IMPROVING THE SIMULATION ENVIRONMENT FOR COMPUTER ARCHITECTURE

A Thesis

by

ALBERTO JAVIER NARANJO CARMONA

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee, Paul V. Gratz
Committee Members, Jiang Hu
Rabinarayan Mahapatra
Head of Department, Chanan Singh

December 2014

Major Subject: Computer Engineering

Copyright 2014 Alberto Javier Naranjo Carmona

ABSTRACT

This work presents the efforts to improve the simulation environment for computer architecture research through two major contributions: The addition of a three level cache hierarchy and implementation of a statistical sampling simulation framework.

Full-system and micro-architectural simulation are the primary and most reliable research tools that the computer architecture community has. However, keeping the simulator up to date with the latest industry products is a challenging task, causing a growing time gap between the release of new commercial products and the implementation of their models in the simulators. Another problem architects have to deal with is the performance gap; the time spent on simulating one instruction is several orders of magnitude bigger than the time the real hardware takes to execute the same instruction. This leads to prohibitively long simulation times that, due to the always efficiency-focused industry trend, is also to be increased. As processors get more complex, so do the simulators. The performance improvement achieved by real hardware changes is too small compared to the overhead induced into the simulator while trying to replicate those same changes.

Although a third level (L3) cache hierarchy is a common feature in current processors and its benefits in performance have been known for decades, currently, it is not supported in most full-system simulators. A modern full system simulator was extended to include a third level cache and experiments show that for the PARSEC benchmarks, the performance of the system with L3 is $\approx 30\%$ better than the baseline.

On the other hand the implementation of statistical sampling simulation allows

a greater improvement in simulation performance while statistics theory guarantees that the subset of instructions executed are a representative sample of the benchmark behaviour. The experiments show a measured CPI error of less than 2.5% while achieving simulation time speed-ups of around 3X.

DEDICATION

To all my family and Miriam.

Without your support I wouldn't have done it, thanks for everything.

ACKNOWLEDGEMENTS

Special thanks to Dr Paul Gratz, for your patience and always helpful comments. Thanks to all the people in the CAMSIN group, I learned so much next to you all. This work was funded by the Consejo Nacional de Ciencia y Tecnología (CONACYT) - Mexican Council of Science and Technology.

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iv

ACKNOWLEDGEMENTS v

TABLE OF CONTENTS vi

LIST OF FIGURES viii

LIST OF TABLES x

1. INTRODUCTION 1

1.1 Thesis statement 3

2. BACKGROUND: CACHE MEMORIES AND STATISTICAL SAMPLING 5

2.1 Cache memories and coherence 5
2.1.1 Organization and policies of cache memories 6
2.1.2 Memory hierarchy and its effects in CPU performance 11
2.1.3 Classic coherence problem 14
2.1.4 Cache coherence protocols 16
2.1.5 The MESI protocol 21
2.1.6 Atomic coherence and race conditions 24
2.1.7 Survey on cache coherence verification techniques 27
2.2 Statistical sampling and confidence intervals 34
2.2.1 Basic properties of confidence intervals 35
2.2.2 Confidence intervals for unknown mean and unknown standard deviation 38

3. IMPLEMENTATION OF A THREE-LEVEL CACHE HIERARCHY IN GEM5 40

3.1 Original two-level cache hierarchy in gem5 41
3.1.1 Micro-architectural model 41
3.1.2 FSM in L1 43

3.1.3 FSM in L2 49
3.1.4 Support to atomic operations 53
3.1.5 Retaking the concurrent store problem 54
3.2 Extension to three cache levels 56
3.2.1 New memory hierarchy 57
3.2.2 New FSM in L1 60
3.2.3 New FSM in L2 65
3.2.4 Locks of states 78
3.2.5 Handling atomic operations with three cache levels 81
3.2.6 Verification of the new cache hierarchy 82
3.2.7 Experiment design and performance results 85

4. STATISTICAL SAMPLING SIMULATION IN GEM5 89

4.1 Survey on sampling simulation techniques 91
4.1.1 Warm-up techniques 91
4.1.2 Sampling simulation for single-threaded programs 93
4.1.3 Sampling simulation for multi-threaded programs 97
4.2 Potential speed-up in gem5 101
4.3 Switching CPU models 107
4.4 Optimal period and sample size 109
4.5 Sampling process 114
4.6 Results 116

5. CONCLUSIONS 120

REFERENCES 122

LIST OF FIGURES

FIGURE Page

2.1 Example of block cache mapping for different associativities. 8

2.2 Classic coherence problem. 15

2.3 State diagram of the MESI protocol. 24

2.4 Example of the operation of the MESI protocol between two caches. 25

2.5 Probability distribution of observing a value greater than $z_{\alpha/2}$ 36

3.1 Original two level cache hierarchy with private L1 and shared L2. . . 43

3.2 FSM describing the coherence protocol originally implemented in L1. 46

3.3 FSM describing the coherence protocol originally implemented in L2. 50

3.4 Example of concurrent store requests. 55

3.5 Proposed three level cache hierarchy. 59

3.6 Proposed state diagram for L1. 64

3.7 Proposed state diagram for L2. 76

3.8 Communication example between L1 and L2 with no race conditions. 79

3.9 Communication example between L1 and L2 with race conditions. . . 81

3.10 Algorithm used to debug and verify the new coherence protocol. . . . 84

3.11 Performance improvement with the L3 cache hierarchy. 86

3.12 Reduction of accesses in LLC with the L3 cache hierarchy. 87

3.13 Performance comparison of a system with or without prefetcher. . . . 88

4.1 Sampling techniques for single-threaded programs. 96

4.2 Summary of all possible combination of models in gem5. 103

4.3 Potential speed-up in the Parsec suite. 106

4.4 Process of switching CPU models. 108

4.5 CPI of blackscholes throughout the ROI. 110

4.6 CPI of vips throughout the ROI. 110

4.7 CPI percentage error for W when U=32k. 111

4.8 CPI percentage error for W when U=64k. 112

4.9 CPI percentage error for W when U=128k. 112

4.10 % CPI error, W=512k. 113

4.11 Variation coefficient, W=512k. 113

4.12 Sampled CPI and interval of ± 5% of the pure detailed CPI 117

4.13 Percentage CPI error and confidence intervals with confidence level of 99% 117

4.14 Speed-up of sampling compared with the pure timing and detailed simulations 118

LIST OF TABLES

TABLE Page

2.1 Description of states in the MESI protocol. 22

2.2 Description of events in the MESI protocol. 23

2.3 Description of actions in the MESI protocol. 23

3.1 Definition of states originally implemented in L1. 47

3.2 Definition of events originally implemented in L1. 48

3.3 Definition of states originally implemented in L2. 51

3.4 Definition of events originally implemented in L2. 52

3.5 Definition of the proposed events for L1. 61

3.6 Definition of the proposed states for L1. 62

3.7 Definition of the proposed states for L2. 66

3.8 Definition of the proposed events for L2. 77

3.9 Specifications for the baseline and proposed hierarchy for the experiment. 85

4.1 Summary of the 1st sampling round 115

4.2 Summary of the 2nd sampling round 115

4.3 Summary of the 3rd sampling round 116

4.4 Percentage of instructions spent in each stage 119

4.5 Recommended sampling period for W=256000 and U=64000 instructions 119

1. INTRODUCTION

Due to the high complexity of modern microprocessors, hardware prototyping is infeasible. The best tools researchers in computer architecture can rely on are simulators that aim to replicate the structure and system performance during the design process. However, modifying and working with the simulators require a deep understanding of the most popular micro-architecture techniques and how they affect other elements inside the system.

Moore's law[25], which allows us to double the amount of switching elements per unit area in a given period, as well as the non-ending trend to find ways to improve performance, have pushed industry to release new upgraded versions of their products with months of difference from the previous ones. Unfortunately, the upgrades are not directly transferred to the simulators field, where researchers expect to have reliable simulators with behaviour close to that seen in real current processors. Implementing those changes takes time and is even harder because industry may not publish in detail their improvements.

All this has led to a huge architectural gap between simulators and real processors. As an example, in order to address the growing disparity of speed between CPU and memory outside the chip known as the ”memory wall”, computer architects implemented cache memories which are smaller but faster memories inside the CPU chip. Adding more levels of cache memories to the system was a relatively straight forward conclusion; the first processors with three levels (L3) of cache memory appeared in the market in 2001. Up to the date of this writing, the stable version of popular full-system simulators like gem5[5], does not support a memory hierarchy with L3. As a result, there is a difference of 13 years between the architecture implementation

of the simulators and the real CPU's.

As simulators try to approach the state of current processors, they get increasingly more complex. Even running over the most recent processors, the simulated hardware is several orders of magnitude slower than real hardware. Full system simulation (including many CPU's, peripherals and other system components) increases the slowdown up to a factor of 10 to 100. In other words, multi-processor simulation could be a million times slower than real hardware[40]. Furthermore, the benchmarks used to test multi-threaded applications are often longer than their single-threaded counterparts. This speed difference leads to prohibitively long run times (months or even years) for simulating complete benchmark application.

Several approaches to this problem have been proposed[37]. One is to use abbreviated instruction execution streams of benchmarks, but studies concluded that abbreviated execution streams may fail to capture the global variations in program behaviour and performance[21]. Others require a previous analysis of the benchmark trace in order to find repetitive and representative instructions patterns and only execute them once[34]. A different technique runs an initial functional simulation and creates many checkpoints, later, it restores them just to run few instructions and then kill the simulation[41, 43].

Another solution is statistical sampling simulation[42] which uses two different CPU models, one slow and cycle-detailed and other fast and functional. Functional simulators just interpret or execute the instructions of a program. On the other hand, cycle-detailed simulations model the micro-architecture of a design and are used to measure the number of cycles required to execute a program. The idea of sampling simulation is to fast-forward most of the instructions with the fast and functional CPU model, sample few instructions with the cycle-detailed model and then switch back to the functional model to repeat the process. Using sampling theory we can

be certain that the measured parameter is within an interval with a given confidence level.

There is no common agreement about which sampling simulation technique the best, but the involved trade-offs (accuracy, simulation time, disk usage and flexibility) indicate the choice depends on the platform, simulator and benchmarks of interest.

Sampling simulation is not implemented in gem5, to reduce the simulation time, the researcher can either fast-forward until a given instruction and continue the remaining simulation in detailed mode, or switch back and forth between the two modes and simulate exactly one half of instructions in each model. Any of this two cases does not achieve the maximum speed-up or provide statistical support about the certainty of the measured parameter. However, these already added features to the simulator and the low disk space usage requirement made of the statistical sampling simulation the most suitable technique to be implemented in gem5.

1.1 Thesis statement

The aim of this work is to improve simulation framework on two different fronts; the implementation of micro-architecture features present in modern system and the reduction of time spent in simulation. Thus, the statement of this thesis is the following: *It is possible to keep increasing the complexity of the simulators and still reduce the simulation time without significant accuracy loss.*

One of the contributions of this work is adding a third level cache to gem5 in the most detailed mode which includes creating a new data coherence protocol for chip multiprocessor (CMP) simulation. With this improvement simulations of the memory transactions throughout all the memory hierarchy resemble more closely the behaviour of current designs. In the following chapters I will explain some design, implementation and verification issues that I faced while creating the new coherence

protocol.

The last contribution of this work is the addition of support to Statistical Sampling Simulation in gem5. In the second part of this work I will explain in detail the theory behind the technique, the process of switching CPU models during run-time, implementation challenges and the results such as the percentage error or the speed-up in run time.

2. BACKGROUND: CACHE MEMORIES AND STATISTICAL SAMPLING

2.1 Cache memories and coherence

An ideal memory system is expected to have infinite capacity, infinite bandwidth, zero latency, non-volatility and zero implementation cost[33]. However, the reality is far from that idealism, the performance of memories has not scaled as fast as the processor performance resulting in one of the biggest challenges in computer architecture known as the *memory wall*.

So far, there is no material or technology capable of satisfying all the aforementioned features, but there do exist some technologies that at least have a good performance in one of those features. Magnetic hard drives offer huge non-volatile storage at low cost but they are ridiculously slow compared with the needs of the processor. The DRAM memory is faster than hard drives, offers higher bandwidth but is expensive, volatile and has less capacity. Finally the SRAM, which should be kept small in order to be as fast as the processor also it is volatile and extremely expensive. With these elements designers have created many memory hierarchies that aim to immediately supply the requested data at almost no cost, however, since all the components inside the hierarchy are not ideal, some clever management needs be done to approach the idealism.

Memory systems exploit an observed attribute of program execution called *locality of reference* which states that programs tend to work only on regions of contiguous blocks of the memory. Specifically the principle of locality can be broken into two concepts.

- *Temporal locality*: A data block accessed, it is very likely to be accessed again in the near future.

- *Spatial locality*: When a block is accessed, the contiguous blocks are very likely to be accessed in the near future.

Thus, if we want a close to ideal performance of the memory system, we better get those likely to be accessed blocks near the processor. Cache memories are small fast memories usually implemented in the same die of the processor that quickly supply all the memory requests, exploit the locality of the programs and diminishes the effects of the memory wall.

2.1.1 Organization and policies of cache memories

A *cache line* or *cache block* is a contiguous series of bytes in memory and is the basic element on which caches operate. The smallest usable block size is the natural word size of the processor because at each access the cache must supply at least that many bytes. If a given cache has block size of 16 bytes and a capacity of 512 bytes, it is composed by $512/16 = 32$ blocks. Thus, the $\log_2 16 = 4$ least significant bits of the address will be used to index to the desired byte inside the block, the remaining higher order bits locate the appropriate block in the cache memory.

At every processor's memory request the caches must quickly determine whether they contain the requested block or not, nevertheless, the look-up latency is not only related to the cache capacity, but also to its internal organization or associativity that determines how blocks are arranged in a cache that contains multiple blocks. Usually, the cache space is divided into sets of blocks where depending on the address, the block is mapped to any available location inside a particular set.

There exist several approaches that play with the number and size of sets, but the simplest one is the *direct mapped* approach that has as many sets as blocks in the cache. Consider a cache with N blocks, in this case the cache would have N sets, each one containing one block. Thus, this is a many-to-one mapping between

addresses and storage locations in the cache and a particular address can only reside in a single location in the cache. The mapping is determined by the operation $(blockaddress) MOD (\#ofblocksincache)$.

Of course, there also exists the other extreme case where for a given cache with N blocks there is only one big set containing all the N blocks. This approach is called *fully associative*, it does an any-to-any mapping between addresses and available storage locations. Any memory address can reside anywhere inside the cache and all entries must be searched to find the right one.

The last approach, *set associative*, lies between the previous two. It proposes a set size greater than one but smaller than the total number of block in the cache and does a many to few mapping. In such way, the cache entry is assigned to a specific set resulting from the operation $(blockaddress) MOD (\#ofsetsincache)$ and it can reside on any available location inside the set. If there are n blocks in a set, the cache is said to be *n-way-set asossiative*[33]. Figure 2.1 shows an example of the different approaches implemented in a cache with eight blocks.

On every cache access the address is used to identify the corresponding set, however, inside the set, the block can reside in any storage location and it wouldn't be easy to find the correct. That is why caches have on each block an additional field called *tag* that gives the block address. Hence, the tag of every cache block inside the selected set is checked to see if it matches the block address from the processor. This comparison process is done in parallel in order to save time.

If one of the comparisons succeeds, the requested block is present in the cache and a *hit* has occurred, otherwise, it is a *miss* and the requested block must be brought from lower levels of memory.

