is triggered. This is shown in Listing 5.10.

```
transition(IM, Data, M) {
  u_writeDataToCache;
  sx_store_hit;
  w_deallocateTBE;
  n_popResponseQueue;
}
```

Listing 5.10 – Transition definition in SLICC

In this example, the transition occurs when a `Data` event is triggered and the states changes from IM to M. In this case, the following actions must be taken : writing data to cache, setting the block as the most recently used, deallocating the MSHR and removing the response from the response queue. A set of states can be expressed as `{State1, State2}`

#### Queue stalling

There is several way to perform stalls in SLICC. The first way is to stall the entire message queue with the internally defined `z_stall` action in a transition definition. This method was the one considered in Section 4.3.4 with snooping protocols. Stalling an entire queue is inefficient, as subsequent requests may be processable, and adding transient states was suggested. However, removing all the stalls from a protocol is not always possible.

The second way is to recycle the stalled message on the input port. This is quite unrealistic and consists in putting the stalled message and the end of the queue and continuing processing messages. This is done with by calling the `recycle` function of the message buffer, for instance `mandatoryQueue_in.recycle()`, in an action definition.

The third way is to stall and wait. This is implemented with the internally defined `stall_and_wait(mandatoryQueue_in, address)` function, where the first argument is the

message buffer and the second the block address, in an action definition. The message is then moved in a table associated to the input port and must be explicitly removed from this table when the appropriate event occurs.

## 5.2 The SPLASH2 and PARSEC3 workloads

Assessing the performance of an architecture cannot be done by launching any kind of workloads. The complete assessed system has to be solicited by the run programs. For instance, mono-threaded or embarrassingly parallel workloads are not, useful for characterising cache coherence protocols, as they will not, or almost not, reproduce a situation in which the multiple cores may be working on a set of shared pieces of data. In this section, the SPLASH2 and PARSEC3 benchmarking collections are introduced.

### 5.2.1 SPLASH2 benchmark collection

The SPLASH2 suite [25] is a set of parallel applications released in 1995 designed to facilitate the study of centralized and distributed shared-memory multiprocessors, and help system designer choosing parameters and pruning the experimental space in a meaningful way, with realistic applications.

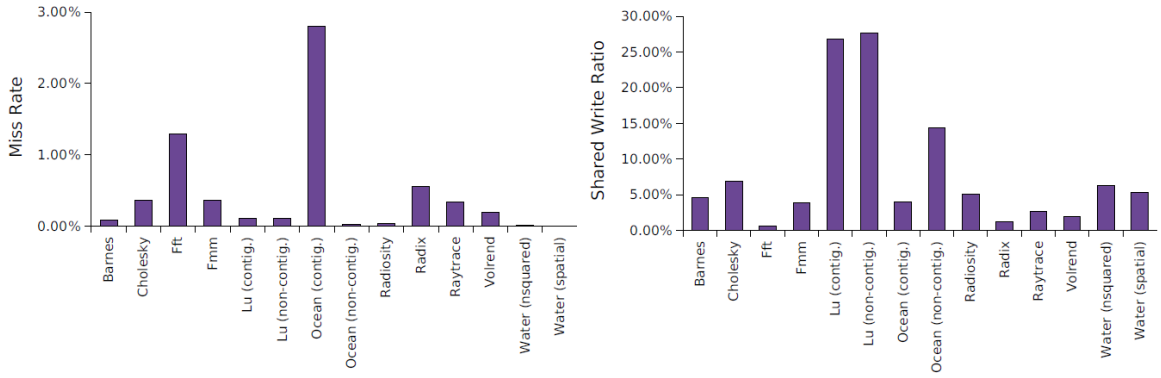
The workloads are a mixture of complete applications and computation kernels : simulation of the interaction of a system of bodies, in 3-D with Barnes-Hutt method (Barnes) or in 2-D with Fast Multipole Method (FMM), sparse (Cholesky) or dense (LU) matrix factoring to the product of a lower triangular matrix and its transpose, Fast-Fourier-Transform (FFT), simulation of large-scale ocean movements (Ocean), computation of the equilibrium distribution of light in a scene (Radiosity), Radix sort (Radix), scene rendering with ray tracing (Raytrace), volume rendering using ray-casting (Volrend), and evaluation of forces and potentials occurring in water molecules with an $O(n^2)$ (Water-Nsquared) or $O(n)$ (Water-Spatial) algorithm.

The basic characterisation of SPLASH2 programs is available in Table 5.1. More detailed information can be found in [25].

| Code | Problem Size | Total Instr (M) | Total FLOPS (M) | Total Reads (M) | Total Writes (M) | Barriers | Locks | Pauses |
|---|---|---|---|---|---|---|---|---|
| Barnes | 16K particles | 2002.79 | 239.24 | 406.85 | 313.29 | 8 | 34648 | 0 |
| Cholesky | tk15.O | 539.17 | 172.00 | 111.86 | 28.03 | 3 | 54054 | 4203 |
| FFT | 64K points | 34.79 | 6.36 | 4.07 | 2.88 | 6 | 0 | 0 |
| FMM | 16K particles | 1250.02 | 423.88 | 226.23 | 38.58 | 20 | 28088 | 0 |
| LU | $512 \times 512$ matrix, $16 \times 16$ blocks | 494.05 | 92.20 | 104.00 | 48.00 | 66 | 0 | 0 |
| Ocean | $258 \times 258$ ocean | 379.93 | 101.54 | 81.89 | 18.93 | 364 | 2592 | 0 |
| Radiosity | room, -ae 5000.0 -en 0.050 -bf 0.10 | 2832.47 | --- | 499.72 | 284.61 | 10 | 231190 | 0 |
| Radix | 1M integers, radix 1024 | 50.99 | --- | 12.06 | 7.03 | 10 | 0 | 124 |
| Raytrace | car | 829.32 | --- | 208.90 | 79.95 | 0 | 94471 | 0 |
| Volrend | head | 754.77 | --- | 152.19 | 59.57 | 15 | 28934 | 0 |
| Water-Nsq | 512 molecules | 460.52 | 98.15 | 81.27 | 35.25 | 10 | 17728 | 0 |
| Water-Sp | 512 molecules | 435.42 | 91.50 | 72.31 | 32.73 | 10 | 353 | 0 |

Table 5.1 – SPLASH2 basic workload characteristics. From [25]

A characterization of typical cache misses for an 4MB cache with 4-way and 64 bytes lines is available in Figure 5.4a and the ratio of shared writes, indicating the amount of communication between threads, is available in Figure 5.4b. [5]




(a) Miss rate

(b) Shared writes

Figure 5.4 – Miss rate and shared writes ratio for SPLASH2 benchmarks. From [5]

### 5.2.2 PARSEC3 benchmark collection

The PARSEC suite [6] is a set of multi-threaded applications first released in 2008 focusing on emerging workloads and designed to be representative of the next-generation shared-memory chip multiprocessors programs.

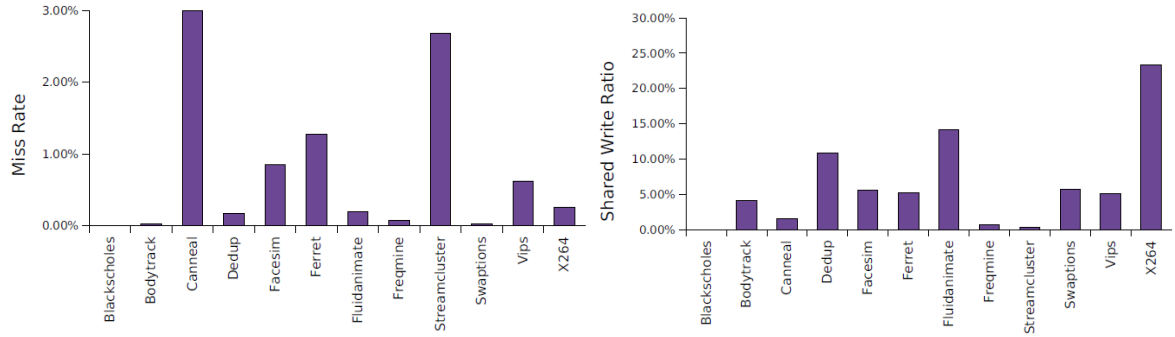
The workloads also are a mixture of complete applications and kernels : price calculation of stock options using partial differential equations (Blackscholes), human body tracking (Bodytrack), chip routing cost minimization using simulated annealing (Canneal), deduplication compression scheme (Dedup), realistic face modelling (Facesim), content-based similarity search (Ferret), incompressible fluid animation (Fluidanimate), Frequent Itemset Mining (Freqmine), online stream clustering solving (Streamcluster), swaptions pricing (Swaptions), the VASARI Image Processing System (Vips), and H264/AVC video encoding (X264).

The basic characterisation of SPLASH2 programs are available in Table 5.2. More detailed information can be found in [6].

| Program | Instructions (Billions) | | | | Synchronization Primitives | | |
|---|---|---|---|---|---|---|---|
| | Total | FLOPS | Reads | Writes | Locks | Barriers | Conditions |
| blackscholes | 2.67 | 1.14 | 0.68 | 0.19 | 0 | 8 | 0 |
| bodytrack | 14.03 | 4.22 | 3.63 | 0.95 | 114,621 | 619 | 2,042 |
| canneal | 7.33 | 0.48 | 1.94 | 0.89 | 34 | 0 | 0 |
| dedup | 37.1 | 0 | 11.71 | 3.13 | 158,979 | 0 | 1,619 |
| facesim | 29.90 | 9.10 | 10.05 | 4.29 | 14,541 | 0 | 3,137 |
| ferret | 23.97 | 4.51 | 7.49 | 1.18 | 345,778 | 0 | 1255 |
| fluidanimate | 14.06 | 2.49 | 4.80 | 1.15 | 17,771,909 | 0 | 0 |
| freqmine | 33.45 | 0.00 | 11.31 | 5.24 | 990,025 | 0 | 0 |
| streamcluster | 22.12 | 11.6 | 9.42 | 0.06 | 191 | 129,600 | 127 |
| swaptions | 14.11 | 2.62 | 5.08 | 1.16 | 23 | 0 | 0 |
| vips | 31.21 | 4.79 | 6.71 | 1.63 | 33,586 | 0 | 6,361 |
| x264 | 32.43 | 8.76 | 9.01 | 3.11 | 16,767 | 0 | 1,056 |

Table 5.2 – PARSEC3 basic workload characteristics. From [6]

A characterization of typical cache misses for an 4MB cache with 4-way and 64 bytes lines is available in Figure 5.4a and the ratio of shared writes, indicating the amount of communication between threads, is available in Figure 5.4b. [5]




(a) Miss rate

(b) Shared writes

Figure 5.5 – Miss rate and shared writes ratio for PARSEC3 benchmarks. From [5]

Compared to SPLASH2, PARSEC has been design for chip multiprocessors. Indeed, their apparition have changed the cost model employed for optimization [5]. SPLASH2 has been designed for distributed shared memory. This means that in those programs, it is better to have more cache misses than communication between the processors, as each processor possesses a part of the memory locally, as it was shown is Section 3.2. This prevents message passing through a slow off-chip interconnect. On the contrary, in chip multiprocessors, as the interconnection is on-chip, communication between processors can be done in a very fast way, while cache misses require fetching data in the off-chip main memory. This means that working on contiguous data is more advantageous in chip multiprocessors than in distributed shared-memory systems.

Since version 3, PARSEC includes the whole SPLASH2 benchmark suite, which can then be used as a complementary set of programs. More information about the differences between the two benchmarks can be found in [5].

## 5.3 Choosing the simulated ISA

The gem5 simulator offers a wide wide range of possibilities. While cache coherence is ISA-independent, this must be fixed to enable the design of the simulation framework, that is, the code that is run by the simulated processor.

Several ISAs are available in gem5 and are listed, with the main compatible features, in Table 5.3. The gem5 description in Section 5.1 states that only the Ruby memory model supports the specification of various cache coherence protocols, eliminating the architectures not compatible with this model.

| | ALPHA | ARM | x86 | SPARC | PowerPC | MIPS |
|---|---|---|---|---|---|---|
| Ruby | ✔ | ✗ | ✔ | ✗ | ✗ | ✗ |
| Full-system | ✔ | ✔ | ✔ | ✔ | ✗ | ✗ |
| KVM | ✗ | ✔ | ✔ | ✗ | ✗ | ✗ |

Table 5.3 – Available gem5 features for implemented ISAs

This is unfortunately the case of the ARM architecture [12]. ARM is indeed an active member of the gem5 project [18], providing graphical tools for analysing gem5 simulations, making ARM one of the most complete and supported ISA on gem5. In a first time, the simulation framework have been developed for this ISA, as the compatibility matrix status provided on gem5 wiki [16] did not mention such an incompatibility. However, the Ruby memory model

only supports a contiguous memory range while ARM processors rather have a fragmented memory map with aliasing and do not interface correctly in practice.

Full-system simulation is also needed because gem5 system-call emulation mode does not support *pthread* natively, due to the lack of an internal thread scheduler. A mock API called *m5threads* is available and statically maps each thread to a processor. However, in addition to preventing from running *Intel Thread Building Blocks* based benchmarks, modifying PARSEC to correctly link with this library would require a huge amount of time as system-call emulation moreover requires static build for the workloads, which is another challenge as the PARSEC suite has many shared-library dependencies and itself consists in several shared modules.

Two architectures support the Ruby model in full-system mode : ALPHA and x86. ALPHA is a now-obsolete architecture developed by DEC and the historical ISA on the GEMS simulator. x86 is the well-known architecture from Intel, available in most desktop personal computers and many servers. It is still topical and gem5 supports its AMD64 extension, as well as Linux KVM capabilities. Therefore, the Intel x86 architecture was chosen as the reference ISA, mainly because it supports the KVM features, making possible to fast-forward very efficiently non-interesting parts of the simulation, such as the system boot process and the decompression of the benchmarks inputs.

The *KvmCPU* model can indeed be used as a fast-forwarding CPU model. This feature enables the CPU model to be switched during the simulation or at the restoration of a checkpoint. As the Ruby and classic memory models are abstracted inside gem5, the checkpoints can be used from one model to another.

## 5.4 Making the simulation framework

For gem5 to run the benchmarks in a full-system simulation, a Linux kernel and an image disk containing the benchmarks is needed. This section describes the choice and methodology to produce those materials.

### 5.4.1 Cross-compilation versus virtual machine

The gem5 simulator is extremely slow due to the level of details it provides, and still does not implement all the features available in real CPUs, and it is thus not thinkable to get a Linux operating system and applications ready from the simulator. There are two main methods for producing an image disk that will run on such a simulator, which may simulates another architecture than the host computer : cross-compilation, or virtualization.

Cross-compilation consists in using a compiler that produces machine code for another architecture. This method would require a Linux root file-system (produced with buildroot, for instance), and copying the built files into this file-system. This is challenging, because of the many shared-library dependencies the PARSEC benchmark suite has.

Virtualization helps solving this difficulty by allowing the compilation procedure to be done on the target system. By using an emulator like QEMU [4], which targets many architectures, a standard Linux distribution targeting that same architecture can easily be installed on a disk image as the user was on a real machine, using its package collection to meet the library requirements and avoid compatibility issues. This solution is used for generating the PARSEC disk image.

An alternative when the host and target architectures are identical is to use `chroot`. This tool enables changing the root folder of a Linux process. It provides the performance of running on the host OS, but would also require a root-filesystem. To remain versatile, and because in a first time, this methodology was applied to the ARM ISA, this work was carried out with QEMU.

### 5.4.2 Configuring and compiling a gem5-friendly Linux kernel

KVM has been introduced in the Linux kernel since version 2.6.20 in 2007. It enables using the virtualization features of the processor to accelerate virtual machines. However, at the time of writing this report, the last x86 kernel image for gem5 available for download is 2.6.18, and the most recent configuration files provided (2.6.28) are too old to compile successfully with recent compilers (GCC5).