An good cache system is expected to show a miss rate close to zero, nevertheless, there are different causes for a miss and not all of them are related to the behaviour

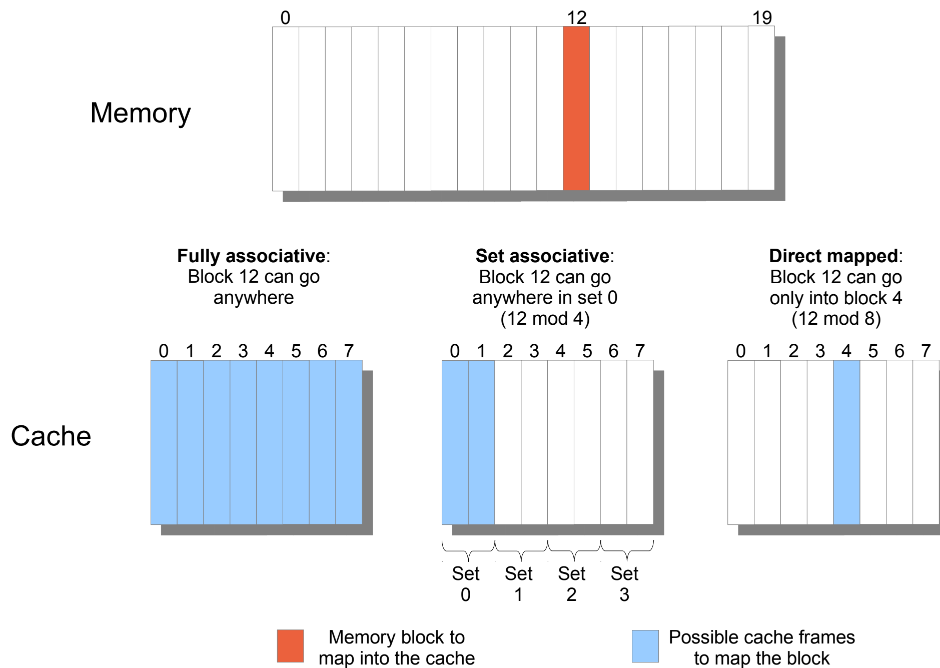

Figure 2.1: Example of block cache mapping for different associativities.

of the program or the capacity of the cache. The categories of the causes of all misses are[19]

- **Cold** or **compulsory**: Since the caches are volatile, the very first access to a block cannot be in the cache and must be brought from lower levels. These misses would occur even with infinite sized caches.

- **Capacity**: Due to insufficient capacity in the cache, the blocks are constantly discarded and later retrieved. A bigger enough cache solves the problem.

- **Conflict**: A block may be discarded and retrieved if multiple blocks map to the same set and accesses to the different blocks are intermingled. These misses are because of imperfect allocation of entries in the cache. Associativity of the

cache affects the misses by conflict, in particular a full set associative cache would eliminate them all.

- **Coherence**: This kind off misses is exclusive of multi-core systems. In order to maintain data coherence, before a particular core is allowed to modify a cache entry, the coherence mechanism must ensure that all the copies of the same block in other core's caches are invalidated, that would eventually cause a miss in the other caches. This misses have nothing to do with the cache size or organization, they are due to non idealisms in the coherence mechanism.

When a miss occurs the cache controller must guarantee that there is enough room for the new block before bringing the data from lower memory levels. However, it is very unlikely to have any available spot for the new block, thus, the controller should select a block to be replace with the new data. With the direct mapped caches this selection is trivial because there is only one block per set, but in cache organizations with more than one block per set the controller must evict the block with less chances to be used in the future. Predicting the future and chose the best block is not an easy task, but there exist several techniques like:

- **Random**: To spread allocation uniformly, candidate blocks are selected by a pseudo-random generator.

- **Least recently used**: Accesses to blocks are recorded, according to the principle of temporal locality the less likely block to be used is the one that has not been used for the longest time.

- **First in, first out**: LRU is hard to implement. This technique tries to approximate LRU by identifying the oldest block.

- **Least frequently used**: Through a period of time the least frequently used block will cause less misses.

Another important aspect of caches worth of analysis is the policies they have to handle writes. If a load/fetch operation misses, the cache controller must allocate (and replace if necessary) a new block in the cache and wait until the data is supplied by lower levels to complete the operation. However, since previous data are not needed for a write, there are two options:

- **No-write allocate**: This is actually not considered as a miss because the processor does not need to wait if the block is not present in the cache. Instead of allocating the new block in the cache the block is modified only in the lower-level memory. Not recommended in multi-processor systems because can cause race conditions when two different processors modify the same block at the same time.

- **Write-allocate**: In order to execute the write, the block must be present in the top-level cache. Hence, if it causes a miss, it is treated as a load miss, and once the block is allocated in the top-level cache, the write operation can complete. This approach is more useful in multi-processor systems because when the low level supplies the block for a write, the coherence protocol guarantees that there are no copies in other caches and it avoids race conditions.

The only presence of a cache memory implies the existence of other copies of the blocks in lower levels of the memory hierarchy. When writes to blocks are executed and the blocks in the top-level cache are updated, a mechanism must ensure that the other copies in lower levels will get updated too. There are two policies to handle this situation, one is *write-through* which simply propagates each write through the

cache to he next level. Although its implementation is straight forward, its main drawback is the amount of required bandwidth. After every store, there must be communication between different memory levels, even if that same block will be overwritten in the same operation.

The other policy is *write-back* which delays updating the copies in lower levels until the block of interest is evicted from the top level. It works under the idea that only the processor needs to get an updated copy of the block at any time, if the latest copy is kept in the highest level of memory, there is no need waste bandwidth and power on updating lower cache levels. However, its implementation is more complex; it requires a *dirty bit* that indicates when the block has been modified and the version in lower levels is out of date. When the block is to be evicted from the top level, if the dirty bit is set the data is written back to lower levels, otherwise the block is just discarded.

2.1.2 Memory hierarchy and its effects in CPU performance

One way to measure how a memory system affects the performance of the CPU is counting the number of cycles the CPU has to stop execution and wait for the memory's response. Assuming a single in-order CPU that stalls whenever a cache miss occurs and whose hits only take one cycle to complete, the amount of cycles the CPU must stall due to cache misses is

$$\begin{aligned} MemoryStallCycles &= NumberOfMisses \times MissPenalty \\ &= InstructionCount \times \frac{Misses}{Instruction} \times MissPenalty \\ &= InstructionCount \times \frac{MemoryAccesses}{Instruction} \times MissRate \\ &\quad \times MissPenalty \end{aligned} \tag{2.1}$$

The first two terms of equation 2.1 are intrinsic of the program and cannot be

changed with a better design. Nevertheless, as explained before, the miss rate changes with the size and associativity of the caches. Another option architects have to alleviate the impact of misses is to reduce the average miss penalty.

As a thumb rule, the bigger the memory, the bigger the latency. Then, it makes sense to have a small enough first level to match the clock cycle of the fast processor. However, in case of miss the cache controller must forward the request to main memory which has a huge latency. Even with low miss rate, the resulting memory stall cycles can be unacceptable. The average memory access time of the first level (experienced by the processor) is:

$$AvgMemAccessTime = HitTime_{L1} + MissRate_{L1} \times MissPenalty_{L1} \tag{2.2}$$

Where $HitTime_{L1}$ is the latency of the first level.

As a simple approximation we can say that the miss penalty of equation 2.2 seen in L1 is constant and is composed by

$$MissPenalty_{L1} = DRAMLatency \tag{2.3}$$

And equation 2.2 can be rewritten as

$$AvgMemAccessTime = HitTime_{L1} + MissRate_{L1} \times DRAMLatency \tag{2.4}$$

A simple solution is to add another level (L2) of cache between the original cache (L1) and main memory. L2 must be bigger than L1 so it can supply most of the L1's misses but smaller than main memory so it doesn't have a prohibitively latency. Since now all L1's misses are the L2's accesses and all L2's accesses correspond the

the accesses to main memory, the new L1 and L2 miss penalty are

$$MissPenalty_{L1} = HitTime_{L2} + MissRate_{L2} \times MissPenalty_{L2} \tag{2.5}$$

$$MissPenalty_{L2} = DRAMLatency \tag{2.6}$$

Thus, after substituting equations 2.5 and 2.6 in the expression of equation 2.2, the average memory access time with two cache levels is defined by

$$\begin{aligned} AvgMemAccessTime &= HitTime_{L1} + MissRate_{L1} \times MissPenalty_{L1} \\ &= HitTime_{L1} + MissRate_{L1} \\ &\quad \times (HitTime_{L2} + MissRate_{L2} \times MissPenalty_{L2}) \\ &= HitTime_{L1} + MissRate_{L1} \\ &\quad \times (HitTime_{L2} + MissRate_{L2} \times DRAMLatency) \end{aligned} \tag{2.7}$$

If we really expect to see a better memory performance, that is, the average memory access time with two cache levels (eq. 2.7) to be smaller than the access time with only one cache level (eq. 2.4), the following condition must hold

$$\begin{aligned} AvgMemAccessTime_{WithL1} &> AvgMemAccessTime_{WithL2} \\ eq.2.4 &> eq.2.7 \\ DRAMLatency &> HitTime_{L2} + MissRate_{L2} \times DRAMLatency \\ DRAMLatency\,(1 - MissRate_{L2}) &> HitTime_{L2} \end{aligned} \tag{2.8}$$

Equation 2.8 tells us that if both, L2's miss rate and hit time are kept small, it is more likely to improve the performance.

The L2 design is not trivial, in reality there is a compromise between the latency and miss rate. The same analysis applies to determine if even more cache levels would be beneficial to the performance of the system. In practice, the power and silicon area constraints stop us to have many more cache levels.

2.1.3 Classic coherence problem

In the last decade, due to several reasons, the interest of researching on techniques that exploit Thread Level Parallelism has grown among the computer architecture community; commercial multi-processor systems appeared as a consequence. Regarding cache memories, it does not make sense to have one huge shared cache in the first level because its latency would negatively affect the processor performance on every memory operation. The most common solution is to have a small and fast first level cache private for each core.

Every core runs different threads and can have its own address space[1]. However, it is also possible that more than one core operate on the same memory address, which implies that the cores share the block and keep copies from it in their individual private caches.

Caching shared data introduces a new problem, that if not handled properly, the cores may end up seeing two or more different values for the same memory location. Figure 2.2 depicts the process that leads to the *Coherence problem*. Consider a system with two processors(CPU0 & CPU1), each one with its own private L1 cache and both share a common L2 cache. At *time=0*, both cores are running their own threads and do not share any memory location. Later on at *time=1*, CPU0 decides to load a new value, so the requested block is brought from main memory, stored in L2 and the private CPU0's L1. Some moments after that, when *time=2* the process

[1]Set of memory addresses to be accessed.

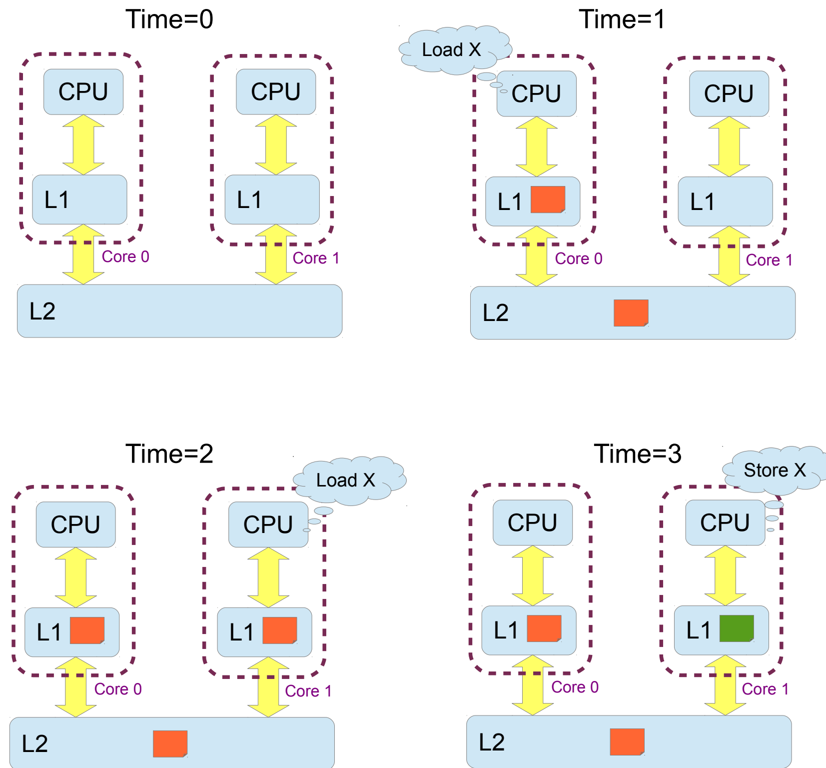

Figure 2.2: Classic coherence problem.

running in CPU1 requests a read to the same address that CPU0 did before, then the same block is copied from L2 to CPU1's L1. Finally, at *time=3* CPU1 does an store and modifies that block, at that moment both cores have different versions of the same data block. If CPU0 reads the block again, it will read the old version of the block and has no way to know that the read value is out of date. That process can keep going indefinitely and each core will read and modify its own copy of the block without being able to determine which version is the right one. This situation violates the *coherence* of the memory system because it is not clear what value can be returned after a read.

In general, a memory system is coherent if it supports:

1. **Write propagation**: Any change to the memory image made by one processor's write is made visible to all other processors in the system before any of the other processors could load or store that specific location. It must always return the latest written value on each load.

2. **Write serialization**: Two writes to the same memory location by any two processors are seen in the same order by all processors.

A coherent view of the memory is hard requirement for shared-memory multi-core systems. Without it, programs that share memory in two or more cores would behave unpredictably. Designers have developed many protocols that handle properly the coherence problem.

2.1.4 Cache coherence protocols

A cache coherence protocol is a set of rules that a memory system must follow after read/write requests in order to have coherence and consistency among multiple data copies, it must also support the principles of write propagation and serialization. To accomplish this, the controller must keep track of the state of any shared data block.

Depending on the kind of action that protocols do to make write visible to other processors, to classes of protocols exist:

- **Write-update**: These protocols are based on a broadcast write-through policy. Whenever a write in a single processor occurs, all copies of the same block in other caches must be immediately updated to the new value. This is done by broadcasting the new data into the shared bus, hence, all the caches should constantly snoop the bus to be able to detect a write and update their own copy. Write-update protocols worked well for systems with few processors,

nevertheless, as the number of processors increases the communication through a shared bus becomes inefficient to handle the excessive bandwidth demands, leading to their virtual extinction.

- **Write-invalidate**: With this approach, only a single processor is allowed to write a cache line at any time. Thus, the processor that wishes to write to a cache line must first make sure that its block is the only valid copy among all the cores. Before performing the write, the local processor must check whether or not the cache line is shared with other cores, if so, it sends out messages to invalidate other copies. Subsequent writes from the same processor are streamlined since no check for remote copies is required. Local processor must share (and invalidate if needed) the cache line upon other processor's requests. Finally, the cache block is written back when it gets evicted from the first cache level.

Because coherence protocols are essentially composed of entities (processors, caches and memory controllers) that receive requests (events) from each other and depending on the status of the requested cache line they respond in different ways, a natural way to model them is through Finite State Machines (FSM). In such model, every cache block must be in the state that defines its read/write permissions and better represents its status. After to the arrival of events it may do the transition to another state after performing some specific actions.

The complexity of the protocol increases with the level of detail required to describe the status of the cache block. For example, if the protocol requires to determine whether the block is present on the cache or not, only two states (Invalid and Valid) are needed. If a block is initially in the Invalid state (not present in the cache), a load request from the processor will make the block move to the Valid state

once it is allocated in the cache.