An investigation was made to obtain a more recent gem5-friendly Linux kernel. Indeed, gem5 does not model an exhaustive system, although it is supposed to work with an unmodified kernel. Moreover, configuring the Linux kernel is challenging because it requires a comprehensive knowledge of the target computer architecture and components. To get things work as quick as possible, the old configuration file was used to seed newer kernels configuration tool and major/minor versions were tested incrementally by only investigating the new parameters.

Several main configuration options have to be set for gem5 to boot the kernel. Those are listed in Table 5.4 and several are set by default. The `CONFIG_ BLK_DEV_INITRD` parameter is particularly important to integrate the initial ramdisk into the kernel image, because gem5 does not support the two separated.

| Feature | Needed parameters |
|---|---|
| Initial ramdisk support | `CONFIG_BLK_DEV_INITRD` |
| Symmetric multiprocessors | `CONFIG_SMP` |
| Serial console display | `CONFIG_SERIAL_8250` |
| SMP interrupt controller | `CONFIG_X86_IO_APIC` |
| KVM support | `CONFIG_HAVE_KVM` |
| PIIX IDE controller | `CONFIG_PCI`, `CONFIG_IDE`, `CONFIG_BLK_DEV_IDEPCI`, `CONFIG_BLK_DEV_PIIX` |

Table 5.4 – Required kernel configuration options for gem5 emulated system

The result of investigations led to a working Linux kernel 3.4.x able to boot with *KvmCPU*, which configuration file is available on-line, and compilation details in Appendix A.2. It was not possible to get the Universal Asynchronous Receiver Transmitter (UART) working with more recent versions, making very difficult finding the failure in emulated components in the gem5 sources. A more recent investigation revealed that the `verify_cpu` function of Linux kernel actually was the origin of this problem, now checking the value of the stack pointer which is usually set by the bootloader (GRUB, LILO...), and not by gem5.

Making the gem5 KVM CPU model run was quite chaotic in a first time. It seems that the reference machine processor (Intel Core 2 Duo E8400) used for testing the simulation framework did not support well the gem5 KVM initialisation process. Indeed, while QEMU was running correctly in KVM SMP mode, the kernel crashed on gem5 with the *cpu_stuck ?* panic message at the boot process of the secondary processors. A later run on another host machine (Intel Core 2 Quad Q9300) revealed this disparity. A pull-request on the gem5 developer board suggests that there indeed is a problem with KVM initialisation on several machines [14]. However, this patch did not solve the problem, and was not merged as it was qualified as having trouble with AMD processors. This was no more investigated, as it is out of the scope of this work, and a working environment was found.

### 5.4.3 Configuring a gem5-friendly Linux distribution with PARSEC

In order to produce the disk image containing all the software required for the simulation, the QEMU tool was used. The chosen Linux distribution was Debian Wheezy, for its large

package collection, available netinstallers for the gem5 supported ISAs, and the use of *init* as initialization system, which is easier to tweak than *systemd*.

The disk image was configured such that all files are put in a unique partition of approximately 10GB. Once Debian is installed, PARSEC can be downloaded, extracted and built. The required files and details for semi-automatically building that disk image are respectively available on-line and in Appendix A.3.

When booting, the operating system loads several services (network, firewall,...) that takes time and may not be completely supported under gem5. To avoid that situation, a new initialisation script `/etc/init.d/rcS` was written to be swapped with the original when booting under gem5. This script, after being loaded by *init*, only mounts the disks of the `/etc/fstab` file in read/write mode and checks for an input from the simulator, otherwise it drops to the shell.

### 5.4.4 Integrating the gem5 MMIO into PARSEC for communication

The gem5 simulator offers several possibilities for communicating between the simulator and the workload. This is useful for several reasons. First, to avoid hardcoding the benchmark to launch in the image disk `/etc/init.d/rcS` file. Indeed, gem5 provides a way to load a file from the host to the guest system. Second, to save checkpoints and use fast-forwarding features. Third, to reset and dump the simulation statistics at desired moments in the code.

This interaction is done via a shared Memory Mapped Input/Output (MMIO) space between the guest operating system and the simulator. This shared space is situated at address 0xFFFF0000 and has a 64KB size. Several functions are possible. The simulator detects when the specific address is written with a function value, process it and returns a return code written to that same shared space. A small standalone program offering these features at the command line level is provided.

A first experience with this feature under gem5 with the KVM CPU model revealed that at reading the simulation script specified to gem5, the guest ended in an infinite loop. An investigation in the assembly interface led to the conclusion that the the read value was always erreneous, and helped finding a patch for that issue on the gem5 developer board.[15], concerning the way thread context are declared dirty by the KVM CPU model. However, this fix was never merged and applying this patch breaks some of the emulated system calls. In this work, as this does not really matter in the desired use case, this patch was applied. Actually this can partly be solved by adding some checks in the gem5 code to know whether simulation is carried out by the system-call emulation or full-system mode.

To only capture the region of interest of the benchmarks, the integration of statistics reset and dumping at the right places in the PARSEC source code was needed. Fortunately, a hook library is provided in PARSEC which already integrates such a support for the commercial SIMICS simulator. A patch for PARSEC3 is available on-line and details on how to apply it to the original source code are available in Appendix A.3.

Several problems occurred during this step. Indeed, the first manner explored to do this integration was to include the specific assembly snippets directly in the PARSEC code. However, PARSEC is made of multiple shared modules and the linker always seemed to break things while indicating success build success, resulting in some failures of the benchmark executions with *trap : invalid opcode* error messages. This problem was fixed by simply let the PARSEC hooks call the standalone program interface, which is a bit less inefficient, but does not really matter for this work purposes.

Once the `/etc/init.d/rcS` is replaced with the one integrating gem5 communication, booting on QEMU won't make it anymore, as the MMIO space is not supported in QEMU. To avoid booting the disk by changing the initialization program and possibly damaging the file-system by killing the virtual machine not properly, the image file-system can be mounted on the host by using the `qemu-nbd` tool. This is described in Appendix A.4.

# Chapter 6

# Analysis of common coherence protocols and hierarchies

Directory coherence protocol have become the state-of-the-art in the recent years due to the lack of scaling of bus snooping protocols. Many processor designers nowadays integrate, on chip multiprocessors, cache coherent interconnect like crossbar, meshes, or event point-to-point connection, with various memory hierarchies. Several complex state-of-the-art memory hierarchies and their tied directory coherence protocols have been available in gem5 since the software apparition. In this chapter, those protocols are introduced with their memory hierarchy and their performance, concerning latency and bandwidth usage, are analysed by running the benchmarks prepared in Chapter 5, under several hierarchy variations. A simple MI protocol is simulated to illustrate the benefits of designing good cache coherence protocols and memory hierarchies in multiprocessors.

While their main aspects are introduced, the protocols implementation details are willingly omitted for two reasons. First, it would require a long discussion on their implementation choice. Second, coherence protocols are often referred to by their stable states as they provide the major optimization for a given hierarchy. The following analysis focuses on the main architectural aspects, and the impact of the memory hierarchy variations on their performance. More detailed descriptions of the protocols are available in Appendix B.

## 6.1 Simulation environment

The basic characteristics of the simulation environment are summarized in Table 6.1. The simulations were first run using the fast-forwarding KVM CPU model, and then re-run using the checkpoint to perform the annotated simulation using the Ruby memory model. The *simsmall* PARSEC input set was used mainly due to the speed of simulation, preventing the acquisition of simulation results for big inputs in a reasonable time.

| **Characteristic** | **Value** |
|---|---|
| Host CPU | Intel Core 2 Quad Q6600 2.4GHz |
| Host memory | 4GB DDR2-800 RAM |
| Guest CPU number | 4 |
| Guest CPU (Ruby) | x86 M5 detailed 1GHz |
| Guest CPU (Fast-forward) | x86 M5 KVM |
| Guest memory | 1GB DDR3-1600 RAM |
| Guest interconnect | Crossbar |

Table 6.1 – Basic characteristics of the simulation environment

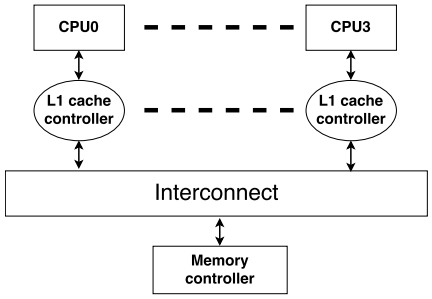
All the hierarchies presented below were simulated with a subset of the PARSEC benchmark suite under several design variations. In order to obtain a sufficient set of simulations under a reasonable time, all those design variations and benchmarks simulations were distributed in a automated way on several identical machines, gathering the simulation results on a single Git repository.

The simulated system variations include the size and associativity of L1 and L2 cache, the cache block size, and the number of directories for each memory hierarchy, making a total of 90 systems variations for each of the 8 run benchmarks. Each of these simulation provides a large set of statistics about the system execution. Analysing such a huge set of data was only possible by writing an analysis tool, in MATLAB, parsing gem5 statistics and making desired projections of these simulation statistics on bar plots. More information about this developed tool can be found in Appendix C.

## 6.2 Proposed hierarchies and protocols

### 6.2.1 One-level MI

The first studied memory hierarchy is illustrated in Figure 6.1a and rely on a MI (or VI) protocol similar to the protocol introduced in Section 4.1.2. The hierarchy has a unique level of instruction and write-back data caches, which are private to each CPU node, and kept coherent via a directory controller (the memory controller). As there is only one level, no inclusion or exclusion property applies here. As for all the following hierarchies, a DMA controller, not represented, is also connected to the network. The default parameters used for simulating this hierarchy are listed in Table 6.1b.




(a) Memory hierarchy

| Parameter | Value |
|---|---|
| Number of directories | 1 |
| L1 instruction cache size | 64kB |
| L1 instruction cache associativity | 2 |
| L1 data cache size | 64kB |
| L1 data cache associativity | 2 |
| Cache line size | 64B |

(b) Default parameters

Figure 6.1 – MI protocol based memory hierarchy and defaults parameters

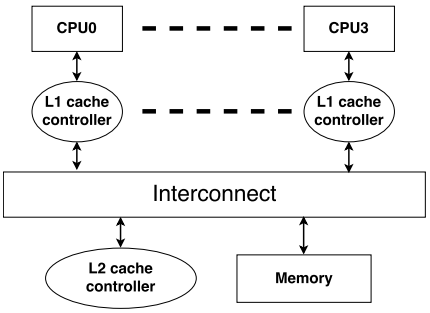
The cache controller mainly works in a similar fashion as described in earlier chapters. Note that the *GetX* notation is used instead of the *GetM* notation introduced earlier. On a replacement, or invalidation, it issues a write-back request to the directory and waits for an acknowledgement. When the directory forwards a request to the controller, the M block is sent to the requester.

The directory controller keeps the owner of the block in its memory, and, on a *GetX* request, fetch the block from memory if it has no entries, or forward the request to the owner and transfers ownership. On a write-back request, it writes data to memory and sends an acknowledgement.

### 6.2.2 Two-Level MESI

Common cache hierarchies nowadays include several level of caches to improve the miss rate. This design proposes a two-level hierarchy illustrated at Figure 6.2a, with the Level 1 cache (L1) at the CPU side and the Level 2 cache (L2) at the memory side. The cache levels have their

own controller which connect on the same interconnect, and are orchestrated by a directory controller (the memory controller), whose block states are located in the L2 cache. Inclusion is maintained between the L1 and L2 cache. The directory is responsible for fetching data from memory. The default parameters for this hierarchy are listed in Figure 6.2b.




(a) Memory hierarchy

| Parameter | Value |
|---|---|
| Number of directories | 1 |
| L1 instruction cache size | 64kB |
| L1 instruction cache associativity | 2 |
| L1 data cache size | 64kB |
| L1 data cache associativity | 2 |
| L2 data cache size | 2MB |
| L2 data cache associativity | 8 |
| Cache line size | 64B |

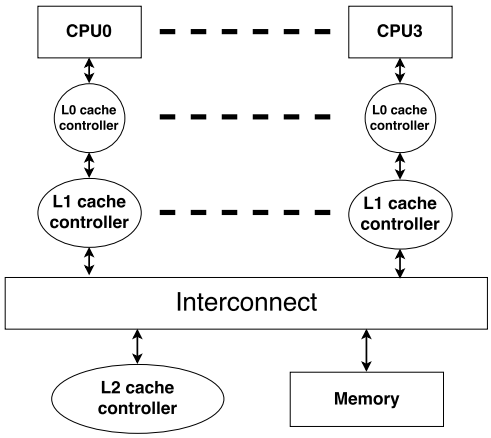
(b) Default parameters

Figure 6.2 – Two-Level MESI protocol based memory hierarchy and defaults parameters

This design is different from the designs studied in Section 4.4.2. First, the L2 and directory have separate controllers. Second, this design supports silent evictions. This means that the sharers list may contain old sharers and forward many messages to non-sharers of a block.

### 6.2.3 Three-Level MESI

To improve the latency, smaller caches can be put in front of other caches. This design extends the previous two-level hierarchy with a third level of caches, actually Level 0 cache (L0). This is illustrated in Figure 6.3a.




(a) Memory hierarchy

| Parameter | Value |
|---|---|
| Number of directories | 1 |
| L0 instruction cache size | 4kB |
| L0 instruction cache associativity | 1 |
| L0 data cache size | 4kB |
| L0 data cache associativity | 1 |
| L1 data cache size | 64kB |
| L1 data cache associativity | 2 |
| L2 data cache size | 2MB |
| L2 data cache associativity | 8 |
| Cache line size | 64B |

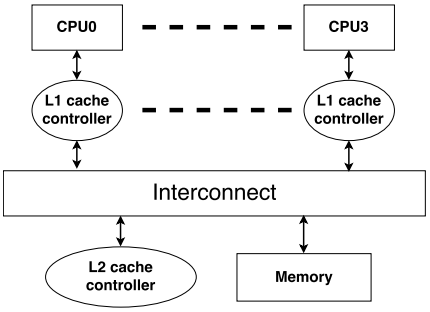
(b) Default parameters

Figure 6.3 – Three-Level MESI protocol based memory hierarchy and defaults parameters

In this design, the L0 caches have their own controllers, and the L1 controllers are modified to receive and respond messages to and from the L0 cache controller. For instance, when L0 issue a request for a read access, L1 will respond with data or initiate such a transaction with L2, using a new state for representing the pending operation that must be forwarded to L0. As the design is inclusive, when a data is evicted from L1, it must not be in L0 anymore. In this case, L1 forwards the eviction to L0.

### 6.2.4 Two-Level MOESI

Keeping blocks in owned state may be very useful to avoid unnecessary traffic to the memory by directly sending dirty data to a cache controller issuing a read request. In this design, illustrated at Figure 6.4a, the Owned state is added to the previous two-level hierarchy using a MESI protocol. The directory is still considered as the memory controller. The default parameters of the hierarchy are given in Figure 6.4b.




(a) Memory hierarchy

| Parameter | Value |
|---|---|
| Number of directories | 1 |
| L1 instruction cache size | 64kB |
| L1 instruction cache associativity | 2 |
| L1 data cache size | 64kB |
| L1 data cache associativity | 2 |
| L2 data cache size | 2MB |
| L2 data cache associativity | 8 |
| Cache line size | 64B |

(b) Default parameters

Figure 6.4 – Two-Level MOESI protocol based memory hierarchy and defaults parameters

This protocol L2 cache controller has many more transient states that those introduced in Section 4.4. It indeed supports a hierarchy of multiple chip multiprocessors. Therefore, the L2 controller implements intra-chip inclusion (between one L2 cache and several L1s) as well as inter-chip exclusion (between several L2s, and thus, several processors), requiring the L2 to act like an L1 with other L2s. However, this feature is not implemented in the configuration scripts and does not seem to be feasible in gem5 [1] and is thus not explored here. Therefore, the protocol overhead has to be taken into account during the analysis.