Furthermore, if more detail is needed to know whether other caches have copies of the same block or not, a three-states protocol (Invalid, Exclusive and Shared) is enough. Assuming the Exclusive state has read/write permission and that Shared is a read-only state, this protocol can successfully handle the classic coherence problem. If initially two processors share the same block, both copies must be in the Shared state (S,S). If one of them wishes to write, it must invalidate the other copy and move to the Exclusive state to have read/write permission (E,I). If the second processor now performs a write, the first processor should invalidate its copy and send the latest data to the second one (I,E). Finally, if the first processor wishes to read again the block, there is no need to invalidate the copy, they just share the block (S,S). Note however, that there are combination of states like (S,E) that by definition should never happen and can cause incoherent data.

In write-back systems it is useful to detect if a block has been modified, so the controller can decide whether to write it back to memory or just discard it during a block eviction. That case requires a forth state (Modified). The resulting four-state protocol, known as MESI, its widely known since it was first introduced in 1984. In the subsection 2.1.5 I will explain this protocol in detail.

Coherence protocols require a mechanism able to track the state of each active cache line, the most convenient place to store the state is in the tag array. We can argue that the dirty bit, used to determine if the block has been modified or not, actually is storing the state of the block. Protocols can have different techniques to keep track of the state of each cache block and they can be classified into two big groups: Snooping and directory-based protocols. Here I will explain the main features, advantages and drawbacks of each group.

2.1.4.1 Snooping protocols

Snooping protocols where originally conceived for systems with few processors which all their caches are connected to each other through a single shared bus. With this approach every single cache is responsible of tracking the status of all the blocks it contains and of broadcasting to other caches the new data whenever a local modification occurs. Cache controllers must constantly monitor or *snoop* the bus. When other processor broadcasts an update/invalidate message through the bus, the local processor should update/invalidate its own block if it matches with the description of the message. The bus acts as a mutex and avoids race conditions, the processor that gains access to the bus first is allowed to modify the block and other processors must update its copies immediately.

The main drawback of snooping implementations is the poor scalability to systems with many processors, specially in terms of bus bandwidth. For example, if two processors wish to do a write (even on different blocks), one will win the race and the other must wait until the broadcast of the new value is done. After that, the second processor can go ahead and access the bus. In general, if we assume that each processor generates bus transactions at a given rate, the frequency at which the bus must be snooped by other processors is directly proportional to the number of processors in the system. Since each snoop requires at least a local cache lookup, the aggregate bandwidth can quickly become prohibitive.

2.1.4.2 Directory-based protocols

Directory-based protocols are a good alternative to alleviate the bandwidth problem. Since their performance does not rely on broadcasts, they do not need a common shared structure to communicate with other components. Furthermore, that feature makes directory-based protocols more suitable to modern designs like Networks on

Chip (NoC's). Unlike snooping protocols where each cache must be aware of all the transactions to update the states, in these protocols a centralized data structure called directory tracks the state of the caches and communicates with them only when it is needed. The information in the directory resides next to each entry of the shared memory and includes the state as well as which caches have copies of the block. In a multi-core system with a shared memory the directory keeps track of the sharing state of the block by attaching to every memory entry a bit vector of size equal to the number of cores, the bits will indicate whether the caches have a copy of the block or not. The storage overhead introduced by the directory structure and bit vector may not scale gracefully for systems with large number of processors[15].

Rather than broadcasting to all the caches, directory-based protocols save bandwidth by sending unicast/multicast messages only to the sharers. On the other hand, the implementation of the directory introduces the problem of indirection. In snooping protocols when two caches needed to communicate, they only had to broadcast the bus; in directory-based protocols, the cache needs to send a message to the directory and if the later determines that it is unable to respond the request (the directory's copy might be out of date due to a write in one cache), it forwards the request to a cache capable of responding. This forced communication with the directory even when it is not really necessary increases the response latency of some memory requests. However, protocols like DiCo_CMP [32] propose some techniques to avoid indirection, reduce the cache miss access latency and reduce the network traffic.

Another inefficiency of the directory-based protocols occurs when the local cache wants to modify a block shared with many caches. Before proceeding with the store instruction, the local cache must wait for the invalidation acknowledgements from all the sharers, resulting also in higher protocol latencies.

The coherence protocol in the directory is also modelled as a FSM, of course different from the FSM in the caches, with events mainly triggered by the caches requests and states that describe the status of the blocks in the higher level. For example, one state describes that the block is only present in the directory and not in the caches, and other state describes when the block is present in only one cache and probably dirty. In the same way FMSs in top level caches must be synchronized between them, there also should be coherence between the directory and the caches. For example, it is a risk condition when according to the directory, the block is not present in the caches but in reality, it is present and probably modified in one cache.

Future CMP designs with tens or hundred of cores will be constrained by area and power, this constrains make impractical the use of a shared bus and protocols that rely on broadcasts for keeping cache coherence. Apparently, on-chip interconnection networks along with directory-based protocols will dominate in future designs[29].

2.1.5 The MESI protocol

The MESI protocol, which is named after the four states it includes, was first presented in 1984 by Papamarcos[27] in the University of Illinois[2]. Due to its simplicity and capability of being used in systems with many cores with good performance, it is usually taken as example for coherence protocols in the literature.

The four states are enough to describe the status of the cache blocks at any time. The Invalid state guarantees no presence of the block in the cache, states Exclusive and Modified guarantee exclusive ownership of the block, and finally, the Shared state guarantees presence but not exclusivity of the block, that is, the block may or may not be the only copy among all the caches. Table 2.1 describes each one of the states.

[2] For obvious reasons it it also known as the Illinois protocol.

Table 2.1: Description of states in the MESI protocol.

State	Description
I	**Invalid**: The block is not present in the cache.
S	**Shared**: The cache entry is potentially shared with one or more caches. The block is clean; it is consistent with the version stored in the directory.
E	**Exclusive**: The cache entry is only present in the local cache. The block is clean; it is consistent with the version stored in the directory.
M	**Modified**:The cache entry is only present in the local cache and it is dirty. Write-back when the block is evicted from the local cache or shared with other caches.

On the other hand, a coherence protocol must react to events that can be originated by a special condition in one of the elements of the cache hierarchy. Depending on the origin of the events, they can be classified as: local reference (due to local CPU's request), remote reference (due to other cache's request) or local capacity eviction (generated inside the local cache). Table 2.2 shows a description of all the events defined in the protocol.

As every FSM, the caches are not just passive entities that receive events and move from one state to another, they are active elements of the protocol that answer to the events depending on the state they are in. A description of the actions they do is shown in table 2.3.

It is worth noting the relationship between the actions done by one cache and the events received from others. For example, if one cache have a read miss, it will issue a GETS request to the interconnection network and other caches will receive that request as a Fwd_GETS (network read) event.
The last missing thing to complete the description of the protocol is the definition of the transitions between states. Figure 2.3 shows the state diagram of the protocol,

Table 2.2: Description of events in the MESI protocol.

Local reference	
Event	**Description**
Ifetch	Local CPU issued an Instruction Fetch request to the cache.
Load	Local CPU issued a Load request to the cache.
Store	Local CPU issued a Store request to the cache.
Remote reference	
Event	**Description**
Fwd_GET_INSTR	Instruction Fetch miss in other cache, local cache must share the block.
Fwd_GETS	Data Load miss in other cache, local cache must share the block.
Fwd_GETX	Store miss in other cache, local cache must invalidate its copy and send the block to the requester.
INV	Either the directory or other cache request the local cache to invalidate its copy of the block.
Local capacity	
Event	**Description**
L1_Replacement	There is no enough room in the local cache to allocate a new cache block, it must evict one.

Table 2.3: Description of actions in the MESI protocol.

Action	**Description**
GET_INSTR	Local cache requests an instruction to the directory.
GETS	Local cache requests a data block *without* intent to modify it, probably because of a load miss.
GETX	Local cache requests a data block *with* intent to modify it, this message implies the invalidation of other copies.
PUTX	Local cache writes the data back to the shared memory due to eviction of the block.

the arrows represent the transitions and the labels next to each arrow show the events that can trigger the transition.

In order to better understand the event-action-event relationship and the inter-

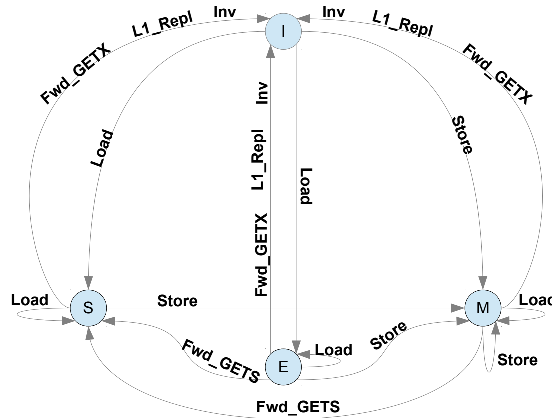

Figure 2.3: State diagram of the MESI protocol.

action of the caches in the protocol, lets consider the following example depicted in figure 2.4. Inside figure 2.4, states in red represent the current state of the cache blocks at a given time. Suppose a two-core system with two private caches and a block that is not present in either cache at time 0. At the following time step cache0 receives the Load event from the local CPU, it allocates the requested block, moves to E sends the data to the CPU. Then, at time=2, the CPU performs a store and the cache changes its state to M. When cache1 receives a Load from its local CPU at time=3, it asks cache0 to share the block and both caches change their states to S. Next, the CPU attached to cache1 does a write to the block which invalidates cache0's copy and makes cache1 move to M. Finally, at time=5 cache0 receives a store request, hence it makes cache1 invalidate its copy and send the block to cache0, which moves to M and performs the store.

2.1.6 Atomic coherence and race conditions

The coherence protocols explained in this work up to this point have assumed atomic operations, that is, no intervening operations can occur while other operation

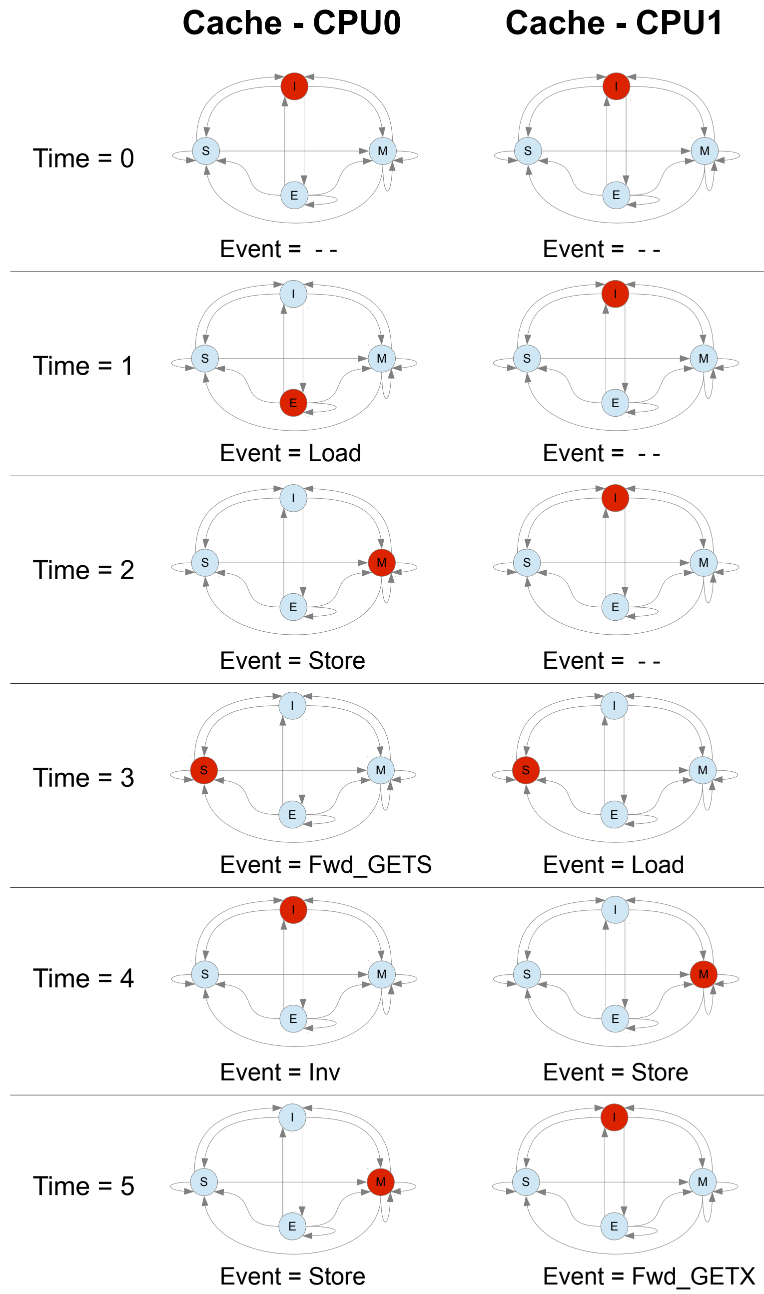

Figure 2.4: Example of the operation of the MESI protocol between two caches.

is in progress. For example at time=1 in figure 2.4 cache0 received the Load request, it allocated the block and did a transition from I to E as a single atomic action. However, the reality is not that simple and many things must happen before a transition can finish. In a more realistic exercise, what could happen after cache0 receives the Load request is:

1. Cache0 issues an GETS request to the directory and waits until the directory answers with data.

2. The directory receives the GETS request from cache0, supposing the directory does not contain the block, it sends another request to lower memory levels and waits until the block is supplied.

3. The directory receives the block from lower memory levels and forwards it to cache0.

4. Cache0 receives the requested block, finishes the transition to E and satisfies the local CPU's needs.

If in addition to all the time waiting we also consider the intrinsic network and buffer delays, the time spent from the beginning of the transition to its end is even bigger. This latency is too large to assume that no other events will be triggered before the transition's end. For example, while waiting for the data, the CPU might wish to do a write into the same block, and moments later cache1 might ask cache0 to share the block that has not been loaded and written yet.

Considering non-atomic transitions makes the protocol more real but increases its complexity because it should handle many probable race conditions. Designing for races is hard. Although states dedicated to handle race conditions may represent a considerable proportion of all states, they only represent a small fraction of all

observed transitions in commercial workloads[39]. Because of their relative in infrequency, race conditions have little impact on performance, but they impose design complexity and verification challenges.

2.1.7 Survey on cache coherence verification techniques

The coherence protocols verification became an area of interest for researchers as the current systems incorporate more cores and other components in a shared memory scheme. Early coherence protocols based on bus-snooping connected few modules through a time shared bus and hence its low complexity made the verification task relatively easy. Nevertheless, the large amount of cores present in modern CMP's along with the need of higher data bandwidth, have motivated the designers to leave aside the time-shared bus and look towards a more reliable interconnection network. Although more complex due to their intrinsic properties, directory based protocols are a better choice for current (and future) CMPs than snooping protocols.

The more modules coordinated by the protocol, the more likely it is to have corner cases as a result of a rare sequence of events, Then, it is crucial to add auxiliary states to correctly handle the corner cases and avoid the protocol to crash. Although these auxiliary states rarely are visited on normal operation, they do significantly increment the complexity of the protocol and therefore, its verification becomes a challenging problem.

The goal of verification is to ensure that a coherence protocol satisfies all the required specifications, in particular, there are three basic properties to verify that Pong *et al.*[30] define as:

- **Data consistency.** On each load, the protocol should always return the latest stored value. Consistency is enforced by allowing only one store in progress for each time at any time.