### 6.2.5 AMD MOESI (MESIF) Hammer

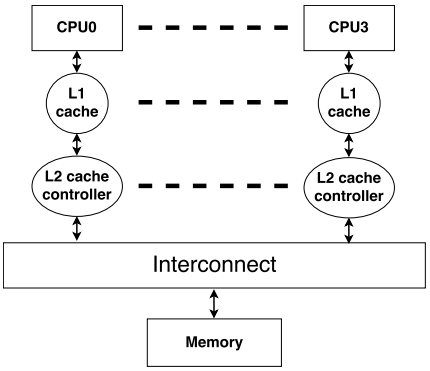
In single-processor systems, components are often connected on buses, and there is only few contention. In shared-memory multiprocessors systems, the interconnect is highly solicited by the several processors that access memory at a high rate. Therefore, the interconnection network may become a huge bottleneck.

The AMD K8 Hammer processor architecture take this fact into account. In this design, illustrated at Figure 6.5a supporting a MOESI protocol, each of the CPU has a wide L2 cache. The default parameters for this hierarchy are listed in Table 6.5b.

However, the coherence protocol differs from the previous introduced hierarchies by several main aspects. First, the directory does not maintain the list of sharers or the owner and broadcast the requests. The protocol however implements the HT (HyperTransport) Assist technology, which is a major optimization of this bottleneck: the L2 act as a directory cache, learning which processor possesses a block (owner and sharers) via the directory broadcast, like switches do on a computer network. Therefore, when this cache contains information, broadcast can be avoided.

Second, exclusivity is transferred from cache to cache, and therefore integrate the conventional Owned state optimization, because requested data in M state can be transferred with exclusivity to the requester.

[1] The gem5 simulator is the merger of M5 and GEMS as a full open-source simulator. GEMS worked with Simics, a commercial simulator, and was the original project the coherence protocols in SLICC are from. The feature does not seem to have been ported during the migration, as gem5 does not seem to support multiple networks at a time, a requirement for such a hierarchy.




(a) Memory hierarchy

| Parameter | Value |
|---|---|
| Number of directories | 1 |
| L1 instruction cache size | 64kB |
| L1 instruction cache associativity | 2 |
| L1 data cache size | 64kB |
| L1 data cache associativity | 2 |
| L2 data cache size | 2MB |
| L2 data cache associativity | 8 |
| Cache line size | 64B |

(b) Default parameters

Figure 6.5 – AMD Hammer MOESI protocol based memory hierarchy and defaults parameters

Third, the O state in this protocol is a (F)orward state as the block is clean in this state, and the protocol is more like a MESIF protocol. The F state is given for blocks whose controllers receive a *GetS* message while they own the block in E state. The controller becomes responsible for responding for data requests in place of the directory/memory. Compared to the two MESI protocols, this design allows having multiple sharers while having a controller respond for requests.

Eventually, L1 and L2 caches both share the same controller to enforce exclusion between L1 and L2 caches, preventing the L2 to store L1 content a second time.

## 6.3 Overall analysis

In this section, the different hierarchies and protocols simulated are compared according to common simulation characteristics in order to highlight their global performance for the given benchmarks and provide general remarks. This section thus gives a particular interest to the benchmark execution times, the performed memory accesses, the network traffic, as well global reactions faced to quantitative design variations.

### 6.3.1 Execution time

Execution time, or speed, is one of the main measures that characterise the performance of computers. Figure 6.6 illustrates the time spent in the selected benchmarks Region of Interest (ROI), that is, the time spent in the true parallel portion of the program, for the default hierarchy parameters.

The overall trend reveals that more sophisticated protocols perform better. For instance, the two-level MESI protocol performs always better than the basic MI, which is obviously an expected result, and the Hammer protocol usually performs better than other protocols. However, two observations can be made from this graphs. First, the three-level MESI seems to barely outperform MI and is not better than the two-level MESI. Yet, adding a level of cache is expected to provide better latency. However, an informed reader will notice that the added L0 cache is only 4kB-long with an associativity of 1. The observed results suggest the cache size is too small, and produces many misses which requires issuing coherence requests to the higher level. Another observation is that Hammer does not seem to perform as good on the *fmm* benchmark. A first hypothesis would be that this specific benchmark makes the caches issue a large coherence requests on a large set of non-cached data, calling the directory controller, hosting its table in main memory. This will be investigated further below.

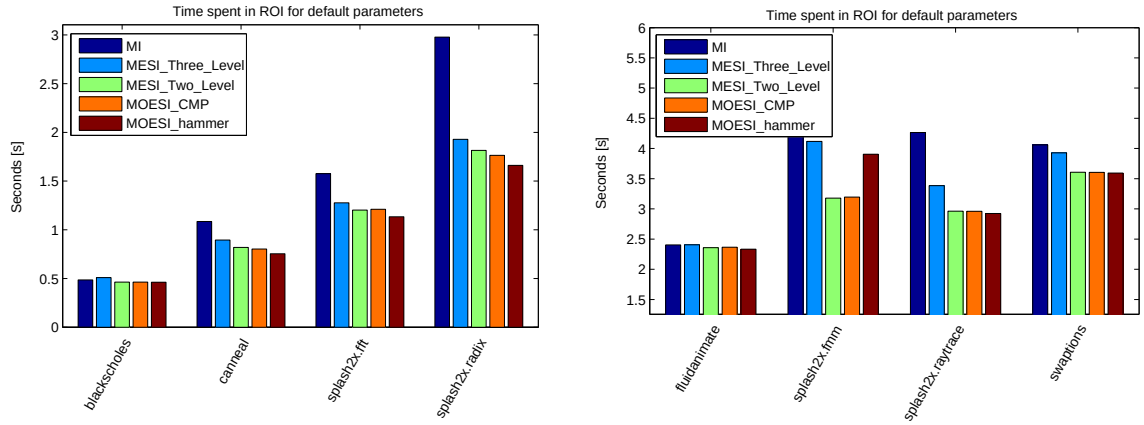



Figure 6.6 – Time spent in the benchmarks ROI, for default parameters

### 6.3.2 Memory accesses

Figure 6.7 illustrates the amount of data written and read from main memory with the default parameters for the different benchmarks. The MI protocol obviously generate a lot of write and reads from memory. Indeed, this simple protocol, whose memory hierarchy has no common cache, forces writing back to memory at each invalidation provoked by the data request from another CPU, making almost as many reads as writes in this parallel portion of the benchmarks.

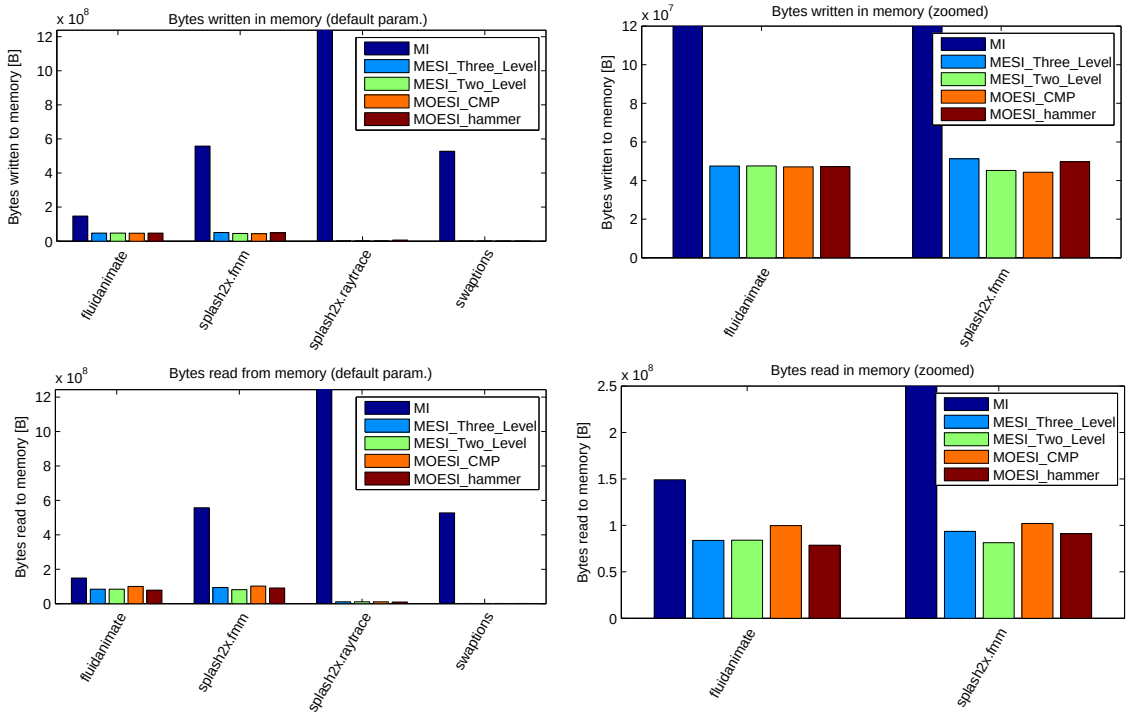
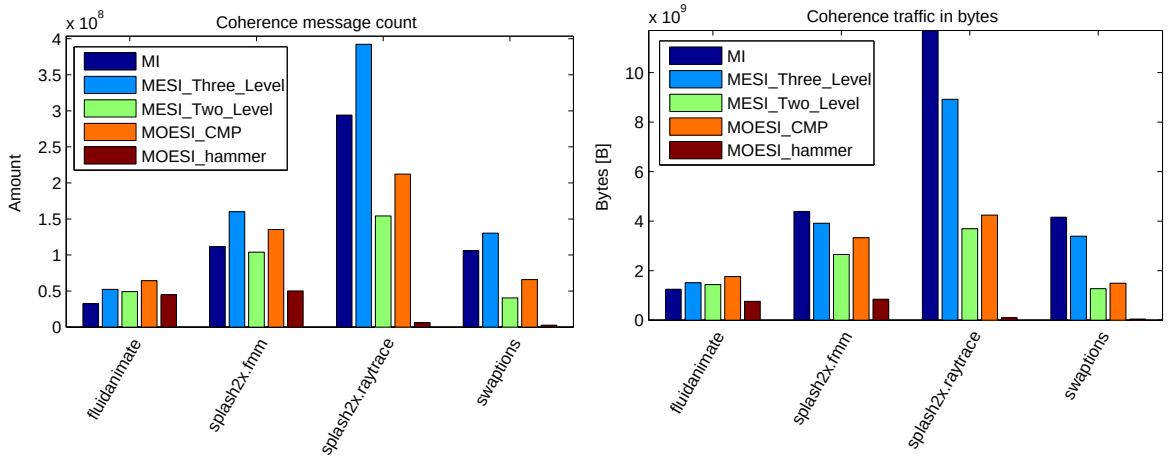



Figure 6.7 – Bytes written and read from memory (zoomed on right)

Other hierarchies seems to behave more uniformly. Like the MI hierarchy, the Hammer hierarchy does not possess a shared cache. However, it still performs pretty well concerning memory accesses compared to other hierarchies sharing a common cache. This has two main reasons: first, the additional states preventing untimely block invalidations reduce the number of memory write-back. Second, the large L2 cache significantly reduces the amount of misses.

### 6.3.3 Network traffic

The total network traffic for selected benchmarks is illustrated in Figure 6.8. Global trend shows that, on the one hand, the MI protocol generates a higher byte traffic and a lower amount of coherence message than the three-level MESI. This can be explained by the fact the MI protocol has only one level of caches communicating with the directory, generating a moderate amount of messages, but also a lot of data transfer with memory, exacerbated by the untimely invalidations. On the other hand, due to its small cache size, the three-level MESI generates a lot of coherence traffic. This will be analysed further.




(a) Amount of coherence messages

(b) Total number of bytes

Figure 6.8 – Network traffic in message count and byte count

While it is quite expected that the two-level MESI perform better than the MI protocol due to the new states impact, the two-level MOESI protocol generates more traffic in terms of amount of messages and byte transfer. Indeed, while the MESI protocols and the Hammer protocol supports silent evictions of clean data, the two-level MOESI does not. Shared data indeed represent a huge part of the total data. Eventually, the Hammer protocol, compared to all other protocol, generates significantly less traffic, mainly due to the unique cache controller for the two cache levels, avoiding message exchange between the different levels, and its exclusiveness handling.

### 6.3.4 Quantitative hierarchy variations

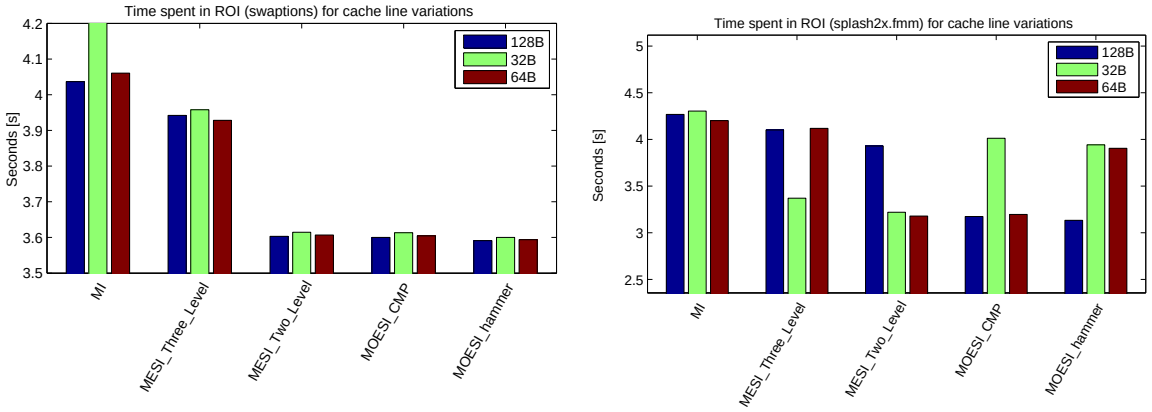
The performance of coherence protocols can be impacted by quantitative changes in the memory hierarchy. This section aims at analysing the impact of such variations.

#### Block size

Increasing the cache block size has the major advantage of better exploiting the locality principle and thus provides intrinsic data pefetching. However, in shared-memory multiprocessors, increasing the cache block size also increase the probability another processor has to access the same block of data.

Figure 6.9 illustrates the *swaptions* and *fmm* benchmarks execution time for varying block size. While the *swaptions* execution time tend to vary in an intuitive way (larger the block size, shorter the latency), this is not always the case for the *fmm* benchmark.

The two MESI protocols do not perform well with a large block size. An hypothesis for explaining this fact is the addition of the Owned state in MOESI protocols. Indeed, the *fmm* benchmark use a significant amount of producer-consumer patterns [3], and the Owned state




(a) *swaptions* benchmark (b) *fmm* benchmark

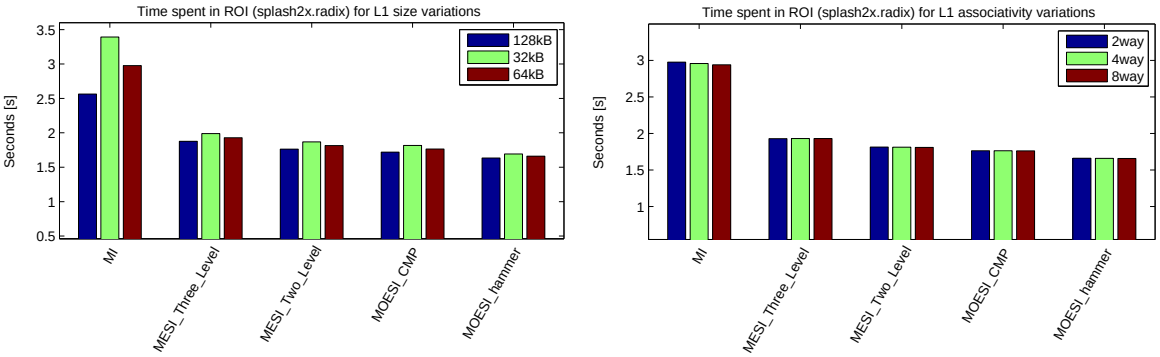
Figure 6.9 – Time spent in the ROI, for varying cache block size

helps reducing the latency in the cases a CPU request a read permission for a modified block cached at another CPU, by avoiding sending dirty data back to the directory.