- **Complete protocol specification.** Protocol incompleteness occurs when possible events or state transitions have been omitted (i.e. a component receives a not specified message in its current state). Since those situations are not specified, the subsequent behaviour of the protocol is unpredictable.

- **Absence of deadlock and livelock.** A deadlock occurs when the protocol enters a state which does not leave because is waiting for an event that never happens. The protocol is blocked and cannot service other requests until it leaves that state. On the other hand, a livelock occurs when the protocol gets stuck in a loop of transitions without making any useful progress.

Informal protocol verification techniques are based on time-consuming and error-prone procedures. As the complexity of coherence protocols increases, it becomes harder to verify them by simply relying on human reasoning. Simulations are conceptually simple but they only guarantee that the protocol works for a particular sequence of events, they would need to indefinitely run a random sequence to completely verify the protocol.

To successfully verify systems of arbitrary complexity, the biggest issue most of verification techniques must deal with is the *state explosion problem*. In these techniques the protocol is characterized by its state and the verification is based on searching all reachable states exhaustively. Hence, the amount of memory required to manipulate the state information and the verification time grow very fast with the number of processors and the complexity of the protocol mechanisms. With the goal of reducing the size of the state space, the research community has proposed several methods that exploit different features of the cache-based systems like homogeneities, regularities and symmetries.

In the following sections I present a summary of the most relevant and accepted

coherence protocol verification techniques.

2.1.7.1 Reachability analysis and state enumeration

The global state of the system is defined as the composition of the states of all its components and the correctness of the protocol is verified on the set of reachable global states. For a given global state, reachable states are found by exhaustively exploring all the possible interactions between entities. If one state fails to preserve the correctness of the protocol it is classified as erroneous, otherwise is permissible.

Conventionally in an exhaustive search algorithm, for each state all its reachable states are added to the working list even if some of them may have been visited previously. Furthermore, a large number of previously visited states are also expanded during the state expansion procedure. This makes the state space grow exponentially with the number of components and the complexity of the protocol. A simple solution is to add a history list that contains all the previously visited states and if the current state is present on the history list it is not expanded or added to the working list.

Even with these assumptions the number of global states is still unmanageable for current coherence protocols, and on the other hand, the agreement between cache states and data copies must also be verified, which means that data values must be modelled along with state transitions.

Several variations have been proposed to overcome the inefficiency and large memory requirement of state enumeration methods, most of them focus on keeping track of only the states on the current expansion path, encoding the state information and using hash tables. However, those techniques are not totally accurate because they do not detect livelocks and several states map to the same hash value[3][30].

[3]An ideal hash function mapping each global state to a unique hash value is not practical.

Other techniques like the Murϕ Verification System[11, 10] exploit the system symmetries. Murϕ is composed by a compiler and a high-level programming language for the description of finite-state asynchronous concurrent systems and has been extensively used to verify coherence and communication protocols. A Murϕ program consists of four parts: declarations, transition rules, start state generation rules, and invariant descriptions (Boolean conditions that have to be true in every reachable state). The compiler generates a C++ program from the FSM which exhaustively generates the reachable states, checks for error conditions and deadlocks.

The Murϕ verifier works by explicitly generating all the reachable states and storing them in a hash table, it also implements some state reduction techniques such as symmetry reduction, exploitation of reversible rules, and verification of systems with varying numbers of replicated components[20]. However, the state explosion problem is still a big issue, it was shown that even for fairly small models of 3 or 4 processors the reduced state space size is above 107 states [30]. Furthermore, Murϕ does not guarantee total correctness of the protocol and their developers recommend its use only as a debugging tool.

2.1.7.2 Model checking

A temporal logic is an extension of predicate logic with additional tense operators for expressing properties evolving with time. Model checking is a formal verification technique that expresses properties of the protocol as formulas in temporal logic. In general, after construction of the state graph of the protocol model, the properties specified as temporal logic formulas are evaluated on the graph.

The strength of this technique is based on the expressiveness of temporal logic, which can handle arbitrary temporal formulas, representing both safety (data consistency) and liveness (livelock and deadlock free) properties. However, since model

checking takes the state graph as a model, it also suffers from the state space explosion problem.

Symbolic model checking is a technique to perform model checking without explicitly representing the state graph [23]. It saves great amount of memory by representing the global state graph by Ordered Binary Decision Diagrams (OBDDs); additionally, it composes finite state modules to build the transition relations among global states. As a consequence, unlike the state enumeration methods, the reachable global states are not produced one by one.

Emmerson and Sistla [13] extended the model checking technique by exploiting symmetry. Since states that are permutations of each other are lumped into a single canonical state, the the OBDD size and the state space after transformation can be significantly reduced.

2.1.7.3 Symbolic state modelling

Two states are equivalent if they are symmetrically identical in methods with symmetry extension. For example, in a system with three caches, the tuples (shared, shared, invalid) and (shared, invalid, shared) represent a similar condition of the system and hence, should be handled in the same way. In regards of verification of the system, a set of equivalent states can be replaced by one canonical state called *symbolic state*.

The symbolic state modelling searches the state space exhaustively just like in the traditional state enumeration methods. The difference is that it uses symbolic states and thus, the system is represented by a symbolic state model (SSM). The SSM method groups caches in the same state into a class and the number of caches in the class is symbolically represented by a repetition constructor, in such way, all the equivalent states are pruned out from the reachable states and the states explosion

problem is reduced. The abstraction in this model is much more powerful than the symmetric relations obtained from symmetry alone.

Pong *et al.*[29] discard redundant states under the premise that the protocol correctness in not dependent on the exact number of cached copies, symbolic states only need to keep track of whether the caches have 0, 1 or multiple copies. With this assumption the verification process is independent of the number of caches and consequently is reliable.

2.1.7.4 Dynamic verification

Dynamic or runtime verification is not a new concept and avoids the complexity of traditional formal verification techniques, such as model checking and theorem proving. It has been applied to cache coherence to detect at runtime and recover from errors caused by manufacturing faults, soft errors, and design mistakes[6]. However, the existing coherence checkers are susceptible to errors and costly to implement.

Rodrigues *et al.* present a centralized mechanism for dynamic verification of cache coherency in snoopy bus multicore systems[31]. They propose the addition of a module called Sentry Core (SC) which they claim to be fault-free. The SC has access to the shared bus, monitors all bus transactions and since it is aware of the coherence protocol, by observing the current state of the cache line it knows the next state for any cache line. They show that implementing the SC will incur into a performance degradation of less than 2% in the worst case.

Meixner and Sorin [24] detail the implementation of a framework for the cache coherency dynamic verification in the SPARCv9 architecture. They constructed the Cache Coherence checker around the notion of an epoch, which is a time interval when a processor has permission to read or read and write a given cache block. The rules they used to determine coherence violations are 1) reads and writes are

only performed during appropriate epochs, 2) read-write epochs do not overlap other epochs temporally, and 3) the data value of a block at the beginning of every epoch is equal to the data value at the end of the most recent read-write epoch.

2.1.7.5 Other techniques

As mentioned before, great part of the complexity of current coherence protocols is caused by the race conditions that they must handle. Therefore, instead of trying to improve verification techniques, Vantrease *et al.*[39] propose to make the verification feasible by simplifying the protocol and eliminating the race condition.

Mutexes are a natural way to support mutual exclusion in the coherence protocol, i.e. the block's coherence state may not be altered until the mutex has been obtained. However, obtaining access to the mutex is an operation that requires time and hence, is one of the main race condition sources. Vantrease *et al.* propose to use on-chip silicon photonics and implement very low latency mutex which will support simple atomic operations. They advocate a return to atomic protocols and show that an atomic implementation of the protocol is much simpler while imposing less than a 2% performance penalty.

The last coherence protocol verification technique covered in this document is the Random Traffic Generation. It consist on stressing the system with the constant injection of random messages and checking if the response of the system is the right one or not. Every time a new request is injected to the system its message type, expected response and maximum round trip time are registered. The tester (located in the CPU's side of the hierarchy) is constantly checking the status of all injected packets. When a response from one of the originally injected messages gets into the tester, it is checked and compared with the previously registered data. If the received data is the same as expected the transaction is said to be successful and

its data is discarded, otherwise a data consistency error is launched. Furthermore, if the tester detects that one of the injected messages has not came back and exceeds the maximum allowed round trip time, the test stops and a possible deadlock error is displayed.

Assuming a ideal random generator a random test sequence must be run indefinitely in order to enter all reachable states. Although it might be more time consuming than others, this technique is able to find coherence problems, data inconsistencies and deadlock and livelock conditions. Also, the random traffic generation is not exposed to the state explosion problem, the amount of memory and the time spent on each test increase linearly with the number of caches simulated and messages injected to the system respectively. Random traffic generators may stress, but not exhaust, potential race combinations.

2.2 Statistical sampling and confidence intervals

In statistics, the goal of sampling is to have an estimate of a population parameter without the need of measuring every element of it. The point estimators involve the use of simple data to calculate a single value and which serves as a "guess" of an unknown population parameters. Some of the most commonly used methods for point estimation include the method of moments and the median-unbiased estimator among others. The discussion and description of point estimators are beyond the scope of this thesis, however it is illustrating to compare them with the interval estimators.

In contrast to point estimation which uses only a single number, interval estimation calculates an interval of probable values of an unknown population parameter. In other words, it outputs an interval in which the parameter of interest is more likely to be and in some cases it also calculates the likelihood of the parameter to be

inside the parameter. This chapter introduces and explains the statistical principles of the confidence intervals which will serve in further chapters as the basis of the statistical sampling simulation techniques.

2.2.1 Basic properties of confidence intervals

A given confidence interval is always calculated by setting a confidence level before, which is a measurement of the degree confidence of the interval. A confidence level of 95% implies that 95% of the samples of the parameter under interest fall within the interval and only 5% of the samples would be above or below the confidence interval. In other words, the bigger the confidence level the more sure we can be that the estimated population parameter is within the interval. The most common confidence levels used in statistics applications are 90%, 95% and 99%.

Both, the confidence level and confidence interval express the accuracy of the estimation. With a high confidence level, if the resulting interval is small we can argue that the parameter estimation is fairly accurate, however if the interval is big there is uncertainty in the parameter estimation.

In order to introduce the concepts and properties of the confidence intervals lets do first two simple and somewhat unrealistic assumptions:

- The population is normally distributed.
- The population standard deviation σ is known.

Let $x_1, x_2, \ldots, x_n$ be the random samples of a population with normal distribution, mean μ and standard deviation σ. It can be shown that the sample mean $\bar{x}$ has a normal distribution with expected value μ and standard deviation $\sigma/\sqrt{n}$ [9] .

The standardization of $\bar{x}$ produces the variable

$$Z = \frac{\bar{x} - \mu}{\sigma/\sqrt{n}} \tag{2.9}$$

which has a normal distribution. If we want to have a confidence level of $C = 100(1-\alpha)\%$, then we must ensure that the standardized variable in (2.9) has C probability to happen. In other words,

$$P\left(-z_{\alpha/2} \leq \frac{\bar{x} - \mu}{\sigma\sqrt{n}} < z_{\alpha/2}\right) = 1 - \alpha \tag{2.10}$$

where $z_{\alpha/2}$ represents the point on the standard normal density curve such that the probability of observing a value greater than $z_{\alpha/2}$ is equal to α, see figure 2.5. For example, if the confidence level is 95%, $C = 0.95$, $\alpha = 0.05$ and $z_{\alpha/2} = 1.96$.

By doing some arrangements in (2.10) we get

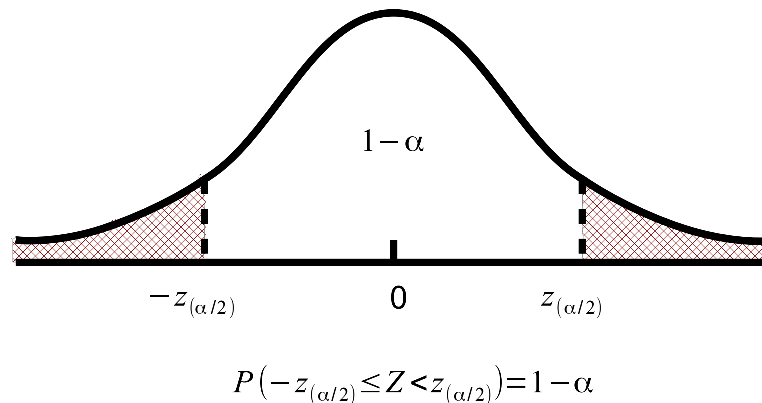

Figure 2.5: Probability distribution of observing a value greater than $z_{\alpha/2}$

$$P\left(\bar{x} - z_{\alpha/2}\frac{\sigma}{\sqrt{n}} \leq \mu < \bar{x} + z_{\alpha/2}\frac{\sigma}{\sqrt{n}}\right) = C \tag{2.11}$$

Equation (2.11) means that with a probability of C, the population mean will be within the interval defined by

$$\left(\bar{x} - z_{\alpha/2}\frac{\sigma}{\sqrt{n}}, \bar{x} + z_{\alpha/2}\frac{\sigma}{\sqrt{n}}\right) \tag{2.12}$$

From (2.12) we know that the interval has its center in $\bar{x}$ and a width of

$$w = \frac{2z_{\alpha/2}\sigma}{\sqrt{n}} \tag{2.13}$$

This implies that for a bigger confidence level (bigger $z_{\alpha/2}$) the width of the interval will also increase. There is more confidence of the mean being within a bigger interval. Actually, for the special case with a the confidence level of $C = 100\%$, the resulting interval is $(-\infty, \infty)$; even before sampling we can be 100% sure that the mean will be somewhere between $-\infty$ and ∞.

Hence, we may end up with a good confidence level but a big interval or a small interval with low confidence level, which in either case does not provide any real certainty about the estimated value μ. In fact, as far as the sample size n and the standard deviation σ keep constant, the only available choice is to play with the trade off between the confidence level and interval size and find the best possible combination. However, this solution may not solve our needs.

Fortunately, we can act in a different way, first define the desired confidence level and interval width and then figure out the sample size that meets those conditions for a given known population standard deviation. After solving for n in equation

(2.13) we get:

$$n = \left(\frac{2z_{\alpha/2}\sigma}{w} \right)^2 \tag{2.14}$$

2.2.2 Confidence intervals for unknown mean and unknown standard deviation

The previous section was based on the (not necessarily real) suppositions of a normally distributed population and a priori known standard deviation, now I present the confidence intervals for those samples that do not meet these suppositions.

Let $x_1, x_2, \ldots, x_n$ be the random samples of a population with a mean μ and finite standard deviation σ. As long as n is large enough[4],the *Central Limit Theorem* states that the distribution of the sample mean $\bar{x}$ will approach a normal distribution, regardless of the population distribution. Then we can claim that $Z = (\bar{x} - \mu)/(\sigma/\sqrt{n})$ has an approximately normal distribution resulting in:

$$P\left(-z_{\alpha/2} \leq \frac{\bar{x} - \mu}{\sigma/\sqrt{n}} < z_{\alpha/2} \right) \approx 1 - \alpha \tag{2.15}$$

One of the practical difficulties of calculating the confidence interval in this way is that σ is rarely known. In this case, the standard deviation σ is replaced by the estimated standard deviation s, which leads to the standardized variable

$$Z = \frac{\bar{x} - \mu}{s/\sqrt{n}} \tag{2.16}$$

Using s instead of σ adds some randomness to Z however, if n is big enough Z keeps the condition of having a standard normal distribution and hence, regardless of the population distribution, the confidence interval for a big sample size n and

[4] A good thumb rule to consider the sample size big enough is if $n \geq 30$ [9]

confidence level $C = 100(1 - \alpha)\%$ is

$$\bar{x} \pm z_{\alpha/2} \frac{s}{\sqrt{n}} \tag{2.17}$$

3. IMPLEMENTATION OF A THREE-LEVEL CACHE HIERARCHY IN GEM5

Even the latest stable version of gem5 only supports by default two cache levels. Adding a new level to the cache hierarchy requires a deep understanding of both, the operation of every component in the hierarchy and the interaction between elements. One component that undoubtedly defines many things (behavioural and structural-wise) in the cache system is the coherence protocol.

The RUBY memory system in gem5 allows the relatively easy design and modelling of coherence protocols. Among all the protocols included in the latest version of gem5 I chose *MESI_CMP_directory* because of its stability, and low complexity (relative low number of states and transitions).

Just like its name implies, it is an implementation of the MESI protocol. However, the coherence protocol introduced in previous chapters is far from the real implementation. Although it contains the most important states and gives a general idea of the interactions and data transfers between cores, there are many things like connection delays, memory latencies, atomic operations or race conditions that need to be specially addressed. Throughout this chapter I will explain in detail how the protocol is implemented in order to give an idea of the challenge that represents to extend the protocol to a third level of cache memory.

This chapter is divided into two big sections, the first one analyses in detail how the cache system is by default implemented in gem5. Concepts, structures and behaviours covered in the first sections are helpful for the second section where all the design, verification and evaluation of the proposed three-level cache hierarchy are presented.

3.1 Original two-level cache hierarchy in gem5

The goal of this section is to describe the operation of a cache hierarchy in gem5 as well as give an idea of the possible challenges and constraints implied in the further addition of one more cache level.

The first subsection describes how the caches communicate between each other and how the memory latencies and channel delays are modelled. The following subsection presents the Finite-State Machine (FSM) that describes the protocol in L1, it makes emphasis in the differences with the protocol previously presented.

MESI_CMP_directory is a directory oriented protocol which means that the sharing status of a particular block of physical memory is kept in one location called *directory*, in this case the directory happens to be in the L2 cache and is also implemented as a FSM. The third subsection of this chapter will explain the FSM in L2 and finally, the fourth subsection explains how atomic operations are handled in the RUBY memory system.

3.1.1 Micro-architectural model

MESI_CMP_directory is designed to be an inclusive protocol, in other words, the cache entries contained in all L1s must be a subset of the entries present in L2. This protocol also uses the write-back policy, which implies that the entries in L1 and L2 may have different data due to a store instruction, L2 keeps the old version and updates its copy until the entries in L1 are evicted.

Figure 3.1 presents the structure of the cache hierarchy. Every CPU has its own private L1 instruction and data cache. Although the kind of request the prefetcher issues are very similar to those issued by the processor, the prefetcher communicates to L1 through a different exclusive queue. This structure is replicated for every CPU in the system and each L1 communicates to the interconnection network through a

set of queues that enables it to send/receive requests/responses. Finally, the figure shows one bigger but unique L2 cache which is also connected through queues to the network. Although it is not shown in 3.1, other important modules such as the DMA or memory controller are connected to the network too.

Note that there is no queue to communicate from L1 back to either the prefetcher or the CPU, the reason is because it is not necessary: consider the case when the prefetcher requests to L1 a cache entry that is already present in L1 then, the request simply is discarded. On the other case where L1 does not have the requested entry, L1 will issue another request to the network and hopefully will get the cache entry before the CPU needs it. In either case, L1 does not need to inform (or the prefetcher does not need to know) if the access was a hit or not.

The situation is slightly different with the CPU requests where the CPU does need to know when the cache entry is available in L1 in order to continue with the execution of the load/store instruction. Whenever the cache entry is available, L1 directly calls a function into the Load/Store unit of the CPU to trigger the execution. In case of a store, the CPU directly modifies the data in the cache and, through the queue. In other words, the mandatoryQueue is only used to inject requests to the caches.

Modelling the cache hierarchy as a set of modules connected by queues allows us to assign different delays to each component and so simulate more accurately the memory latencies and the delays each packet suffers while travelling through the network.

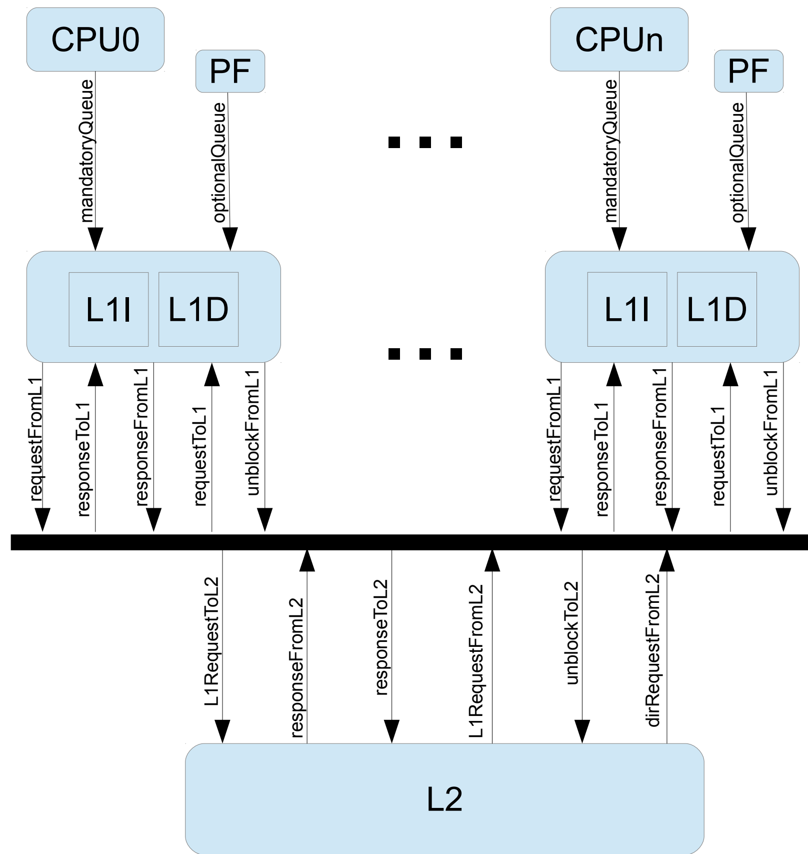

Figure 3.1: Original two level cache hierarchy with private L1 and shared L2.

3.1.2 FSM in L1

In previous sections when the MESI coherence protocol was introduced we assumed that all the memory transactions where atomic. This means that the coherence messages were immediately transmitted from source to destination and hence, it is assumed that no other events can occur between while a state transition is in process. During this big window time a lot of events (either originated by the local CPU or other CPU's) can happen, resulting in complex race conditions.

How to proceed when to processors want to modify the same cache block at the same time? Consider the case where the same entry is shared between many L1s, suddenly L1-A decides to modify the block, so it sends invalidation requests to all other L1s and waits for their invalidation acknowledgement before proceeding with the store (it cannot modify the data until being sure that there is no valid entry in other L1). While waiting for the acknowledgements, L1-A receives a request from L1-B asking to invalidate all the copies because L1-B aims to modify the data too. If L1-A decides to invalidate its copy it will never satisfy its local store request, on the other hand if L1-A decides to go ahead and modify its data and move to *M*, then L1-B would be indefinitely waiting for L1-A's invalidation acknowledgement.

Note that L1-B is also in the same situation than L1-A, but there could be many more cores wanting to modify the same block, and whichever decision they take, it must satisfy all's requests and more importantly guarantee data coherence at all times.

The previously presented four-states protocol is not robust enough to handle this situations. Some auxiliary transient states need to be added so we know that if those states are ever reached we should proceed in a different way. Figure 3.2 shows the state diagram of how gem5 implements *MESI_CMP_directory* in L1. In blue the figure shows the original states of the 4-states MESI protocol, however it also shows in white the transient states that needed to be added to handle all the possible race conditions. The ovals on orange represent the temporary states to handle possible prefetcher requests. Table 3.1 describes each one of the states.

As figure 3.1 shows, the packets (request or responses) travel through the queues, the arrival of a packet to any cache memory is considered to be an event in the coherence protocol. Transitions between states in figure 3.2 are signalled by the arrows and the labels next to them indicate the event that triggered the transition.

Table 3.2 describes each one of the events and also the queue from which it was received.

Due to the large number of states and events, the resulting number of possible race conditions to consider is intractable. Looking for simplification, gem5 allows to block some queues and ”listen” only to those which requests/responses are critical for the protocol (that is the reason why figure 3.2 does not show the transitions of all possible events on each state). For example, the protocol listens to mandatoryQueue (that transmits the processor requests) only in the permanent states (M, E, S, I), then we don’t need to worry about the mandatoryQueue requests while in other states. When the processor issues a request while the protocol is in any transient state, the queue acts like a FIFO and will pop the oldest request as soon as the protocol moves to a permanent state and the queue is unblocked. Some of the queues are stalled on specific states as long as they simplify the protocol and avoid deadlocks.

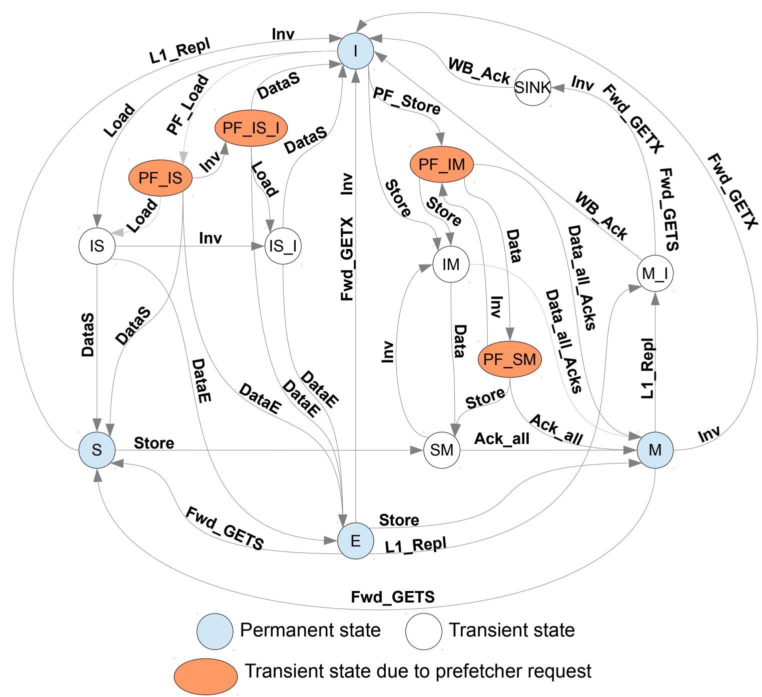

Figure 3.2: FSM describing the coherence protocol originally implemented in L1.

Table 3.1: Definition of states originally implemented in L1.

State	Description
I	Invalid
S	The L1 cache entry is potentially shared with other L1's
E	The cache entry is only present in local L1 and is consistent with the data in L2
M	The cache entry in only present in local L1 and have been modified (write-back when replaced)
IS	L1 issued GETS, have not seen response yet
IM	L1 issued GETX, have not seen response yet
SM	While in S, L1 received from the processor a Store request, L1 issued a GETX but it is waiting for the invalidation acknowledgement from other sharers before proceeding with the modification of the data
IS_I	While waiting in IS, L1 saw an invalidation request
M_I	L1 replacing, waiting for WB_ACK from L2 before moving to I
SINK_WB_ACK	While in M_I saw a Fwd_GETS/GETX, L1 sent the data to the requester an still waiting for the WB_ACK from L2 before moving to I
PF_IS	Issued GETS due to a prefetcher request, have not seen response yet
PF_IM	Issued GETX due to a prefetcher request, have not seen response yet
PF_SM	Issued GETX due to a prefetcher request, received data, waiting for acks
PF_IS_I	Issued GETS due to a prefetcher request, saw inv before data

Table 3.2: Definition of events originally implemented in L1.

Event	**Description**	**Related queue**
Load	Load request from the home processor	mandatoryQueue
Ifetch	Instruction fetch from the home processor	mandatoryQueue
Store	Store request from the home processor	mandatoryQueue
L1_Replacement	Replacement in L1 triggered by a processor request	mandatoryQueue
PF_Load	Load request from the local prefetcher	optionalQueue
PF_Ifetch	Instruction fetch request from the local prefetcher	optionalQueue
PF_Store	Store request from the local prefetcher	optionalQueue
Fwd_GETX	L1 received a GETX request from other processor	requestToL1
Fwd_GETS	L1 received a GETS request from other processor	requestToL1
Fwd_GET_INSTR	L1 received a GET_INSTR request from other processor	requestToL1
Data	Local L1 receives data from L2, data considered as shared	responseToL1
Data_Exclusive	Local L1 receives data from L2 with the certainty of exclusivity	responseToL1
DataS_fromL1	Local L1 receives shared data from other L1 as a response to a GETS request	responseToL1
Data_all_Acks	Local L1 receives data along with the certainty that all other L1's invalidated their copy	responseToL1
Ack	Invalidation acknowledgement to local L1 from other L1	responseToL1
Ack_all	Last acknowledgement to receive before considering the data is no longer shared with other L1's	responseToL1
WB_Ack	acknowledgement from L2 after replacing a block and writing back	responseToL1
Inv	L2 asks L1 to invalidate the data block	requestToL1

3.1.3 FSM in L2

The directory acts as the arbiter of the protocol, it keeps track of the sharing state of each one of the cache entries and grants modification permissions to L1's. The fact that there is only one inclusive L2 for all the system, makes the L2 cache the best place to implement the directory. As well as the first level, the coherence protocol is implemented through a FSM and must face all the special circumstances that L1 does.

Figure 3.3 depicts the state machine of L2 (or the directory) while the states and events descriptions are shown in tables 3.3 and 3.4 respectively. The states labelled as "Blocking" stalls all the requests coming from the queue L1requestToL2. When the events triggering the transitions to other states get to L2, the queue gets unblocked and the remaining requests are serviced with a first come first serve policy.

It is worth noting that both state machines (L1 & L2) must be synchronized at all times. Therefore, in order to guarantee consistency, coherence and inclusivity, there are some combinations of states that should never happen. For example if L2 is in state *SS* there is a pool of permanent and transient states in L1 compatible with L2 like *I*, *S*, *IS*, *SM* or *PF_IS*. However, if *SS* in L2 and *M* in L1 coexists, that could result in a coherence violation.

Furthermore, the only L1 states able to coexist with *NP* in L2 are *I*, *IS*, *IM*, *PF_IS* and *PF_IM*. The situation of having any other state in L1 would imply a violation to the inclusivity principle because L1 contains an entry that L2 does not.