### L1 size and associativity

Increasing the L1 cache size aims at reducing the miss rate by allowing more blocks to be stored. Figure 6.10a illustrates the time spent in ROI by the *radix* benchmark for varying L1 cache size. Not surprisingly, the latency tend to decrease with a higher cache capacity. However, these differences are more significant for the MI hierarchy, as it requires the memory to process data at each coherence request. Indeed, missing data from one cache can be hold by another cache (another level, or another processor cache).

The cache associativity can be increased to reduce the eviction of conflicting data. As illustrated in Figure 6.10b, varying the L1 cache associativity does not have a significant impact on the protocol performance. This observation has to be related to the fact the benchmarks input data was not too large to enable faster simulations, probably leading to only a few conflicts.



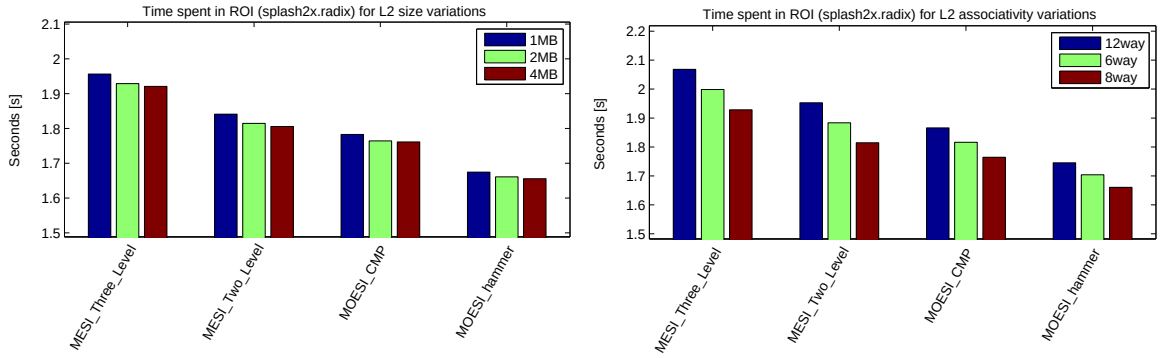

(a) Varying L1 size (b) Varying L1 associativity

Figure 6.10 – Time spent in the ROI for *radix* benchmark, for L1 parameters

### L2 size and associativity

Varying the L2 cache properties has the same goal as varying the L1 cache properties: reducing the number of capacity and conflict misses. This observation is quite similar for the L2 caches, as illustrated in Figures 6.11a and 6.11b for the same *radix* benchmarks. It is interesting to notice

the more significant time variations when varying the L2 associativity in the hierarchies. This is mainly due to the higher associativity degrees simulated. A L2 cache with a lower degree of associativity than a L1 cache would make the system perform particularly bad when the L2 is inclusive, as conflicts in L2 would trigger evictions from L1, requiring coherence messages. While this is also true for single processor systems, this is exacerbated in multiprocessors when the L2 is shared, as the L1 caches may contain different blocks of data.

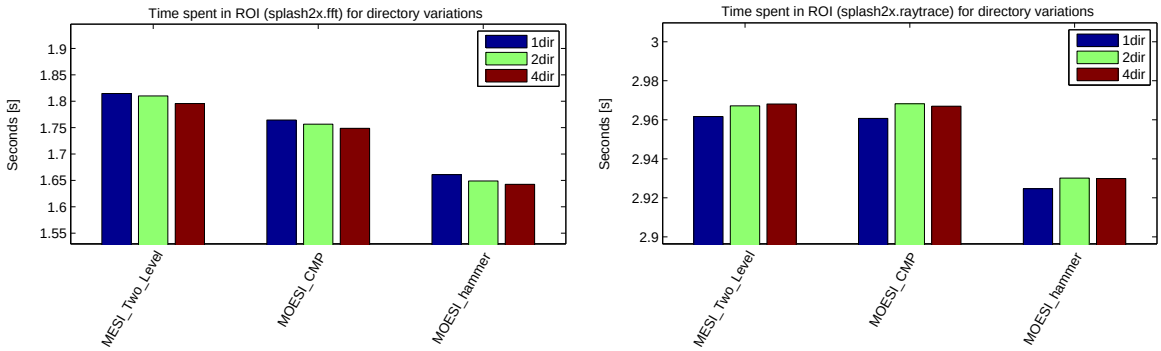



(a) Varying L2 size (b) Varying L2 associativity

Figure 6.11 – Time spent in the ROI for *radix* benchmark, for L2 parameters

### Distributed directories

In Chapter 4, distributed directories were introduced as a way to reduce the contention on the network links, providing higher bandwidth. Figure 6.12a illustrates the performance of the *fft* benchmark with different amounts of directory in the system. While these results shows a latency improvement for an increasing number of directories, this is not the case for the *raytrace* benchmark. Actually, the presented memory hierarchies did not include distributed memory chips. This means that the requests of all directory requiring actions from the main memory will eventually trigger contention on these links.




(a) Varying L2 size (b) Varying L2 associativity

Figure 6.12 – Time spent in the ROI for *fft* and *raytrace* benchmarks, for directory variations

As the *fft* benchmark does not feature much communication between the threads [3], it is less likely to saturate the links, and can still benefit from such variations if the required data by the processors are cached in other private caches. However, increasing the number of directories does not worth it in UMA systems, as the bottleneck only translates to the main memory links.

## 6.4 Detailed protocol analysis

The previous section presented global performance analysis based on execution time and memory accesses observation for the different protocols to compare their individual responses. However, no specific coherence controller was analysed, mainly because of the design particularities between all the hierarchy and protocols. This section aims at analysing this missing point by providing a controller-level analysis for each of the protocols.

### 6.4.1 One-level MI

#### Directory controller

The amount of events recorded at the directory during the execution of the benchmarks is illustrated at Figure 6.13. In these graphs, *GetX* refers to a requests for M state by another controller, and *M.GetX* refers to a request for M state by another controller while the state in directory is M. This syntax is used by the simulator and will be used throughout this discussion.

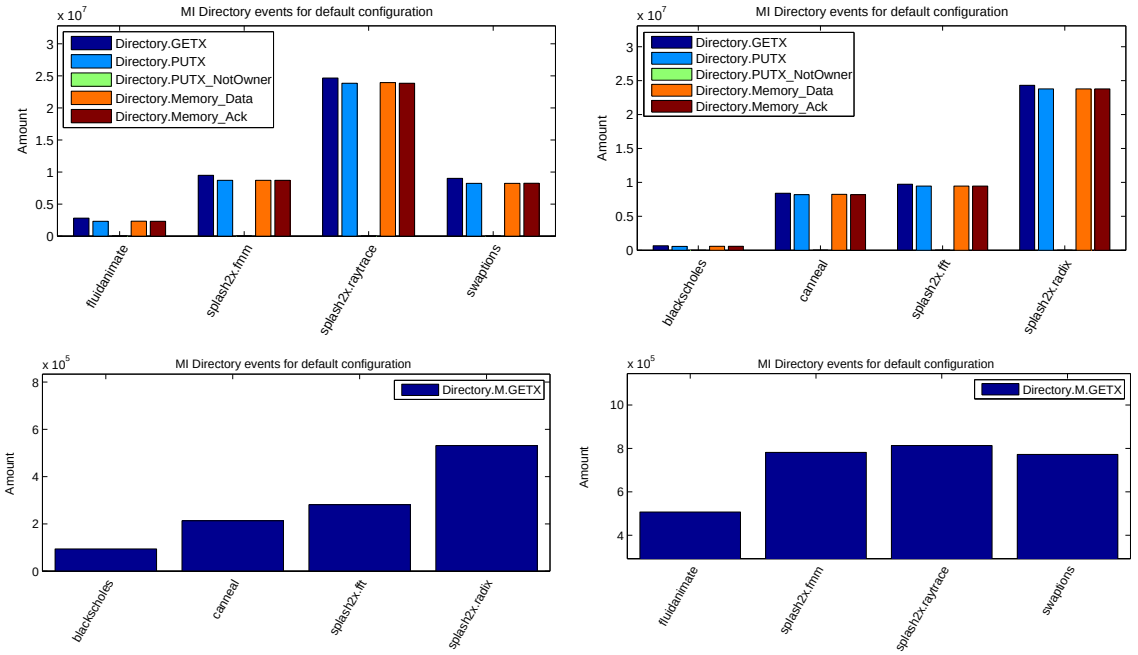



Figure 6.13 – MI directory events during executions of benchmarks

One of the main things to observe in these graphs is the similar total amount of *GetX*/*PutX* compared to the amount of memory accesses. There is also as many *PutX* than *GetX*, indicating the write-back of almost all cached data. Indeed, in this protocol, requesting a cached value (either for read or write permission) result directly in an invalidation. This result in a lot of events, because each invalidated data in directory has to be fetched again in memory. Another important remark is that, as for all the protocols, data and instructions are not differentiated. Therefore, the amount of *GetX* also include read instructions.

The number of data requested while in state M at the directory is quite large but is significantly exacerbated by the lack of an S state, not differentiating reads and writes, forcing invalidation of clean data. However, this illustrates the need for Exclusive and Owned states. Indeed, theses states will provide optimizations in those cases as the Exclusive state prevents requesting the write permission for a uniquely-cached block while the Owned state prevents from sending back data to the directory when a dirty copy is requested for read.

### L1 cache controller

Figure 6.14 illustrates the number of main events recorded at the L1 cache controllers during the execution of several benchmarks under the default hierarchy configuration. The main observation that can be done is the order of magnitude of these different events. The actual number of invalidations requested by the directory is actually quite low : several hundreds. On the other hand, the number of forwarded *GetX* messages received is in the order of several ten thousands, which has to be compared with the total amount of data requested by the controller, which is a hundred time larger in terms of magnitude.

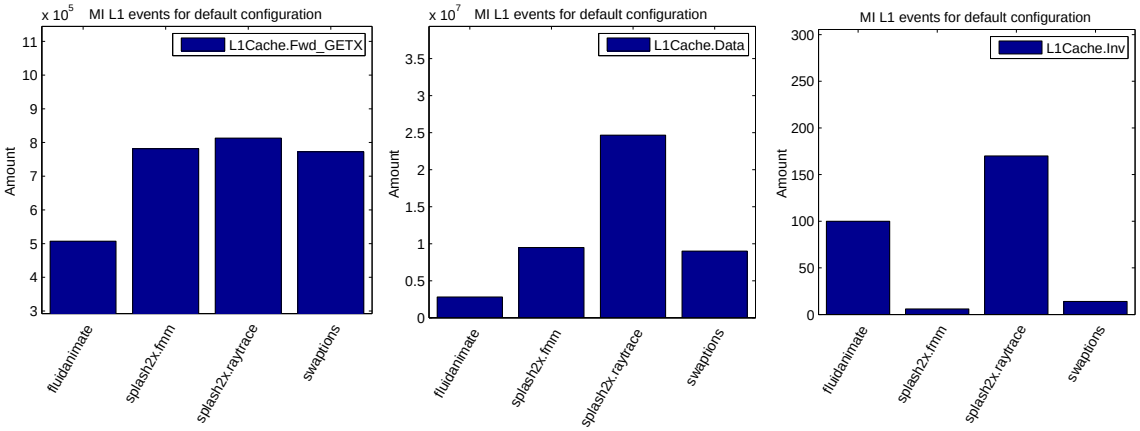



Figure 6.14 – MI L1 controller events during executions of benchmarks

As already said, the MI protocol will triggers untimely invalidations, making a processor, that has invalidated its data due to a forwarded request but wants to use it again, request that block again.

### Network

Figure 6.15 illustrates the amount of messages and bytes sent over the different separate networks used for the protocol.

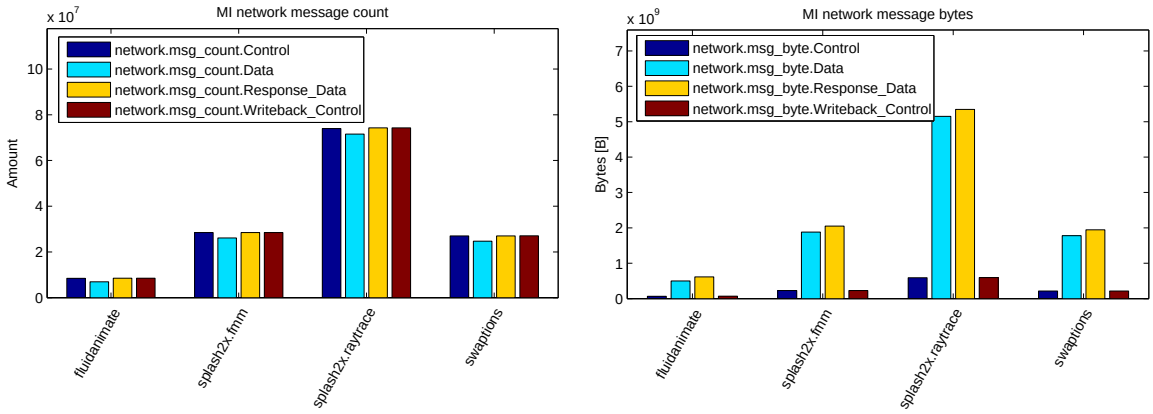



Figure 6.15 – MI network traffic for several benchmarks in default conditions

One can easily see that the message count is almost identical for each network. This is quite expected as the protocol control messages basically trigger memory operations. The bytes count confirms that the most consuming networks are the data networks.

### 6.4.2 Two-level MESI

#### Directory controller

Figure 6.16 illustrates the amount of events recorded at the directory controller. The aware reader will notice the the typical magnitude of these amounts is ten times lower than the typical amounts for the MI hierarchy. The *Fetch* event corresponds to either a *GetX* or a *GetS* and merely corresponds to the *GetX* amount of MI hierarchy. The *Data* event corresponds to a data coming from the network, the *Memory_Data* event to a data response from memory and both barely corresponds to the *Memory_Data* event of the MI hierarchy. Eventually, the *Memory_Ack* event corresponds to a write-back acknowledgement.

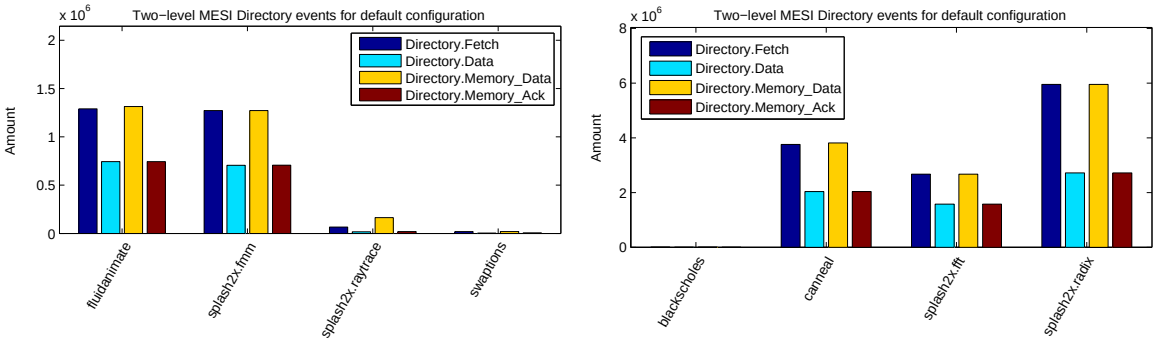



Figure 6.16 – Two-level MESI directory events during executions of benchmarks

The main reason why these amounts are significantly lower than the MI hierarchy ones is the addition of the S state. Indeed, by preventing untimely invalidations due to read requests, the cache controllers can stop reissuing requests for data they had to invalidate in read state. In the MI hierarchy, only one cache controller could have a valid copy of a block at a time. Introducing the S state avoids this situation by enabling several controllers to have read access and significantly improves the performance.