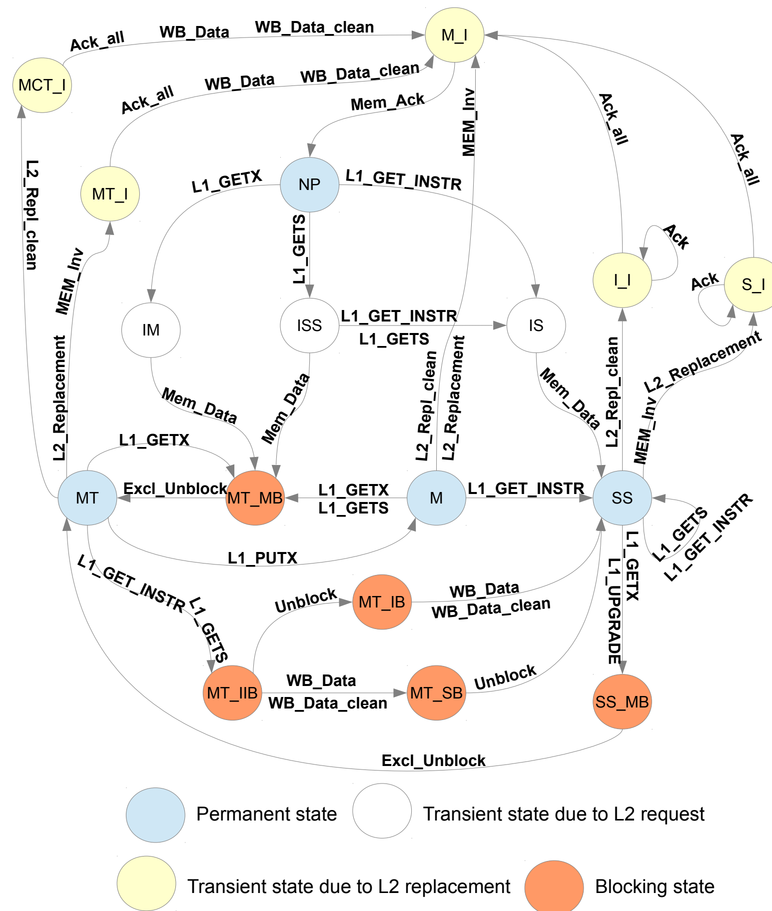

Figure 3.3: FSM describing the coherence protocol originally implemented in L2.

Table 3.3: Definition of states originally implemented in L2.

State	Description
NP	Not present in either cache
SS	L2 cache entry Shared, also present in one or more L1s
M	L2 cache entry Modified, not present in any L1s
MT	L2 cache entry Modified in a local L1, assume L2 copy stale
M_I	L2 cache replacing, have all acks, sent dirty data to memory, waiting for ACK from memory
MT_I	L2 cache replacing, getting data from exclusive
MCT_I	L2 cache replacing, clean in L2, getting data or ack from exclusive
I_I	L2 replacing clean data, need to inv sharers and then drop data
S_I	L2 replacing dirty data, collecting acks from L1s
ISS	L2 idle, got single L1_GETS, issued memory fetch, have not seen response yet
IS	L2 idle, got L1_GET_INSTR or multiple L1_GETS, issued memory fetch, have not seen response yet
IM	L2 idle, got L1_GETX, issued memory fetch, have not seen response(s) yet
SS_MB	Blocked for L1_GETX from SS
MT_MB	Blocked for L1_GETX from MT
MT_IIB	Blocked for L1_GETS from MT, waiting for unblock and data
MT_IB	Blocked for L1_GETS from MT, got unblock, waiting for data
MT_SB	Blocked for L1_GETS from MT, got data, waiting for unblock

Table 3.4: Definition of events originally implemented in L2.

Event	**Description**	**Related queue**
L1_GET_INSTR	A L1I issued a GET_INSTR request	L1RequestToL2
L1_GETS	A L1D issued a GETS request	L1RequestToL2
L1_GETX	A L1D issued a GETX request	L1RequestToL2
L1_UPGRADE	A L1D is sending a dirty version of its data to other L1D's, upgrade the L2 copy	L1RequestToL2
L1_PUTX	L1 replacing data	L1RequestToL2
L1_PUTX_old	L1 replacing data, but no longer sharer	L1RequestToL2
L2_Replacement	L2 Replacement	L1RequestToL2
L2_Replacement_clean	L2 Replacement, but data is clean	L1RequestToL2
Mem_Data	Data from memory controller	responseToL2
Mem_Ack	Acknowledgement from memory controller	responseToL2
WB_Data	Dirty write-back data from L1	responseToL2
WB_Data_Clean	Clean write-back data from L1	responseToL2
Ack	Write-back acknowledgement between L1's	responseToL2
Ack_all	Last write-back acknowledgement between L1's	responseToL2
Unblock	Unblock from L1 requestor	unblockToL2
Exclusive_Unblock	Exclusive unblock from L1 requestor	unblockToL2
MEM_Inv	Invalidation request from memory controller	responseToL2

3.1.4 Support to atomic operations

Before proceeding with an example to see how the state machines in L1 and L2 interact with each other, it is important to analyse how a key concept in thread synchronization is handled: atomic operations.

Although they do not contribute to the system's coherence, atomic operations appear in most of the modern multi-processor systems where it is possible that different cores operate on the same data block concurrently. Sometimes, the programmer may need some certainty that, at least during a small fraction of time, no other processor can access a specific data block; atomic operations give that certainty. Even though they are not formally part of the coherence protocol definition, atomic operations do benefit from the way the protocol guarantees a block's exclusivity.

Atomic operations are those operations to be executed without any other process being able to read or modify the state that is used during the operation, they usually read and modify a given memory location and are called "atomic" because they appear to occur at a single instant between its invocation and its response.

In reality there is no such "atomicity" while reading and modifying the caches, nevertheless, there exist some techniques that avoid threads to access the same memory location while the operation is in execution. Implementing a single atomic operation introduces some challenges, since it requires both a memory read and a write in a single, uninterruptible instruction.

An alternative is to have a pair of instructions where the second instruction returns a value from which it can be deducted whether the pair of instructions was executed as if the instructions were atomic. The pair of instructions is effectively atomic if it appears as if all other operations executed by any processor occurred before or after the pair[19].

There exist many atomic operations (*test-and-set*, *fetch-and-increment*, *read-modify-write*, *load-linked/store-conditional*, etc) but their principle is the same, the first part of the operation reads the memory and the second writes it. Independently of the ISA or the type of atomic operation, the coherence protocol handles the atomic operation as a pair of stores.

Lets suppose the atomic operation is to be executed in a L1 cache entry whose initial state is *S*. L1 will issue the first store, asking to the others L1's to invalidate their copies. After all the invalidation acknowledgements are received, L1 changes its state to *M* and is confident that no other L1 can access the cache entry. This is the moment when the first part of the atomic operation (read) is executed and L1 gets blocked, this means, except from mandatoryQueue, all the input queues get stalled. At this point, L1 is unable to receive requests others than those coming from the CPU (see figure 3.1 as reference). In such way, L1 ensures that even other cores may ask for the cache entry, the entry won't be shared until the atomic operation is concluded.

Eventually, the second part of the atomic operation (second store) is executed, L1 gets unblocked and services all the previously stalled incoming requests in a normal fashion.

3.1.5 Retaking the concurrent store problem

With all being said, in order to illustrate how the state machines in L1 and L2 interact with each other, we are in a good position to retake the situation where two different L1s wish to write the same cache block at the same time. The process is depicted in the time diagram of figure 3.4.

The initial state is when the L1-A is in state *S* and as far as L1-A concerns, L2 should be in *SS* and the other L1s in either *S* or *I*. Then L1-A gets a store request

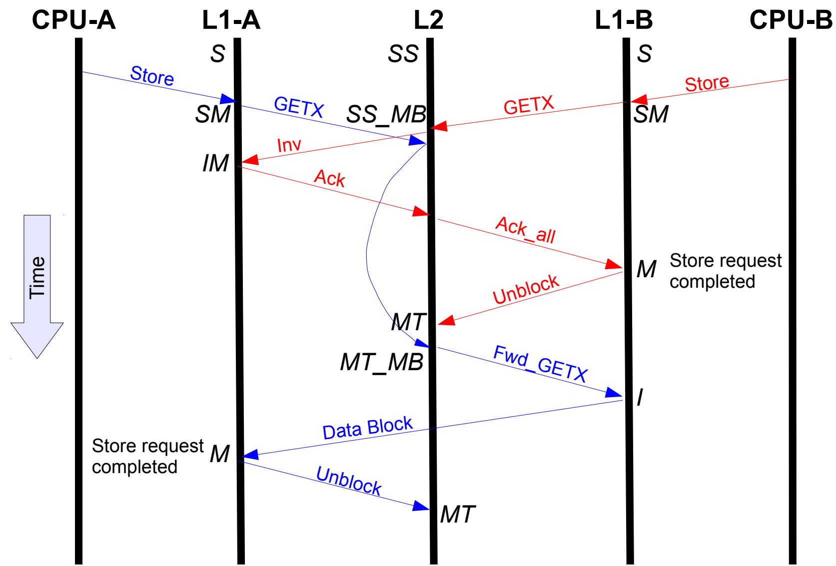

Figure 3.4: Example of concurrent store requests.

from its local CPU, sends the GETX request to L2, moves to the state *SM* and waits for all the invalidation acknowledgement.

While waiting in *SM*, L1-A received an invalidation request from L2 stating that another L1 (L1-B) needs to modify the same cache block (this implies that L1-B is waiting in the state *SM* too). L1-A is receiving this request because the L1-B's GETX request arrived before to L2 than L1-A's (may be L1-A issued the request before, but due to bus delays the request from L1-B won the race). Thus, when the request from L1-B arrived to L2, L2 changed its state from *SS* to *SS_MB* and the L1-A's GETX request will be stuck in the queue until L2 moves to a non-blocking state.

L1-A assumes that by the time its request arrived to L2, L2 was already blocked due to a someone else's request and that L2 will service its request when it gets unblocked Thus, it has no better choice but to invalidate its copy, change the state to *IM* and send the acknowledgement.

When L1-B gets all the acknowledgements, it changes its state to *M*, performs the store and sends an unblock message to L2 indicating that the block was received and it is safe now to service the other L1's requests.

After receiving the unblock message, L2 will move from *SS_MB* to *MT* and, supposing the L1-A's request is the next in the queue, L2 will send a Fwd_GETX message to L1-B asking to invalidate its copy and send the data to L1-A. L2 gets blocked again and moves to *MT_MB*.

L1-B receives the Fwd_GETX request, moves from *M* to *I* and forwards directly the data block to L1-A. Then, L1-A receives the package, moves from *IM* to *M*, performs the store operation and sends another unblock message to L2 to trigger the transition from *MT_MB* back to*MT*.

In this way, all the requests were satisfied, all the operations got the most recent version of the block and both, L1 and L2 ended up in permanent and compatible states, allowing further requests to be serviced.

3.2 Extension to three cache levels

Although gem5 is one of the most popular full-system simulators among the computer architecture, so far, its stable version does not support a coherence protocol with three levels of cache memories. Implementing an L3 cache hierarchy in gem5 would make the simulations more similar to today's systems.

The previous sections described how the L2 cache hierarchy implemented by default in gem5 works, it also walks through some of the implementation details. If an additional cache level is to be added to the original L2 hierarchy three things must be guaranteed for the proper operation of the system:

- Compatibility with the former CPU interface.
- Compatibility with the former memory controller interface.

- Data coherence between CPU's.

In the first parts of this section, I will explain the implementation details of the proposed cache hierarchy. Then, some popular verification techniques for coherence protocols are presented and a description of the verification technique applied to this new system takes place. Finally, some simulation results of the performance of the system with three cache levels are shown.

3.2.1 New memory hierarchy

The memory hierarchy to implement the three levels of caches was chosen to be very similar to current multi-core implementations where each core has its local private memory and connected to all of them, there is a bigger and shared Last Level Cache (LLC). Figure 3.5 shows the block diagram of the proposed hierarchy as well as all the queues used for communication between the caches. Compared to the original hierarchy (figure 3.1) a new level was added between L1 and L2, hence the former L2 keeps acting as the LLC but is renamed as L3.

Thus, every core has its own private L1D and L1I directly attached to it. A prefetcher may issue some requests to the first level as well. Going down the hierarchy we find a private second level which unifies both instruction and data caches. L2 connects to the bus or network that allows it to communicate with the LLC or other cores.

At first sight, the addition of this new level looks very straight forward, but it is not. By far, the most complex element in the hierarchy is L2 which has to satisfy constraints like:

- Full inclusive hierarchy ($L1I/D \subset L2 \subset L3$).

- Write-through policy between L1 and L2.

- Write-back policy between L2 and L3.
- Data coherence between different L2's.
- Data coherence between pairs of L1 and L2.

Now, L2 has two major roles, first it has to handle all the coherence issues between different cores, just in the same way as L1 does in the L2 hierarchy and described in the previous sections. Second, it has to ensure that the data versions contained in L1 and L2 are the same at all times, or at least from the other core's point of view.

Since L1 is smaller than L2, it is also faster, and many requests could get to L2 and can't be serviced immediately because L2 is working with other previous requests. This latency disparities forces us to communicate L1 and L2 with buffers, consequently, the communication from L1 to L2 is not immediate and depends on the buffer saturation.

Every time the CPU modifies a cache entry in L1 the change must be "immediately" reflected in L2 (write-through policy) however, due to the buffer delay there is a time window where L1 and L2 have different data and that could leave the door open to many events to happen. For example, if the L2 receives an sharing request while L1 and L2 are different, L2 could potentially share with other cores the "old" version of the cache block. In other words, before sharing with other L2's or writing-back to L3, the local L2 must be sure that it has the most recent version of the data.

Thus L2 must handle not only race conditions inherent to the coherence between cores, but also race conditions due to L1-L2 synchronization and even the combination of both kinds of race conditions.

Regarding the implementation, the FSM in L3 is exactly the same as the FSM present in L2 in the previous hierarchy (figure 3.3).

In the following subsections I will explain some implementation details of the states machines in L1 and L2 as well as some possible race conditions and the way they are handled.

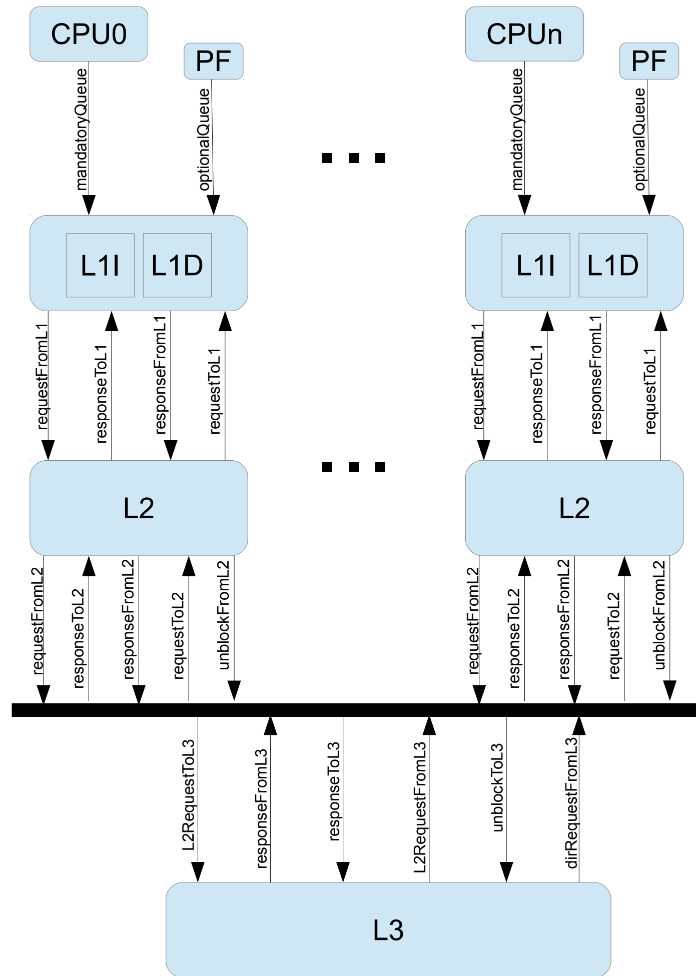

Figure 3.5: Proposed three level cache hierarchy.

3.2.2 New FSM in L1

Since in this new hierarchy L1 does not have any direct contact with other processor's caches, it does not have to worry about keeping coherence with other cores. The main task of L1 is dispatch all the CPU's requests, if the request can be solved locally L2 does not need to know about that. When the requested cache entry is not in L1, the request is forwarded to L2 and L1 waits until the entry is supplied. In case of store, first L1 should acknowledge L2 so it can verify that the block is coherently safe to modify. Then after the store execution, the new value must be propagated to the second level and L1 should not receive more CPU requests until being sure that L2 is updated.

At first sight, it seems that a two-state model to express only the validity and non-validity of the block is enough to service the CPU's requests and communicate with L2. However, with this two states there is no way to differentiate if the block is shared with other cores and if it is coherently safe to write to it. Because of that, there are three different permanent states in L1; one to signal when the cache block is not present or invalid, other when the block may be potentially shared with other cores (read only state), and the last state where where the machine has the certainty that no other L1 has the same block (read-write state).

The state diagram of the proposed FSM in L1 is shown in figure 3.6. It shows in different colors the transient and permanent states, as well as those states created exclusively to handle the prefetcher requests. The arrows represent the transitions between states and close to each transition, in bold capital letters, there are the events that can trigger each transition. Also, the most important actions L1 does during each transition are signalled in italic letters bellow each arrow. Tables 3.6 and 3.5 offer a full description of the states and the events, respectively.

In the same way as in the previous hierarchy, L1 can block some queues to stall the incoming packets at specific states and make the design simpler. For instance, it is not possible to receive any request from the CPU if the current state is transient, the request will be received and handled whenever L1 gets into a permanent state.

Table 3.5: Definition of the proposed events for L1.

Event	**Description**	**Related queue**
Load	Load request from the home processor	mandatoryQueue
Ifetch	Instruction fetch from the home processor	mandatoryQueue
Store	Store request from the home processor	mandatoryQueue
L1_Replacement	Replacement in L1 triggered by a processor request	mandatoryQueue
PF_Load	Load request from the local prefetcher	optionalQueue
PF_Ifetch	Instruction fetch request from the local prefetcher	optionalQueue
PF_Store	Store request from the local prefetcher	optionalQueue
PL1_Replacement	Replacement in L1 triggered by a prefetcher request	optionalQueue
DataS_to_L1	Local L1 receives data from local L2 potentially shared by other cores	responseToL1
DataE_to_L1	Local L1 receives data from local L2 present exclusively in the home core	responseToL1
Ack_to_L1	acknowledgement from L2 to L1	responseToL1
Move_toS	Message from L2 to L1 to inform that the data will be shared so it should not be modified without the directory permission	requestToL1
Inv_L1	L2 asks L1 to invalidate the data block	requestToL1

To better understand the operation of the FSM in L1, let us consider the case where a CPU issues a read request over a cache block, after the read is done, the CPU will attempt to do a store. Suppose the cache block is initially invalid (I),

Table 3.6: Definition of the proposed states for L1.

State	Description
I	Invalid
VS	Valid in shared mode. Load and Ifetch are executed immediately, cannot perform Stores without the permission of the directory
VE	Valid in exclusive mode. Loads, Ifetch and Store are executed immediately
IV1	L1 issued GETS/GET_INSTR, waiting for the response
IV1_I	L1 was waiting in IV1 and received an invalidation request from L2. Block must be invalidated after it gets to L1 and the processor reads the data
IV1S	While in IV1, L1 received a Move_toS request so even if it receives DataE_to_L1 the target state is VS and not VE. This is due to a race condition between the packets Move_toS and DataE_to_L1
IM1	L1 issued GETX, waiting for the response
IM1_I	L1 was waiting in IM1 and received an invalidation request from L2. Block is on its way to invalidation
VM	A L1 valid data block was modified, sent write through packet to L2, waiting for the acknowledgementfrom L2 before doing other memory operations in order to ensure data consistency between L1 and L2 at anytime
VM1_I	L1 received an invalidation request while waiting for the from L2. The block is on its way to invalidation
PIV1	Due to a prefetcher request, similar to IV1
PIV1_I	Due to a prefetcher request, similar to IV1_I
PIV1S	Due to a prefetcher request, similar to IV1S
PIM1	Due to a prefetcher request, similar to IM1
PIM1_I	Due to a prefetcher request, similar to IM1_I

then L1 receives a load request from the local CPU. Since the requested block is not present, L1 forwards the request to L2, does the transition to *IV1* and waits there for the L2 response. After sometime, L2 answers the request and sends the data marked as shared to L1. This means that the block is potentially being shared with other cores, hence, L1 moves to *VS* and services the load request to the CPU. In that state L1 can service as many load/ifetch requests as the CPU issues.

Some time later, the local CPU will attempt to perform a store in the cache block. Although the block is present in L1, the request cannot be serviced immediately because, since the block is shared, that would violate the system's coherence. Instead, L1 sends another request to L2 and waits for the response in the *S2E* state.

Once L2 made sure that copies of the same block in all other private caches are invalid, it can go ahead and send the response to L1. As soon as L1 gets the response, it does the transition to *VM*, services the store request to the CPU and sends to L2 the new version of the block (write-through data). L1 waits until it receives an acknowledgement from L2 signalling that L2 has updated its block version, then, L1 moves to *VE* and is ready to receive further requests from the processor.

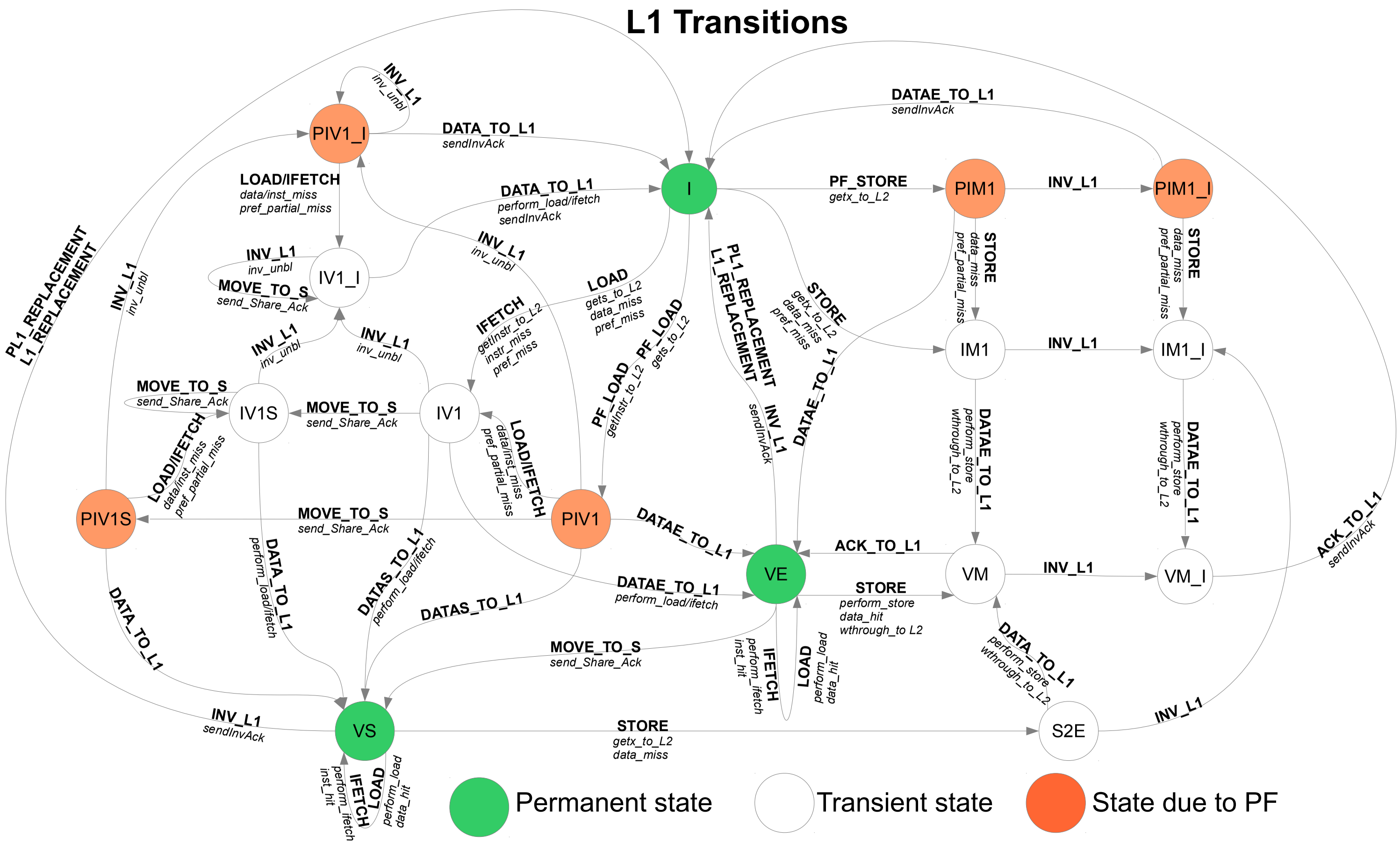

Figure 3.6: Proposed state diagram for L1.

3.2.3 New FSM in L2

The state machine implemented in L2 is by far the most complex among the cache hierarchy, mainly because of the great amount of race conditions that is exposed to. The L2's design complexity yields on the arbitration and handling of both, the requests coming from L1 and the requests regarding the coherence protocol (coming from the directory or other L2's) at the same time. Hence, L2 should concurrently answer to requests on two different fronts in such a way that the state in L1 corresponds at all times to the state in L2 & L3 (directory).

This FSM tried to follow the same the main principles of the L1's FSM in the L2 hierarchy plus additional extensions to support communication with the upper level (L1). The proposed state machine for L2 is shown in figure 3.7. States in green and blue represent the permanent and transient states present in L1 with the original gem5 cache hierarchy. The remaining states in white are transient and were added to handle all the race conditions. In the same mode as previous state diagrams, the transitions, events and main actions performed during the transition are also described in 3.7. The former transient states are inherited from the original L2 MESI protocol and their objective is to maintain coherence between other L2 Caches and the interconnection network. New transient states were added with the aim of dealing with the race conditions while synchronizing L1 and L2. The arrows represent the transitions and the legends next to them indicate the event that triggered causing the transition (in capital letters) and the most important actions done in the transition.

Tables 3.7 and 3.8 contain a full description of the states and events, respectively. Inside table 3.7 the recently added transient states appear in italic font type.

Table 3.7: Definition of the proposed states for L2.

State	**Description**
I	Invalid
S	Data is potentially shared with other cores. Directory must respond to any data request
E	Data is only present in L3 and the local core. Local L2 must respond to requests from other cores
M	Data is only present in the local core and is dirty. In case of replacement, invalidation or share request from other cores, local L2 must write the data back to L3
IS	When in state I, L2 saw a GETS/GET_INSTR request from L1. L2 Forwarded that request to the intrachip network, waiting for the response with data
IS_A	Saw an invalidation request while waiting in IS and forwarded the invalidation request to L1, waiting for L1_Inv_Unbl in order to ensure that L1 is aware that the data block must be on its way to invalidation. L2 in blocked because will only forward the data or answer to other requests after it gets the L1_Inv_Unbl from L1
IS_I	In IS_A received L1_Inv_Unbl, waiting for the data to forward it to L1 and then invalidate the block

Table 3.7: Continued.

State	**Description**
IM	When in state I, L2 saw a GETX request from L1. L2 Forwarded that request to the intrachip network, waiting for the answer with data and the exclusivity permission before modifying the block
SM	Moving from Shared to Modified. While in S saw a GETX request from L1, issued an upgrade request to the intra-chip network and is waiting for the acknowledgements from all possible sharers before modifying the block
EWI	Exclusive Waiting for Invalidation acknowledgement from L1. Base state of the cluster of states EWI. While in E L2 received an invalidation request and forwarded it to L1 to ensure the principle of inclusivity. Waiting for L1_Ack_Inv before invalidating the block in L2 and sending the invalidation acknowledgement to the intra-chip network
EWW	Member of the cluster of states EWI. L2 received a GETS/GET_INSTR request from L1 while waiting in EWI, must answer to this request and make sure the block in L1 is invalid

Table 3.7: Continued.

State	**Description**
MWI	Modified Waiting for Invalidation acknowledgement from L1. Base state of the cluster of states MWI. While in M L2 received an invalidation request and forwarded it to L1 to ensure the principle of inclusivity. Waiting for L1_Ack_Inv before invalidating the block in L2 and write the data back to L3
MWW	While in a state of the cluster MWI, L2 received a GETS/GET_INSTR request from L1, must answer to this request and make sure the block in L1 is invalid
MWX	While in a state of the cluster MWI, L2 received a GETX request from L1, must respond to this request and wait for two events to happen before moving to MWI: L1_PUTX (write-through to ensure consistency between L1 & L2) and L1_Ack_Inv
MWY	While in a state of the cluster MWI, L2 received a PUTX message from L1. Must update the data in L2 to ensure consistency and wait for L1_Ack_Inv before moving to MWI
MWZ	Member of the cluster MWI. While waiting in MWX there is a race condition between the messages L1_PUTX and L1_Ack_Inv, in this case L1_Ack_Inv won the race and is waiting for L1_PUTX to moving to MWI

Table 3.7: Continued.

State	Description
FWI	Willing to Forward data Waiting for Invalidation acknowledgement from L1. Base state of the cluster of states FWI. While in E or M L2 received Fwd_GETX request from other core and sent an invalidation request to L1. Waiting for L1_Ack_Inv before forwarding the block to the requester
FWW	While in a state of the cluster FWI, L2 received a GETS/GET_INSTR request from L1, must answer to this request and make sure the block in L1 is invalid
FWX	While in a state of the cluster FWI, L2 received a GETX request from L1, must respond to this request and wait for two events to happen before moving to FWI: L1_PUTX (write-through to ensure consistency between L1 & L2) and L1_Ack_Inv
FWY	While in a state of the cluster FWI, L2 received a PUTX message from L1. Must update the data in L2 to ensure consistency and wait for L1_Ack_Inv before moving to FWI
FWZ	Member of the cluster FWI. While waiting in FWX there is a race condition between the messages L1_PUTX and L1_Ack_Inv, in this case L1_Ack_Inv won the race and is waiting for L1_PUTX to moving to FWI

Table 3.7: Continued.

State	Description
MER	Modified or Exclusive block must be Replaced. Base state of the cluster of states MER. Sent an invalidation request to L1 to ensure the principle of inclusivity. Waiting for L1_Ack_Inv before writing the data back to L3
MERW	While in a state of the cluster MER, L2 received a GETS/GET_INSTR request from L1, must answer to this request and make sure the block in L1 is invalid
MERX	While in a state of the cluster MER, L2 received a GETX request from L1, must respond to this request and wait for two events to happen before moving to MER: L1_PUTX (write-through to ensure consistency between L1 & L2) and L1_Ack_Inv
MERY	While in a state of the cluster MER, L2 received a PUTX message from L1. Must update the data in L2 to ensure consistency and wait for L1_Ack_Inv before moving to MER
MERZ	Member of the cluster MER. While waiting in MERX there is a race condition between the messages L1_PUTX and L1_Ack_Inv, in this case L1_Ack_Inv won the race and is waiting for L1_PUTX to moving to MER
M_I	L2 has already sent the write-back data to L3, it is waiting for L3_WB_Ack before moving to I and freeing the data block

Table 3.7: Continued.