#### L1 cache controller

Figure 6.17 illustrates the events recorded by the L1 cache controller during the executions of selected benchmarks under default configuration. The *Data_All_Acks* event corresponds to data response leading to states M and I, while the *Data_Exclusive* event corresponds to the data that were delivered tagged as exclusive for the controller.

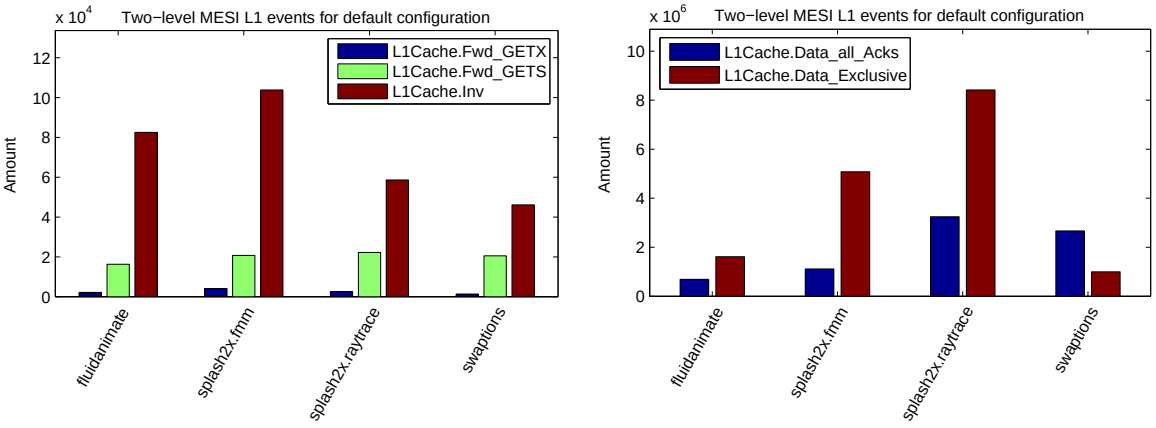



Figure 6.17 – Two-level L1 cache events during executions of benchmarks

Several major observations can be made from these graphs. First, the order of magnitude of the amount of events is again lower compared to the MI hierarchy. Second, the controllers ask for read permission more often than write permission. This is an important result as it illustrates the significant gain of adding the S state in a coherence. Third, many more invalidations are observed. This is a direct consequence of adding the S state and making the multiple sharers invalidate their copies. Fourth, a significant part of the requested data is flagged as exclusive. This is another important result as it illustrates the fact that, in multiprocessors, many block are actually often shared by zero or one processor. Thus, giving exclusivity to the cache controller allows it upgrading the block state to M without issuing a new request.

### L2 cache controller

The events recorded at the L2 cache controller confirm the trend observed for the L1 cache controller. A part of these records are illustrated in Figure 6.18. An informed reader can notice the significant amount of *PutX* (equivalent of *PutM*) performed by the different L1 controllers. These are triggered by evictions, requiring the owner controller to send the dirty copy to the higher level.

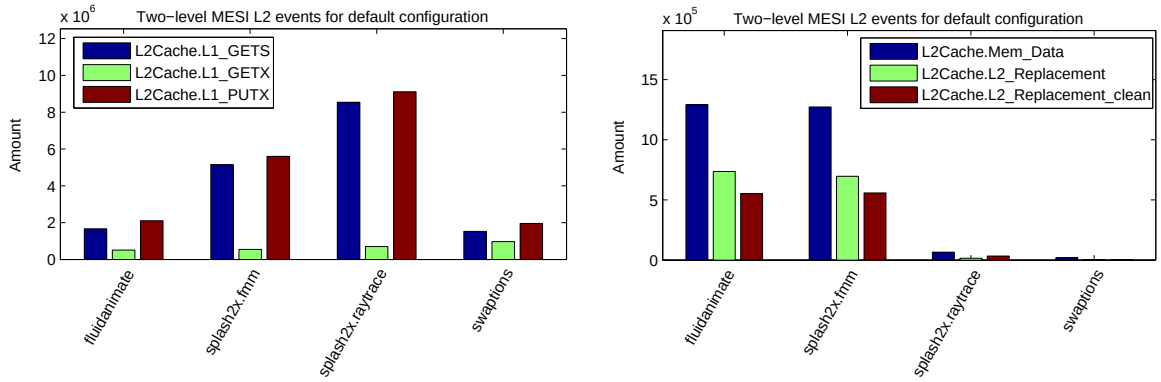



Figure 6.18 – Two-level MESI L2 cache events during executions of benchmarks

The interaction between main memory and L2 cache controller in the right part of Figure 6.18 shows that a large part of the data coming from memory to L2 cache (*Mem_Data*) is then replaced from the cache for almost all the benchmarks. This suggests that increasing the L2 cache size, for example, would increase the overall performance. This is illustrated by Figure 6.19. The intuition is confirmed for at least the *fmm* benchmark were increasing the L2 size significantly reduces the messages sent between the two components.

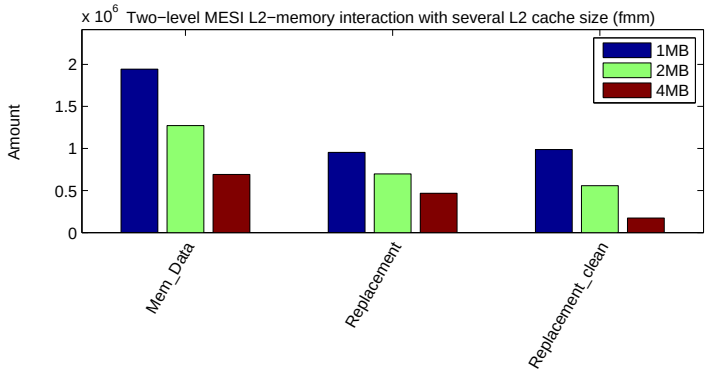



Figure 6.19 – Two-Level MESI L2/memory interactions for varying L2 size *fmm* benchmark

### Network

The network traffic for the two-level MESI hierarchy is illustrated in Figure 6.20 for a set of benchmarks. The network is subdivided in several virtual networks for requests, responses and write-back, with data and control messages (for instance, acks...) separated. The main observation that can be made is the significant reduction of write-backs and data transfers performed by the controllers, mainly due to the additions of the S and E states compared to the MI hierarchy.

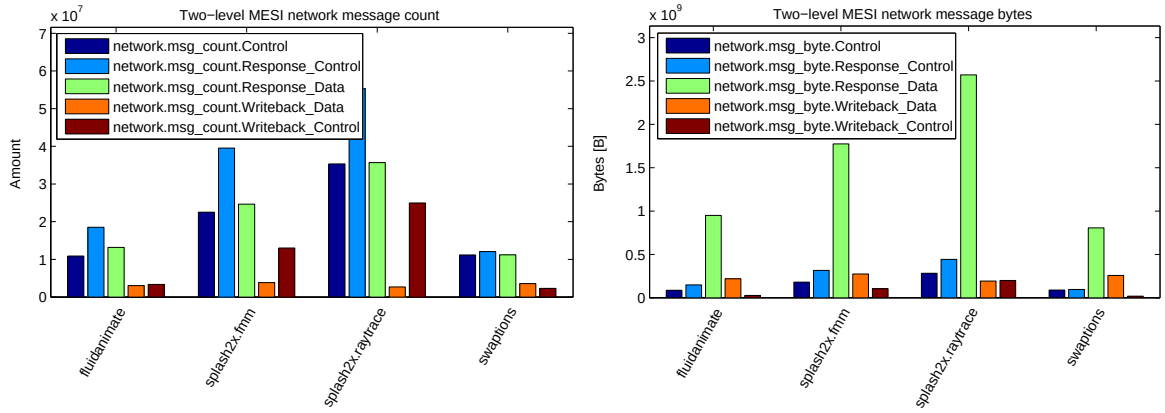



Figure 6.20 – Two-Level MESI network traffic for several benchmarks

This main observation for data transfers translates to the transferred bytes amount which is also significantly decreased compared to the MI hierarchy.

## 6.4.3 Three-level MESI

### Directory controller

The three-level MESI hierarchy is the two-level MESI hierarchy augmented with an L0 private cache level. Not so surprisingly, the events recorded at the directory are quite similar, and are available in Figure 6.21.

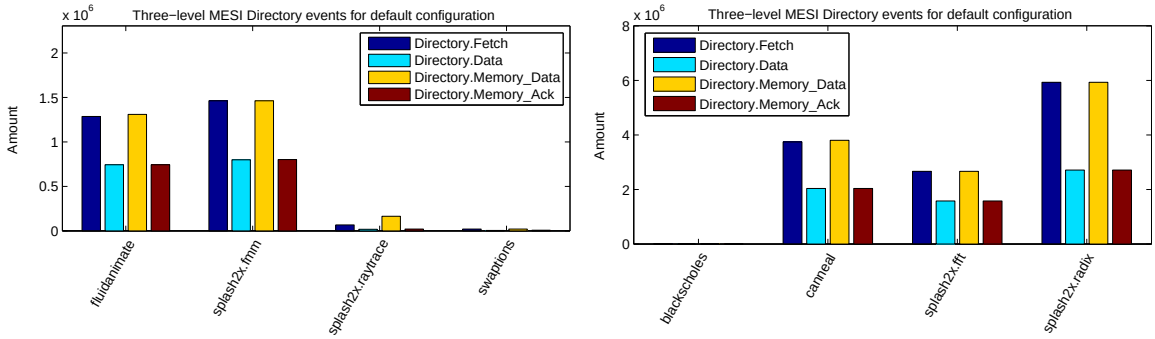



Figure 6.21 – Three-level MESI directory events during executions of benchmarks

Indeed, the architecture change suggests that the L0 cache is added to provide better latency (which was an expected result, but not achieved in practice) and not to interact in the coherence protocol itself, or, at least, not significantly. Indeed, the L0 cache is only connected to the L1 cache and does not communicate directly with the directory and the L2 cache.

### L0 cache controller

The previous analysis showed that the three-level MESI hierarchy performs very badly in terms of latency. An hypothesis that was suggested is the format of the L0 cache, which is a direct-mapped 4kB cache. This hypothesis is confirmed by Figure 6.22. The L0 controller is solicited for replacements a huge amount of time. Invalidations from L1 controller, represented on the right side of the figure, are two order of magnitudes lower that these replacements, suggesting that a complete redesign of the L0 is required.

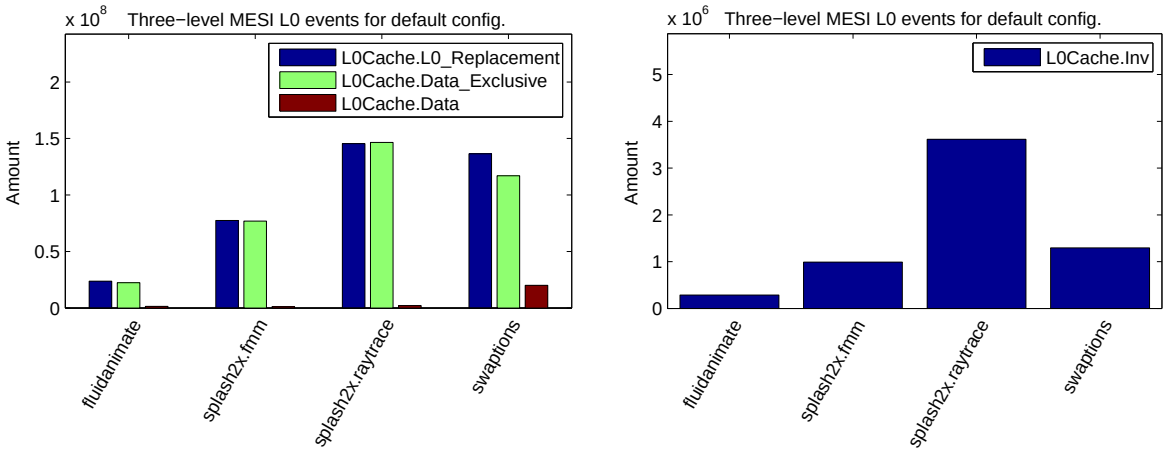



Figure 6.22 – Three-level MESI L0 cache controller events during executions of benchmarks

This is an important result as it shows that the performance of a coherence protocol can be significantly impacted by a hierarchy bottleneck such as this. Without redesigning the L0, the hierarchy will not perform well, even with a finer-designed protocol to ensure coherence.

### L1 cache controller

The events recorded by the L1 cache controller are illustrated in Figure 6.23. As expected by the design variation between the two-level hierarchy and this one, the *GetS*, *GetX* requests amounts are similar. The number of invalidations from L2 observed at L1 is a bit decreased, and the number of exclusive data get from the L2 has significantly increased. This may be explained by the fact the lower cache levels are caching less data due to the design bottleneck and make much more replacement.

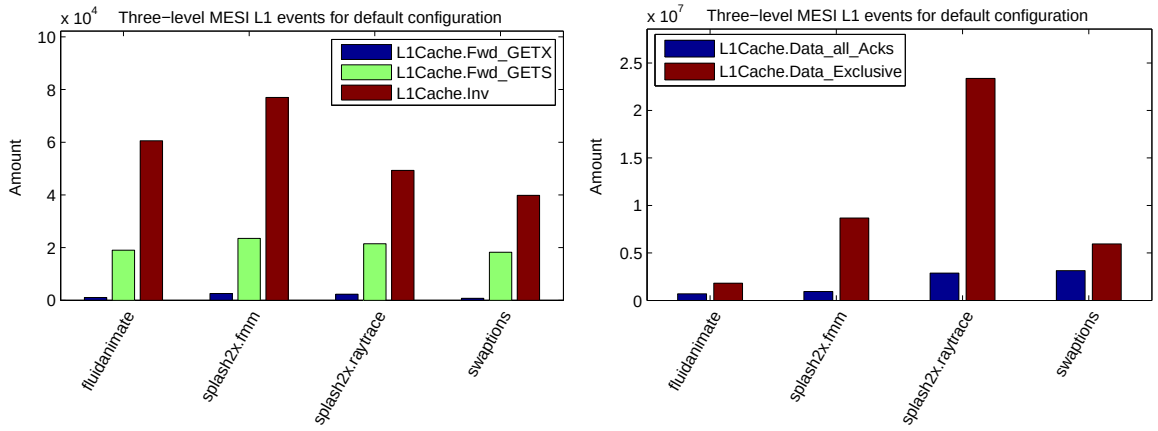



Figure 6.23 – Three-level MESI L1 cache controller events during executions of benchmarks

### L2 cache controller

Events recorded at the L2 cache controller have a similar shape than the ones recorded by the L2 controller in the two-level hierarchy. However, a slightly larger amounts of events are recorded. Those events are illustrated in Figure 6.24. This is also a consequence of the small L0 cache not holding much data. Blocks that are not in L0 have a chance to be evicted by the L1 caches, maintaining inclusion, to make room. This suggests intuitively a higher amount of coherence messages to get back those data down to L0 caches.

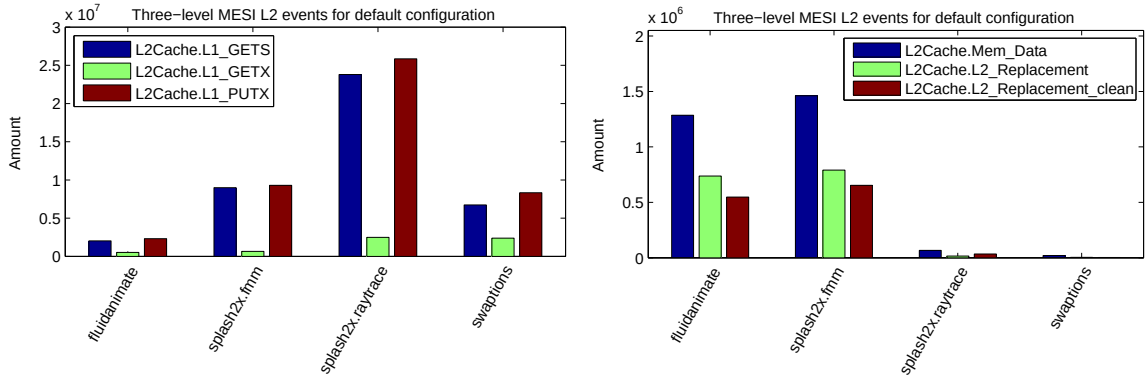



Figure 6.24 – Three-level MESI L2 cache events during executions of benchmarks

The last two controller analysis showed that a design flaw in the bottom of the memory hierarchy tree may have an impact up to the root, that is, the last level cache.

### Network

Figure 6.25 illustrates the message count and byte traffic of the different virtual networks used by the coherence protocols. Those are identical as the two-level hierarchy's ones.

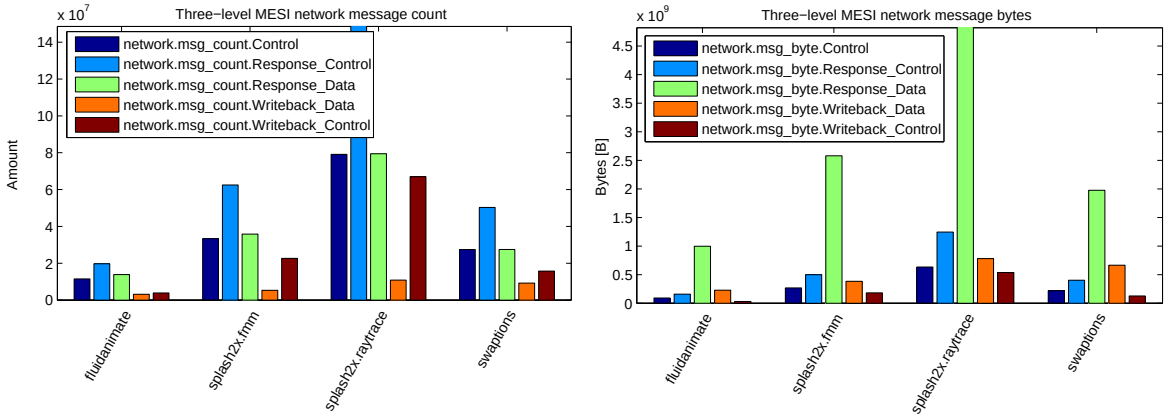



Figure 6.25 – Three-Level MESI network traffic for several benchmarks

An informed reader will notice the significant increase of control and response messages, especially visible for the *raytrace* and *swaptions* benchmarks. This is a pure consequence of what have been told in the previous controller analysis. As the L1s, maintaining inclusion, are more likely to evict blocks that will be used by L0, which is not holding the data, will favour an increasing amount of requests from the L0 to the L2, resulting in the poor performances observed when comparing the execution time.

### 6.4.4 Two-level MOESI

#### Directory controller

Figure 6.26 illustrates the amount of events recorded by the two-level hierarchy MOESI directory during the executions of the benchmarks. The *PUTO_SHARERS* events corresponds to the arrival of *PutO* message to the directory. The extremely low amount of write-back data in state O suggest that the optimization brought by this state does not concern many blocks at the first sight.

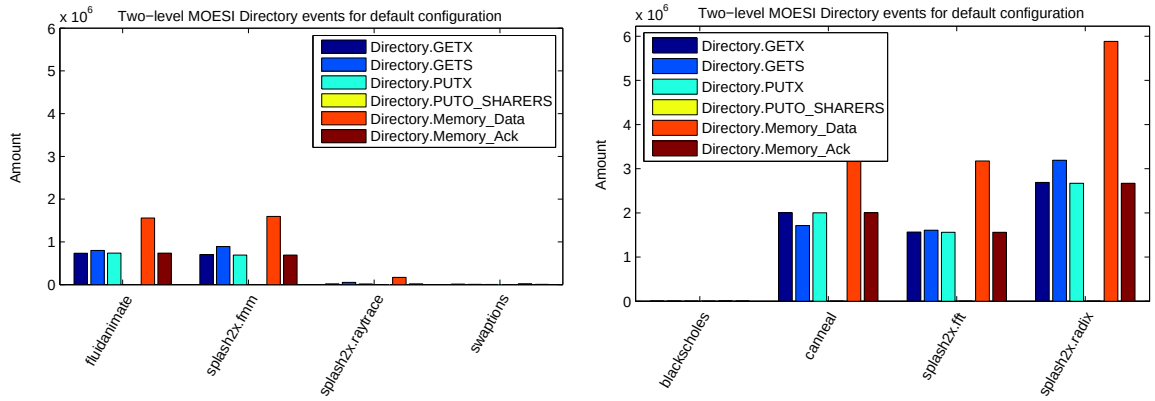



Figure 6.26 – Two-level MOESI directory events during executions of benchmarks

The other amounts of events observed are of the same order of magnitude that the amounts observed for the two-level MESI hierarchy. The *Memory_Ack* event amount roughly is the sum of the *PutX* and *PutO* events.

#### L1 cache controller

Figure 6.27 illustrates the different events recorded at the two-level MOESI L1 cache controller. Compared to the MESI protocols analysed above, these results allow highlighting important changes in those values.

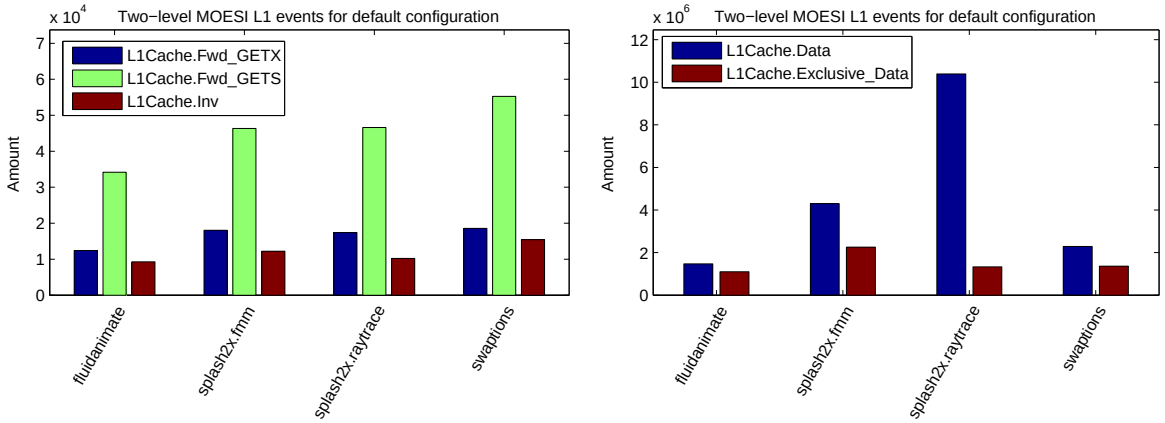



Figure 6.27 – Two-level MOESI L1 cache events during executions of benchmarks

First, the number of invalidations is significantly reduced. Indeed, by using explicit evictions, the sharers list is exact, and there is no need in sending invalidations to old sharers. Second, the number of exclusive data sent to the cache controllers is not as high as in the MESI protocols. This can be explained by addition of the Owned state to this protocol. Indeed, in

the MESI protocols, when a dirty block is requested by *GetS*, it is first evicted. However, it was shown that a large part of data is shared by zero or one processor. Therefore, at the eviction time, the sharers count is zero and exclusive data is given to the *GetS* request. The Owned state prevents this situation.

Figure 6.28 illustrates the actual amount of transactions for which the Owned state may be useful. The forwarded *GetS* observed by the cache controller still represent a large amount of the total *GetS* observed. This means that, if a small amount of blocks are concerned by the O state, those blocks are often accessed, making the optimization still worth the implementation. Indeed, most benchmarks communicate via a producer/consumer or migratory scheme, making the other processor reads the values produced by other processors. [3]

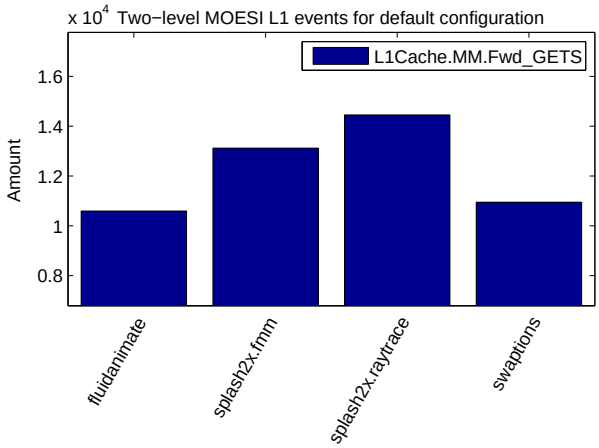



Figure 6.28 – Two-level MOESI L1 cache blocks concerned by the Owned state

In the simulations done, only four processors were used. The previous result suggests that in a many-processors system, implementing the Owned state and running programs with such sharing patterns would lead to many more performance gain.

### L2 cache controller

The events recorded at the two-level MOESI L2 controller are illustrated in Figure 6.29. In this protocol, the directory maintain a sharers list and blocks in state S are explicitly evicted. A first observation is that explicit eviction generates a lot of messages between the controllers and the MESI protocols did perform better on this point by allowing silent evictions.

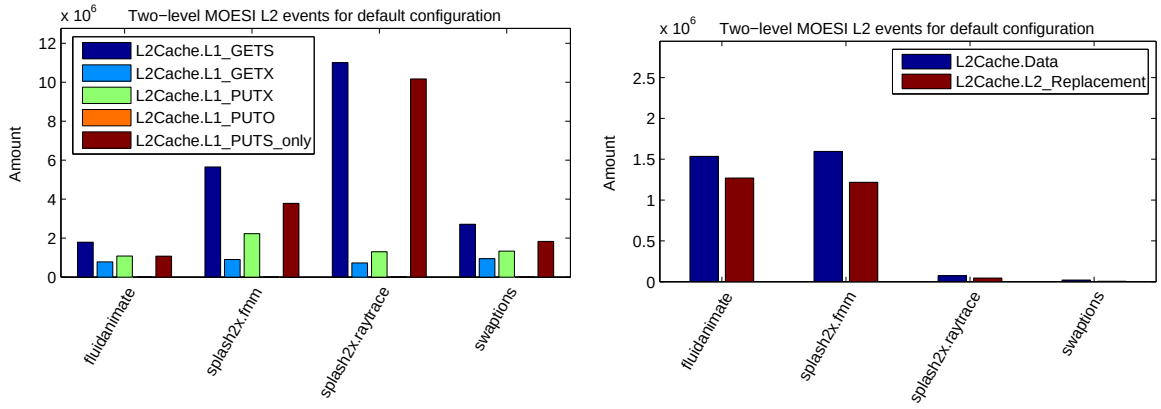



Figure 6.29 – Two-level MOESI L2 cache events during executions of benchmarks

Compared to the two-level MESI L2 cache controller, whose recorded events were illus-

trated in Figure 6.18, several observations can be made. First, the amounts of *GetS* observed by the controller are of the same order of magnitude. Second, the two-level MOESI L2 controller generates significantly less *PutX* messages. This is an important result as it illustrates the main utility of the Owned state introduction. While not so many blocks are going to this state, they are widely shared across the system, making a huge reduction of transactions and blocks. Indeed, in a typical MESI protocol, each *GetS* transaction on a block in M state would generate a *PutX* message from the L1 controller to give the dirty value back.

Once again, the amount of replacements done in the L2 cache suggests that the L2 either is too small or has a too small associativity. For the sake of briefness, this is not illustrated here.

#### Network

The network message count and byte count for the two-level MOESI protocol are illustrated in Figure 6.30. Several observations can be made compared to the message counts of the two-level MESI protocol. First, forwarded messages travel on a separate virtual network. Second, the amount of response control is dramatically reduced, mainly due to the protocol transition optimizations, that are not studied here for the sake of simplicity. Third, the amount of requests and data are of the same order of magnitude than in the two-level MESI protocol. Eventually, many more write-back control is generated. This flaw is obviously due to the explicit *PutS*, requiring the controller to send a message for each eviction of a block in state S.

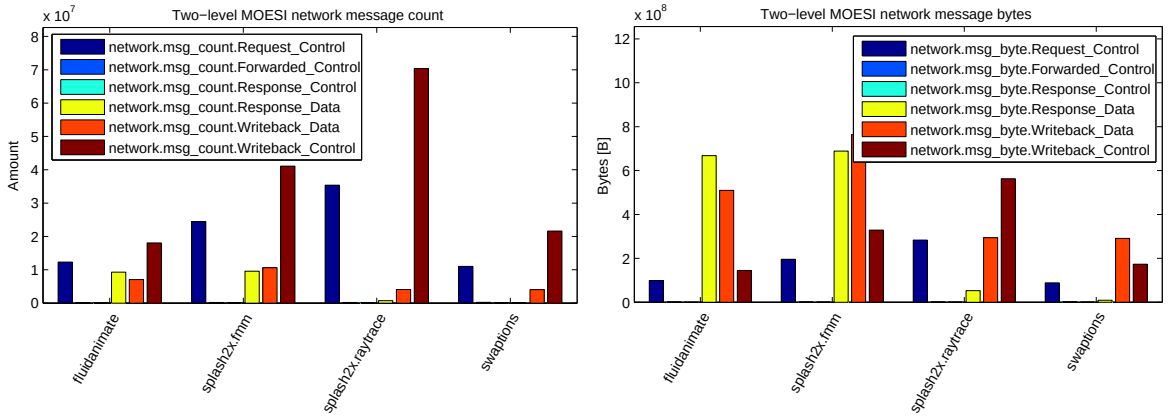



Figure 6.30 – Two-Level MOESI network message count for several benchmarks

### 6.4.5 AMD MOESI (MESIF) Hammer

#### Directory controller

Figure 6.31 illustrates a partial set of the events recorded at the MOESI Hammer directory. Some interesting facts have to be noticed in comparison with the previous two-level MOESI protocol introduced. First, the Hammer protocol supports silent evictions of blocks and therefore avoids sending a large number of *PutS* messages that were sent to the two-level MOESI L2 controller, as it was seen in Figure 6.29.

However, it can be observed that the amounts of memory transfers (*Memory_Data*) and write-back acknowledgements (*Memory_Ack*) are typically the same, due to the fact data is only written-back when it is actually dirty. Due to the amount of eliminated *PutS* messages, it can thus be concluded that this design is quite interesting.

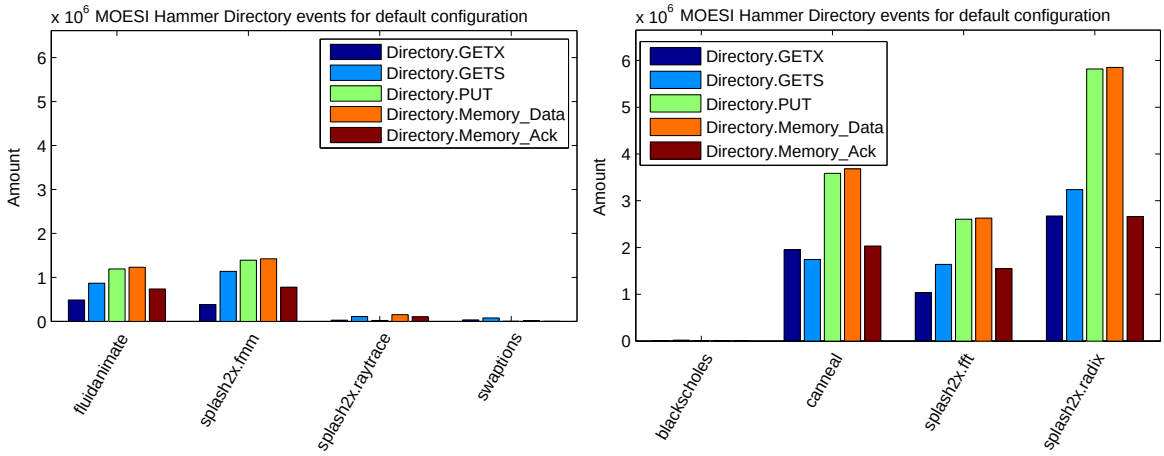



Figure 6.31 – Hammer directory events during executions of benchmarks

### L1/L2 cache controller

The partial set of events observed at the L1/L2 cache controller is available in Figure 6.32. Several main observations can be made. Moving the L2 cache ahead of the interconnect and merging both the L1 and L2 cache controller reduces the global amount of messages that must be processed, as there is no more explicit communication between the L1 and L2 cache.

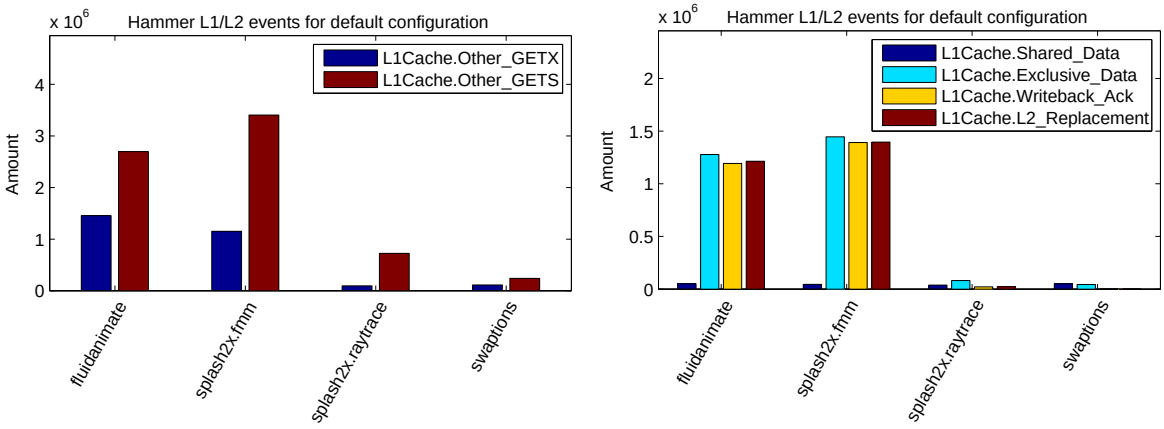



Figure 6.32 – Hammer L1/L2 cache events during executions of benchmarks

The directory broadcast requests if it does not find a potential list of sharers in the cache. Comparing Figure 6.31 and 6.32 reveals that broadcast was avoided in several situations as the number of *GetS* observed by the cache controllers do not exceed the number of *GetS* observed by the directory times the number of CPUs. This observation can also be made for *GetX* messages.

The way the cache controllers manages the exclusivity makes the reception of exclusive-tagged data by the controller very frequent compared to single shared data. This reduces the number of messages sent when the processor issuing a *GetS* on a block cached on another cache in M state then wants to upgrade this block in M, typically in migratory sharing patterns.

### Network

The network message counts and transferred bytes observed for the different virtual networks in the Hammer protocol are available in Figure 6.33. These messages counts are an order of

magnitude behind those observed for the previous two-level MOESI protocol. This is mainly due to the presence of only one controller with a large cache size. While the previous MOESI protocol did not made an intensive usage of its forwarding network, the Hammer broadcast network is the most used network by the protocol. The response control link is also much used due to the broadcast, which forces the controller to answer to requests, positively or negatively. Eventually, while the data network is used in a similar fashion, the write-back control link is significantly less used due to the implicit evictions.

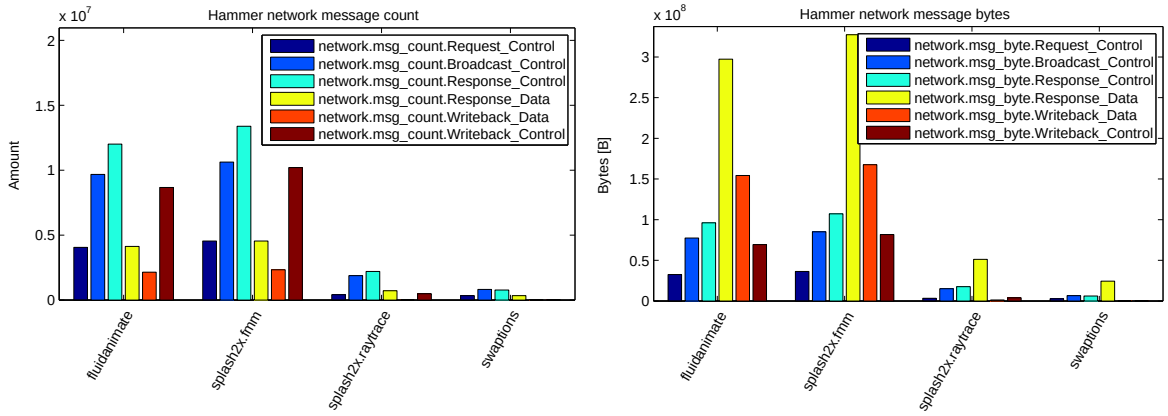



Figure 6.33 – Hammer network message count for several benchmarks

## 6.5 Concluding remarks

The detailed analysis performed in this chapter has shown several important results that can be re-used when designing a cache coherence protocol:

- The (S)hared, (E)xclusive and (O)wned/(F)orward states actually brings significant performance improvements compared to a simple binary scheme that can be found in single-processors systems. The S state brings the possibility of having multiple readers at the same time. The utility of the Exclusive state has been shown by illustrating the amount of data often shared by zero or one processor. Eventually, the Owned and Forward states may be useful for decreasing the overall traffic

- Augmenting the number of controllers inevitably increases the network traffic as they will have to communicate. The MI protocol example has shown a moderate amount of coherence messages compared to other more sophisticated protocols. Moreover, the Hammer protocol has shown the significant decrease of coherence traffic that can be made by merging cache controllers together and having a unique controller managing a larger cache space, despite a possible increase of the controller complexity.

- A design bottleneck situated somewhere in the cache hierarchy can make the coherence protocol behave very badly. Indeed, as it was shown with the three-level MESI protocol, its very small caches triggered untimely evictions that were propagated up to the last level cache, making a huge communication overhead. This was also shown for the cache block size.

- Keeping or not the sharers list in the directory requires making change in the way states that are attributed according the number of sharers (exclusivity). The MESI and Hammer protocols have shown two implementations that were transferring exclusivity between the cache controllers and managed to avoid explicit *PutS* evictions, reducing the amount of messages across the network.

Designing a cache coherence protocol therefore needs to take into consideration the typical sharing patterns that can happen in multiprocessing workloads (random, read-only, migratory and producer/consumer [3]) as well as the complete memory hierarchy aspects on which it will be running. Several trade-offs have been analysed concerning the directory state as several possibilities showed up in the simulated protocols, and confirmed that workload-driven evaluation is an efficient way to assess the performance of such designs. Indeed, many conclusions could only be made by observing the running results.

# Chapter 7

# Conclusion

This master's thesis proposed an introduction to shared-memory multiprocessing and cache coherence to readers having only basic knowledges of computer architecture. The way components interconnect and the main design challenges were briefly introduced. A detailed study of cache coherence and coherence protocols was then proposed, by analysing in details the typical logical implementations of common protocol families found on the market : snooping and directory-based protocols.

Over the years, directory-based protocols have become the state-of-art for multiprocessors from moderate to large size due to the snooping incapabilities to scale up to many processors. Mixed protocols have also appeared sometimes and recent researches have proposed a subsuming classification of these protocols : token coherence [19]. Token coherence solves the problem by assigning processors tokens, and determines their permissions based on the number of tokens they had instead of using state bits.

To support this study, some state-of-the-art protocols were simulated and compared to each other using an architectural simulator : gem5. Getting the tool work and preparing the material to obtain interesting simulations asked a certain amount of technical work: designing a Linux image disk and kernel able to run in gem5 under QEMU while using the KVM extensions to only perform the slow annotation process in the interesting parts of the programs. Fortunately, after having faced several issues, the simulation framework was got to run perfectly.

Simulations were thus run for various implemented protocols provided with the gem5 simulator under several variations. The slow simulation time during annotation suggested distributing the different workloads across multiple machines. Each simulation providing a set of hundreds statistics, a rudimentary analysis tool was written in MATLAB to highlight the most interesting data in an easy way.

The performance analysis allowed to highlight the performance improvements brought by adding states to the coherence controllers and making trade-offs on the directory structure, as well as the impact of the memory hierarchy and workloads on the design of a coherence protocol.

Unfortunately, snooping protocols could not be compared to directory protocols due to the lack of existing implementations. While the gem5 SLICC language was studied to apply retro-engineering on the simulated protocols, no new implementation was proposed. Several factors, such as time and implementation complexity, did not allow going into this way. Indeed, the analysis omitted many implementation details, and DMAs, to make the discussion simple. Designing a state-of-the art protocol in gem5 capable of competing with the existing implementations would have required more technical knowledge and time. However, snooping protocols are no more state-of-the art protocols and the interest in simulating them is therefore reduced.

In conclusion, this work is aimed at providing a detailed summary of the main concepts of cache coherence, their implementation into gem5, as well as a methodology to analyse their performance in terms of latency, memory accesses and bandwidth usage.

## Further work

The cache coherence domain is really vast and, while this master's thesis introduced the main concepts and analysis methods, there are several ways yet to explore :

- Power consumption : Hardware designers always look for improving the Performance-Power-Area (PPA). In this master's thesis, only the performance aspects were studied, and even if the network traffic gives an insight on the magnitude of power consumption, it was not discussed here. Coherence controllers are implemented with finite state machines, which are easily modelled, and memory accesses can be pondered with datasheet information. However, to be able to discuss the power consumption of coherence protocols, interconnection networks and their implementations should be studied in details in order to model their power consumption, as they also are key actors in the protocol operation.
- Token coherence : Token coherence is a recent abstraction of current protocol families proposed by [19], in which states are replaced by tokens that are transferred from controller to controller. In this master's thesis, the impact of such a redesign was not studied, first to focus on the numerous main concepts and, second, because those designs still have no or rare commercial applications. Analysing the gain that could be obtained from implementing token coherence would be very interesting.
- Snooping protocols : The previous chapters have shown that snooping protocols are no more widely spread due to their non-ability to scale to numerous CPUs. However, their performance on small multiprocessors have not been studied and therefore it has not been proven that this protocol family are worth being forgotten.

# Bibliography


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

[2] R.J. Baron and L. Higbie. *Computer Architecture*. Addison-Wesley series in electrical and computer engineering. Addison-Wesley Publishing Company, 1992.

[3] Nick Barrow-Williams, Christian Fensch, and Simon Moore. A communication characterisation of splash-2 and parsec. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 86–97. IEEE, 2009.

[4] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[5] Christian Bienia, Sanjeev Kumar, and Kai Li. Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 47–56. IEEE, 2008.

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[8] Nathan L Binkert, Ronald G Dreslinski, Lisa R Hsu, Kevin T Lim, Ali G Saidi, and Steven K Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, (4):52–60, 2006.

[9] Alan Charlesworth. Starfire: extending the smp envelope. *Micro, IEEE*, 18(1):39–49, 1998.

[10] CCIX Consortium. Ccix : Cache coherent interconnect for accelerators. http://www.ccixconsortium.com/, 2016. Online, accessed May 24, 2016.

[11] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[12] Discussion for the users of the gem5 simulator. Using ruby with arm fs. http://comments.gmane.org/gmane.comp.emulators.m5.users/14858, 2013. Online, accessed February 15, 2016.

[13] gem5 documentation. Ruby - gem5. http://www.m5sim.org/Ruby, 2016. Online, accessed February 8, 2016.

[14] gem5 Review Board. x86: kvm: Fix the kvm cpu in se and fs on intel cpus. http://reviews.gem5.org/r/2557, 2014. Online, accessed March 2, 2016.

[15] gem5 Review Board. cpu, kvm: fix mmio handling. http://reviews.gem5.org/r/2774/, 2015. Online, accessed March 2, 2016.

[16] gem5 Wiki. Status matrix. http://www.m5sim.org/Status_Matrix, 2015. Online, accessed March 2, 2016.

[17] D.M. Harris and S.L. Harris. *Digital Design and Computer Architecture*. Morgan Kaufmann. Morgan Kaufmann, 2013.

[18] ARM Limited. Streamline for gem5. https://www.arm.com/products/tools/streamline-for-gem5.php, 2013. Online, accessed February 15, 2016.

[19] Milo MK Martin, Mark D Hill, and David A Wood. Token coherence: Decoupling performance and correctness. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 182–193. IEEE, 2003.

[20] Milo MK Martin, Daniel J Sorin, Anatassia Ailamaki, Alaa R Alameldeen, Ross M Dickson, Carl J Mauer, Kevin E Moore, Manoj Plakal, Mark D Hill, and David A Wood. Timestamp snooping: an approach for extending smps. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 25–36. ACM, 2000.

[21] Milo MK Martin, Daniel J Sorin, Bradford M Beckmann, Michael R Marty, Min Xu, Alaa R Alameldeen, Kevin E Moore, Mark D Hill, and David A Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[22] S. Mueller. *Upgrading and Repairing PCs*. Upgrading and Repairing. Pearson Education, 2015.

[23] D.J. Sorin, M.D. Hill, and D.A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture Series. Morgan & Claypool Publishers, 2011.

[24] Paul Sweazey and Alan Jay Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. In *ACM SIGARCH Computer Architecture News*, volume 14, pages 414–423. IEEE Computer Society Press, 1986.

[25] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *ACM SIGARCH computer architecture news*, volume 23, pages 24–36. ACM, 1995.

# Appendix A

# Workbench configuration

This Appendix summarizes the steps that need to be followed to get a working simulation environment under gem5. Some steps are completely automated, and the required scripts are hosted online, making this short Appendix exhaustive. For this work purposes, a specific gem5 repository has to be maintained to include patch that are not merged yet in the main development branch.

## A.1 Installing gem5

The gem5 installation was done on Fedora 23. It should be reproducible on an RHEL-based Linux distribution by following the steps behind:

1. Install dependency packages:

```
dnf install git scons swig gcc-c++ protobuf-devel python-devel gperftools-
    devel m4
```

2. Clone the custom git repository:

```
git clone https://github.com/anthonygego/gem5.git
```

3. Make an optimized build of gem5 via the scons tool:

```
scons build/<CONFIG>/gem5.opt -j<M>
```

   where `<CONFIG>` is the gem5 build configuration available in `build_opts` folder specifying the ISA and the coherence protocol, and `<M>` is the number of threads used for compilation.

## A.2 Compiling the x86 KVM-enabled gem5 Linux kernel

A Linux kernel from stable branch 3.4.x was used to run the workloads. It can be compiled by following the steps behind:

1. Clone the disk image files git repository:

```
git clone https://github.com/anthonygego/gem5-parsec3.git
```

2. Download and extract the Linux kernel 3.4.x source code:

```
wget https://cdn.kernel.org/pub/linux/kernel/v3.x/linux-3.4.112.tar.xz
tar xf linux-3.4.112.tar.xz
```

3. Copy the configuration file to the source folder and make the build/cross-compile for an x86 machine:

```
cp gem5-parsec3/kernel/x86_64_smp_kvm_gem5.config linux-3.4.112/.config
cd linux-3.4.112 && make -j<M>
```

   where <M> is the number of threads used for compilation.

4. Files `vmlinux` and `arch/x86/boot/bzImage` include the default ramdisk and are respectively used by gem5 and QEMU for loading the kernel.

## A.3 Making the PARSEC image disk

The PARSEC image disk can be produced by following the steps behind:

1. Install QEMU-KVM:

```
dnf install qemu-kvm qemu-img
```

2. Create a qemu disk image for the OS files and swap. A size of 12GB is sufficient:

```
qemu-img create -f raw linux-x86.img 12G
```

3. Donwload the Debian Wheezy netinstall kernel and ramdisk:

```
wget http://ftp.debian.org/debian/dists/wheezy/main/installer-amd64/current/
    images/netboot/debian-installer/amd64/linux
wget http://ftp.debian.org/debian/dists/wheezy/main/installer-amd64/current/
    images/netboot/debian-installer/amd64/initrd.gz
```

4. Boot the downloaded kernel (not the gem5 kernel) and ramdisk using a default PC and serial output:

```
qemu-kvm -M q35 -kernel linux -initrd initrd.gz -hda disks/linux-x86.img -m 2
    G -nographic -append "auto=true root=/dev/sda1 console=ttyS0 hostname=
    debian domain= url=<URL>"
```

   where <URL> is the HTTP link (netinstaller only support this) to the `debian-preseed` file. HTTPS is not supported neither. Uploading the file content on Pastebin and using the raw URL is a quick workaround.

   This will create the base image automatically with `root/root` and `tux/tux` sets of user/passwords.

5. Once the installation done, reboot QEMU with the freshly built kernel in Section A.2 and no ramdisk.

```
qemu-kvm -M q35 -kernel binaries/bzImage -hda disks/linux-x86.img -m 2G -
    nographic -append "root=/dev/sda1 console=ttyS0"
```

6. On the target machine as root, clone the disk image repository:

```
git clone https://github.com/anthonygego/gem5-parsec3.git
```

7. Launch the setup script located in the `disk` folder:

```
cd gem5-parsec3/disk
./setup.sh
```

This will automatically install PARSEC dependencies, build and copy the m5 executable in the `/sbin` folder, copy the new `init` startup file and build the whole PARSEC suite.

8. When ready to launch on gem5, switch between the startup files:

```
mv /etc/init.d/rcS /etc/init.d/rcS.orig
mv /etc/init.d/rcS.gem5 /etc/init.d/rcS
```

## A.4 Mounting the disk file-systems

In case something gone wrong, the generated disk can be mounted and the file-systems can be accessed using `qemu-nbd` tool.

1. Activate the kernel module as `root` user:

```
modprobe nbd max_part=8
```

2. Link the disk image to the virtual device

```
qemu-nbd --connect /dev/nbd<X> --format raw <FILE>
```

where `<X>` is the device identifier (0-16) and `<FILE>` is the disk image file.

3. The file-systems can now me mounted as usual file-systems:

```
mount /dev/nbd<X>p<Y> <MOUNT_PT>
```

where `<X>` is the device identifier (0-16), `<Y>` the partition identifier, and `<MOUNT_PT>` the mount point.

4. The disk image can be unlink from the virtual device:

```
qemu-nbd --disconnect /dev/nbd<X> <FILE>
```

where `<X>` is the device identifier (0-16).

## A.5 Running a full-system gem5 simulation with the disk image

Using the previously built kernel and disk image, the workloads can be launched by following the steps behind:

1. Create a folder with the following structure:

```
<IMG_PATH>/
 + binaries/
       + vmlinux
       + bzImage
 + disks/
       + linux-x86.img
```

2. Export the `M5_PATH` environment variable:

```
export M5_PATH=<IMG_PATH>
```

   where `<IMG_PATH>` is the path to the previously created folder.

3. Create a `myscript` script file, containing, for example:

```
#!/bin/bash
cd /root/parsec-3.0
source env.sh
parsecmgmt -a run -p blackscholes -c gcc-hooks -i simsmall
/sbin/m5 exit
```

4. Eventually, launch a gem5 full-system simulation with current Ruby configuration by typing:

```
./build/X86/gem5.opt --outdir=<OUTDIR> -e --stderr-file=<OUTDIR>/stderr.txt -
    r --stdout-file=<OUTDIR>/stdout.txt configs/example/fs.py --mem-size 1GB
    --script myscript --num-cpus 4 --caches --ruby
```

   where `<OUTDIR>` is the chosen output folder for the simulation files.

# Appendix B

# Simulated protocols tables

A detailed documentation appendix in paper format should have been made available alongside this master's thesis. If this is not the case, the complete protocols documentation can be found at :

- Either by cloning this Git repository :

  ```
  git clone https://github.com/anthonygego/gem5-slicc-latex
  ```

  The documentation files are available in HTML format, with an `index.html` file in each folder, alongside with a tool to generate automatically all the tables in LaTeX format. To run this tool:

  ```
  python gen_latex_table.py
  ```

- Either by cloning the gem5 Mercurial repository :

  ```
  hg clone http://repo.gem5.org/gem5/
  ```

  To generate the HTML documentation, gem5 must be compiled for the different protocols with the `HTML_SLICC=True` parameter, which will generate a `mem/protocol` documentation folder into the build dir.

# Appendix C

# Simulation distribution and analysis

Several scripts and tools had to be written to carry out the benchmark simulations in an easy fashion.

- The scripts made for distributing the workloads across several machines as well as the simulation statistics set are available by cloning the following Git repository:

  ```
  git clone https://github.com/anthonygego/gem5-ruby-sim
  ```

  The distribution scripts push the simulation results on a common Git repository using the following folder structure :

  ```
  <REPO_PATH>/
      + <benchmark1>/
          + <protocol1>/
              + <variation_type1>/
                      + <variation1>
                      + <variation2>
                      ...
                  + <variation_type2>/
                      ...
              + <protocol2>/
                  ...
  ```

  where `<benchmarkX>` corresponds to the executed benchmark, `<protocolX>` to the simulated Ruby hierarchy, `<variation_typeX>` to the hierarchy parameter to vary and `<variationX>` to the actual value of the parameter variation.

- The analysis tool, written in MATLAB, is available by cloning the following Git repository, and using the API as described in the `make_graphs.m` file:

  ```
  git clone https://github.com/anthonygego/gem5-ruby-analysis
  ```

  This analysis tool is made to work with the folder structure described above.

# List of Figures


2.1 Direct mapped cache with 8 sets . . . 4
2.2 Content of a fully associative cache with 8 blocks . . . 5
2.3 2-way associative cache with 8 blocks . . . 5
2.4 Direct mapped cache with 2 sets and 4-word blocks . . . 6
2.5 Miss rate with varying parameters on SPEC benchmark. From [17] . . . 7

3.1 Shared bus architecture . . . 9
3.2 Crossbar-based topologies . . . 9
3.3 Mesh topologies . . . 10
3.4 Common shared-memory hierarchies. Adapted from [11] . . . 11

4.1 Reference system model used . . . 13
4.2 Example of incoherence in a multiprocessor system . . . 14
4.3 Coherence controllers . . . 15
4.4 Transitions between stable states at cache controller. From [23] . . . 16
4.5 Venn diagram of the MOESI states . . . 18
4.6 MSI snooping protocol transitions. From [23] . . . 21
4.7 MESI snooping protocol transitions. From [23] . . . 24
4.8 MOSI snooping protocol transitions. From [23] . . . 26
4.9 An atomic bus . . . 27
4.10 A pipelined bus . . . 27
4.11 A split-transaction bus . . . 27
4.12 Directory entry for a block in a N-nodes system . . . 31
4.13 Avoiding deadlocks in directory protocols . . . 31
4.14 MSI directory protocol transitions. From [23] . . . 32
4.15 MESI directory protocol transitions. From [23] . . . 34
4.16 MOSI directory protocol transitions. From [23] . . . 36
4.17 Possible directory representations . . . 38
4.18 System model with distributed directories . . . 39
4.19 Multiple multi-core chips system . . . 41

5.1 Speed/accuracy trade offs in gem5. From [7] . . . 43
5.2 Ruby memory model. From [13] . . . 43
5.3 Ruby memory model. From [13] . . . 44
5.4 Miss rate and shared writes ratio for SPLASH2 benchmarks. From [5] . . . 50
5.5 Miss rate and shared writes ratio for PARSEC3 benchmarks. From [5] . . . 51

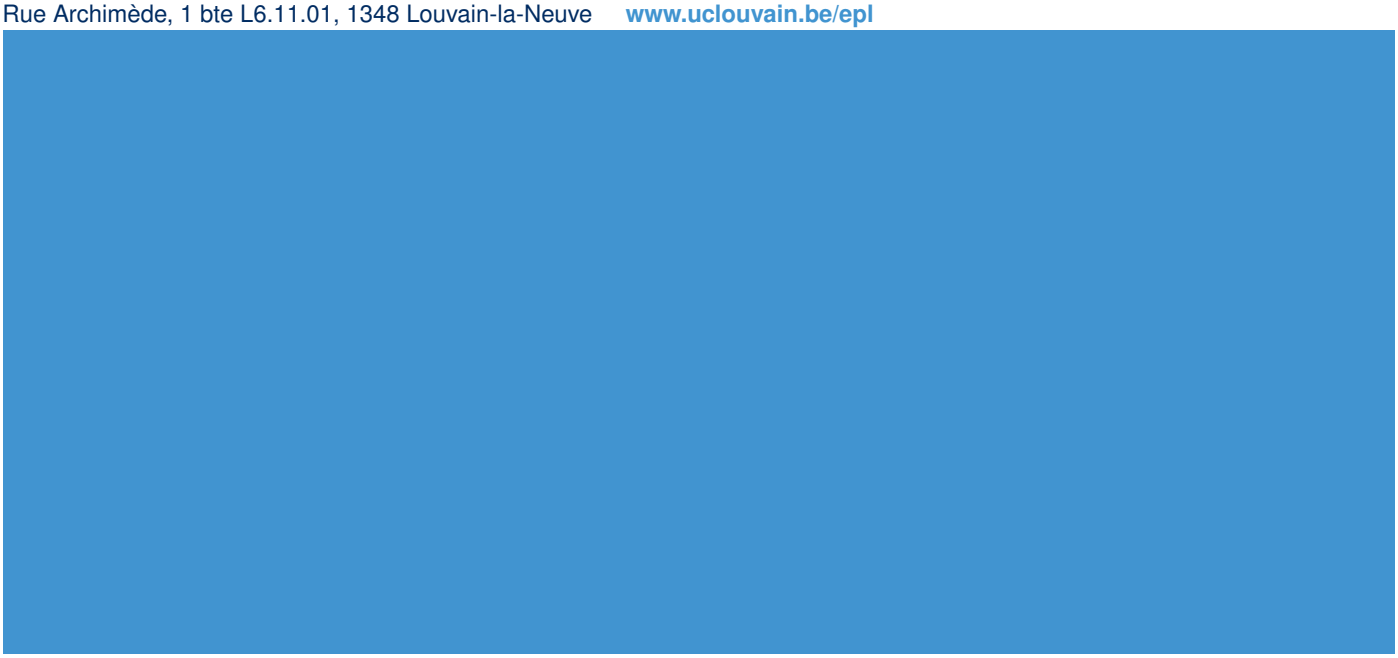
6.1 MI protocol based memory hierarchy and defaults parameters . . . 56
6.2 Two-Level MESI protocol based memory hierarchy and defaults parameters . . . 57
6.3 Three-Level MESI protocol based memory hierarchy and defaults parameters . . . 57
6.4 Two-Level MOESI protocol based memory hierarchy and defaults parameters . . . 58
6.5 AMD Hammer MOESI protocol based memory hierarchy and defaults parameters 59

6.6 Time spent in the benchmarks ROI, for default parameters . . . . . 60
6.7 Bytes written and read from memory (zoomed on right) . . . . . 60
6.8 Network traffic in message count and byte count . . . . . 61
6.9 Time spent in the ROI, for varying cache block size . . . . . 62
6.10 Time spent in the ROI for *radix* benchmark, for L1 parameters . . . . . 62
6.11 Time spent in the ROI for *radix* benchmark, for L2 parameters . . . . . 63
6.12 Time spent in the ROI for *fft* and *raytrace* benchmarks, for directory variations . . 63
6.13 MI directory events during executions of benchmarks . . . . . 64
6.14 MI L1 controller events during executions of benchmarks . . . . . 65
6.15 MI network traffic for several benchmarks in default conditions . . . . . 65
6.16 Two-level MESI directory events during executions of benchmarks . . . . . 66
6.17 Two-level L1 cache events during executions of benchmarks . . . . . 66
6.18 Two-level MESI L2 cache events during executions of benchmarks . . . . . 67
6.19 Two-Level MESI L2/memory interactions for varying L2 size *fmm* benchmark . . 67
6.20 Two-Level MESI network traffic for several benchmarks . . . . . 68
6.21 Three-level MESI directory events during executions of benchmarks . . . . . 68
6.22 Three-level MESI L0 cache controller events during executions of benchmarks . . 69
6.23 Three-level MESI L1 cache controller events during executions of benchmarks . . 69
6.24 Three-level MESI L2 cache events during executions of benchmarks . . . . . 70
6.25 Three-Level MESI network traffic for several benchmarks . . . . . 70
6.26 Two-level MOESI directory events during executions of benchmarks . . . . . 71
6.27 Two-level MOESI L1 cache events during executions of benchmarks . . . . . 71
6.28 Two-level MOESI L1 cache blocks concerned by the Owned state . . . . . 72
6.29 Two-level MOESI L2 cache events during executions of benchmarks . . . . . 72
6.30 Two-Level MOESI network message count for several benchmarks . . . . . 73
6.31 Hammer directory events during executions of benchmarks . . . . . 74
6.32 Hammer L1/L2 cache events during executions of benchmarks . . . . . 74
6.33 Hammer network message count for several benchmarks . . . . . 75

# List of Tables


4.1 Cache controller specifications. Adapted from [23] . . . . . 16
4.2 Memory controller specifications. Adapted from [23] . . . . . 16
4.3 Common transactions. Adapted from [23] . . . . . 19
4.4 Common processor requests to cache controller. Adapted from [23] . . . . . 19
4.5 MSI snooping (atomic req./trans.) cache controller. Adapted from [23] . . . . . 22
4.6 MSI snooping (atomic req./trans.) memory controller. Adapted from [23] . . . . 22
4.7 MSI snooping (non-atomic requests) cache controller. Adapted from [23] . . . . . 23
4.8 MSI snooping (non-atomic requests) memory controller. Adapted from [23] . . . 23
4.9 MESI snooping cache controller. Adapted from [23] . . . . . 25
4.10 MESI snooping memory controller. Adapted from [23] . . . . . 25
4.11 MOSI snooping cache controller. Adapted from [23] . . . . . 26
4.12 MOSI snooping memory controller. Adapted from [23] . . . . . 26
4.13 Split transaction bus MSI snooping cache controller. Adapted from [23] . . . . . 28
4.14 Split transaction bus MSI memory controller. Adapted from [23] . . . . . 28
4.15 Non-stalling split transaction MSI snooping cache controller. Adapted from [23] 29
4.16 Non-stalling split transaction MSI memory controller. Adapted from [23] . . . . . 29
4.17 MSI directory protocol cache controller. Adapted from [23] . . . . . 33
4.18 MSI directory protocol cache controller. Adapted from [23] . . . . . 33
4.19 MESI directory protocol cache controller. Adapted from [23] . . . . . 35
4.20 MESI directory protocol cache controller. Adapted from [23] . . . . . 35
4.21 MOSI directory protocol cache controller. Adapted from [23] . . . . . 37
4.22 MOSI directory protocol cache controller. Adapted from [23] . . . . . 37

5.1 SPLASH2 basic workload characteristics. From [25] . . . . . 49
5.2 PARSEC3 basic workload characteristics. From [6] . . . . . 50
5.3 Available gem5 features for implemented ISAs . . . . . 51
5.4 Required kernel configuration options for gem5 emulated system . . . . . 53

6.1 Basic characteristics of the simulation environment . . . . . 55

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**