State	Description
SINK_WB_ACK	When L2 is in M_I waiting for L3_WB_Ack and receives requests from other cores or the directory like Fwd_GETS/Fwd_GET_INSTR/Fwd_GETX/Inv. L2 dispatches those requests and moves to this state where is still waiting for L3_WB_Ack
METS	Modified or Exclusive moving to Shared. Base state of the cluster of states METS. While in E or M L2 received Fwd_GET_INSTR/Fwd_GETS request from other core and sent a Move_toS request to L1. Waiting for L1_Share_Ack before sharing the block with requester
MESW	While in a state of the cluster METS, L2 received a GETS/GET_INSTR request from L1, must answer to this request and make sure the block in L1 is in a shared state
MESX	While in a state of the cluster METS, L2 received a GETX request from L1, must respond to this request and wait for two events to happen before moving to METS: L1_PUTX (write-through to ensure consistency between L1 & L2) and L1_Share_Ack
MESY	While in a state of the cluster METS, L2 received a PUTX message from L1. Must update the data in L2 to ensure consistency and wait for L1_Share _Ack before moving to METS

Table 3.7: Continued.

State	Description
MESZ	Member of the cluster METS. While waiting in MESX there is a race condition between the messages L1_PUTX and L1_Share_Ack, in this case L1_Share_Ack won the race and is waiting for L1_PUTX to moving to METS
SWI	Shared Waiting for Invalidation acknowledgement from L1. L2 received an invalidation request while in S, forwarded the request to L1. Waiting for L1_Ack_Inv
SWW	While waiting in SWI L2 received L1_GETS/L1_GET_INSTR, must answer the request and wait for L1_Ack_Inv before moving to SWI
SR	Shared block must be Replaced. Sent an invalidation request to L1 to ensure the principle of inclusivity. Waiting for L1_Ack_Inv before replacing block in L2
SRW	While in SR L2 received a GETS/GET_INSTR from L1, must answer this request and then wait for L1_Ack_Inv before moving back to SR
SRX	While in SR or SRW L2 received L1_GETX. Cannot immediately answer to this request because the data is shared with other cores. Issued an upgrade to the inter-chip network and waiting in this state for the acknowledgements from all sharers before modifying the block and then replace it

Table 3.7: Continued.

State	Description
A	In SWI or SWW L2 received L1_GETX. Cannot answer to L1 request because the block is shared. Cannot issue an upgrade in the network because the directory is blocked waiting for the local core to invalidate its data. Local L2 sends the invalidation acknowledgement to unblock the directory and then an unpgrade to receive the data and the permission of all other cores. This state is like IM with the difference that L1 is in a shared state instead of invalid
B	L2 received a data package while waiting in A, now must wait for the acknowledgements from all other sharers. This state is like SM with the difference that L1 is in a shared state instead of invalid
C	this is a modified state where L2 needs to make sure that data is invalid in L1 before moving to M to ensure synchronization between L1 and L2. This is the base of the cluster of states C
CW	While in a state of the cluster C, L2 received a GETS/GET_INSTR request from L1, must answer to this request and make sure the block in L1 is invalid

Table 3.7: Continued.

State	Description
CX	While in a state of the cluster C, L2 received a GETX request from L1, must respond to this request and wait for two events to happen before moving to C: L1_PUTX (write-through to ensure consistency between L1 & L2) and L1_Ack_Inv
C	While in a state of the cluster C, L2 received a PUTX message from L1. Must update the data in L2 to ensure consistency and wait for L1_Ack_Inv before moving to C
CZ	Member of the cluster C. While waiting in CX there is a race condition between the messages L1_PUTX and L1_Ack_Inv, in this case L1_Ack_Inv won the race and is waiting for L1_PUTX to moving to C

There are cases when after receiving an event, the FSM needs to do some actions but remains in the same state. For example, like when it is in *S* and receives from L1 a GETS, it sends the block to L1 but remains in *S*. These situations should be drawn in the diagram as an arrow reaching the same state from which it originated but in order to keep the diagram in figure 3.7 relatively simple, this transitions are not shown.

In general, L2 must must guarantee:

- **Inclusivity**:Before replacing/invalidating any data block, L2 needs to make sure the block is already invalid in L1 an that won't become valid at least while the block still valid in L2.

- **Coherence**: If L2 is in an exclusive state and is asked to share its data, should not proceed until it is sure that L1 moved to a shared state and hence won't attempt to modify the block without permission of L2. On the other hand if L1 wants to modify a shared data must wait for the directory and L2 permission.

It would be a wrong condition if L1 is in an exclusive state (VE) and L2 and L3 in a shared state (S & SS, respectively), this can lead to have two versions of the same block because L1 may write some data without the L2's permission. Note, that the combinations of states [L1,L2,L3]={[VS,S,SS], [I,S,SS], [I,I,SS]} are allowed, however the combinations [L1,L2,L3]={[VS,I,SS], [VS,S,I], [I,S,I]} violate the principle of inclusivity and may lead to a coherence conflict.

Because of the large number of states and sequences of possible events that can affect L2 (race conditions) the complexity of the state machine is expected to increase. The solution proposed in this work is a structure called "lock of states", it can be seen as a loop that L2 cannot leave until a condition in L1 is achieved. Figure 3.7 shows several clusters or subsets of states that share a common pattern of transitions between them and whose operation will be explained in the next section.

L2 Transitions

Permanent state

Former transient state

New transient state

Figure 3.7: Proposed state diagram for L2.

Table 3.8: Definition of the proposed events for L2.

Event	**Description**	**Related queue**
L1_GETS	L1 asks for data for reading purposes	requestFromL1
L1_GET_INSTR	L1 asks for instructions	requestFromL1
L1_GETX	L1 asks for data for writing purposes	requestFromL1
L1_PUTX	L1 just modified the data block, write-through message	requestFromL1
L2_Replacement	Replacement in L2, L1 request a new data block but there is no room for it	requestFromL1
L1_Ack_Inv	Invalidation acknowledgement from L1	responseFromL1
L1_Inv_Unbl	Invalidation acknowledgement from L1 prior to the data arrival. This means L1 is aware that the data block should be on its way to invalidation as soon as it gets there	responseFromL1
L1_Share_Ack	Data in L1 is on a shared state, L2 can go ahead and share it with other cores	responseFromL1
Inv	Invalidation request from directory or other core	requestToL2
Fwd_GETS	GETS request from other core, local L2 must share the data	requestToL2
Fwd_GET_INSTR	GET_INSTR request from other core, local L2 must share the data	requestToL2
Fwd_GETX	GETX request from other core, local L2 must invalidate its block and send the data to the requester	requestToL2
Data	Data to local core	responseToL2
Data_Exclusive	Local core has the guarantee that data is not present in any other core	responseToL2
DataS_fromL2	Shared data, forwarded by other L2. Need to unblock the directory	responseToL2
Data_all_Acks	Data for local L2 and received the invalidation acknowledgement from all the former sharers	responseToL2
Ack	A invalidation acknowledgement from a sharer was received	responseToL2
Ack_all	The number of invalidation acknowledgements received equals the number of sharers minus one. Data block is only valid in the local core	responseToL2
WB_Ack	Write-back acknowledgement from L3	responseToL2

3.2.4 Locks of states

Although some of the queues are stalled whenever is possible in order to simplify the protocol, the state diagram is still very complex. Actually, the reader may detect that there is a common structure of states and transitions that repeats five times in different parts of the diagram. I call these structures *locks of states* and they ensure that the machine does not leave the lock until certain conditions are met in L1, regardless of all the race conditions that may occur.

Even though they have the same structure, it is not possible to simplify further or merge all the locks into one sole structure because their location in the state machine is different and in some cases, the transitions are different too.

The purpose of the *locks of states* which base states are *MER*, *MWI*, *FWI* and *C*, is to ensure that L1 is invalid before leaving the cluster; while the purpose of the cluster *METS* is to ensure that L1 is in a shared state before leaving the cluster and sharing the data with other cores.

Let's start by analysing the *lock of states* that contains the states *FWI*, *FWW*, *FWX*, *FWY* & *FWZ* and is located at the bottom right corner of figure 3.7.

Assume that the initial state of the system includes the local L2 being in *M*, and L1 in *VE* state. Since L2 is in *M* the cache block is not present in any other private cache. In a situation without race conditions, suppose L2 gets a *FWD_GETX* request which means that other core wants to modify the same data block, so L2 must invalidate all local copies and send to the other core the requested data. In other words, the process is L2 sends an invalidation request to L1 and moves to *FWI*. L1 initially in *VE*, gets the invalidation request from L2, invalidates the data block, sends the invalidation acknowledgement to L2 and moves to *I*. L2 gets the invalidation acknowledgement from L1, sends the data to the other core that will

perform a write and moves to *I*.

Figure 3.8 illustrates the message interchange between L1 and L2. The vertical lines represent each element on the cache hierarchy and the arrows are the messages travelling between them. The resulting state after the arrival of a message is labelled next to the corresponding vertical line.

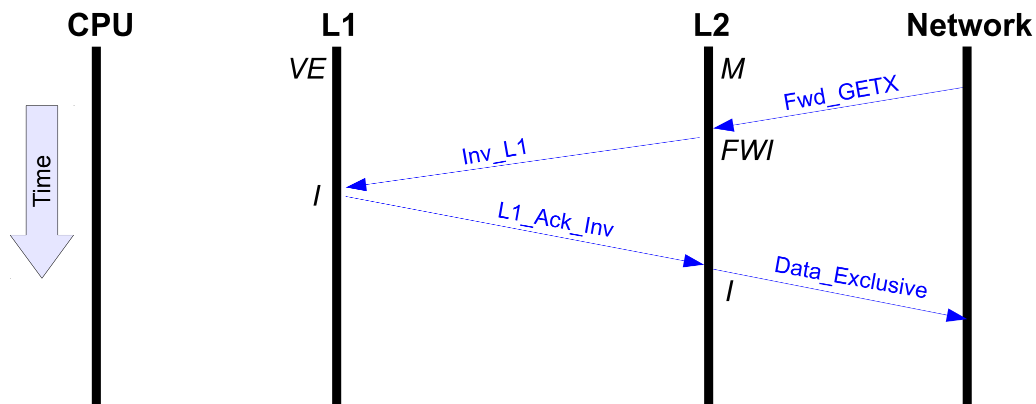

Figure 3.8: Communication example between L1 and L2 with no race conditions.

The transitions in the previous example went really smoothly, however during all the time spent waiting for acknowledgements a lot of events could happen and cause race conditions. The example considers race conditions and is depicted in figure 3.9.

Suppose the initial state of the system again is with L1 in *VE* and L2 in *M*. L2 gets the *FWD_GETX* message, sends the invalidation request to L1 and moves to *FWI*. However, before L1 receives the invalidation request from L1, the local CPU performs a Store and makes L1 forward the new version of the block to L2 and move to *VM*. Few moments later the invalidation request from L2 arrives to L1 but, since

L1 in *VM* is only waiting for L2's acknowledgement, the *requestToL1* queue is stalled and will be serviced once L1 moves back to *VE*.

On the other hand, while in *FWI*, L2 received the write-through data, meaning that a store request won the race and the invalidation request will eventually be serviced. Thus, L2 updates its copy of the block with the new data, sends the acknowledgement to L1 and remains in the same state (*FWI*) waiting for the invalidation acknowledgement.

Few moments later L1 receives the acknowledgement meaning that L2 got the updated block version and moves back to *VE*. Immediately after, the *requestToL1* queue gets unstalled and L1 receives the invalidation request from L2, hence, it sends the invalidation acknowledgement and moves to *I*.

Later on, the CPU issues another store request and L1 issues a *GETX* packet to L2 and moves to *IM* waiting for the block before performing the store. Due to different delays in the queues, the *GETX* request gets before to L2 than the invalidation acknowledgement. That makes L2 send the data block to L1 and move to *FWX*.

L1 gets the data block, performs the store operation, sends the write-through message to L2 and moves to *VM* waiting once again for the acknowledgement. Next, L2 receives the write-through message, updates its block version, sends the acknowledgement to L1 and moves to *FWY*. Few cycles later L2 receives the invalidation acknowledgement that has been in flight all this time sends another invalidation request to L1 (because L1 is still valid) and moves to *FWI*.

Finally, L1 receives the acknowledgement from L2 and moves to *VE*. Moments after, it also gets the invalidation request, invalidates the local copy, sends the invalidation acknowledgement and moves to *I*. After receiving the invalidation acknowledgement from L1, L2 can forward the data block to the requesting core, invalidate

its copy and move to *I*.

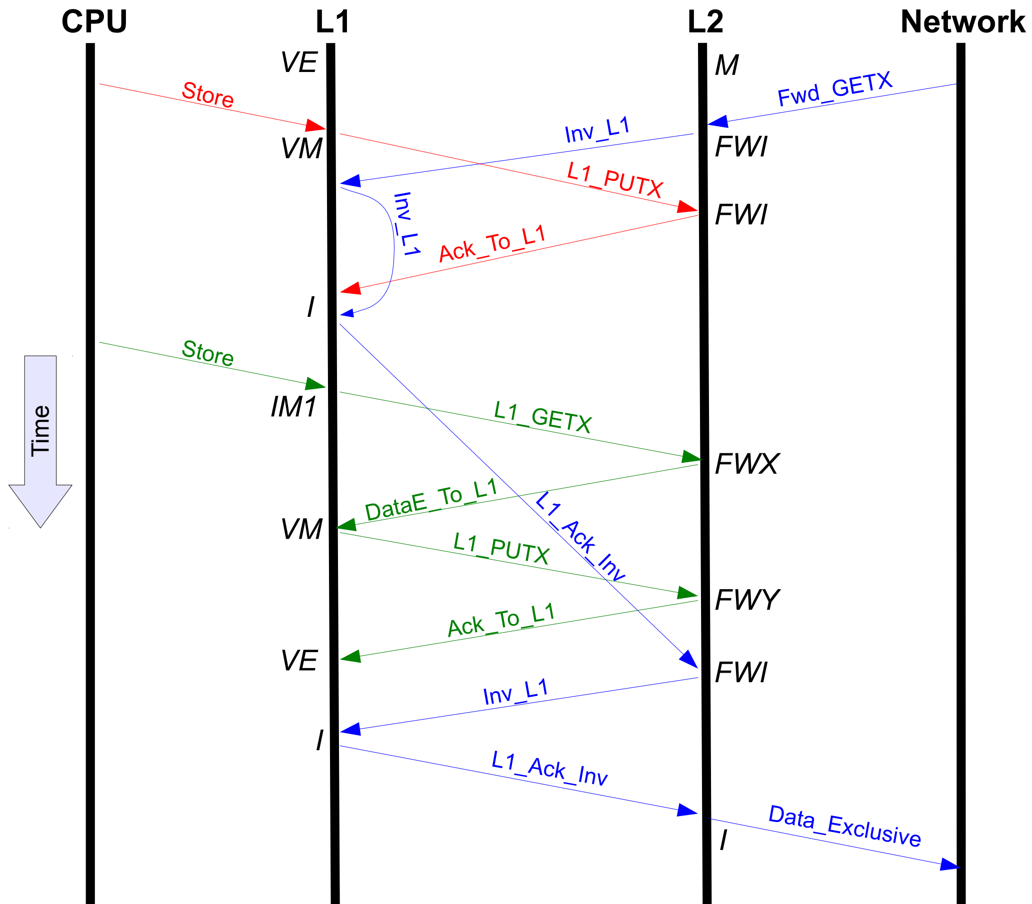

Figure 3.9: Communication example between L1 and L2 with race conditions.

3.2.5 Handling atomic operations with three cache levels

An atomic operation ensures that, while the operation is in process, no other processor will modify the data block the operation works on. In the L2 hierarchy this is handled by first, getting exclusive access to the cache entry; second, blocking the all

the queues that communicate the local private cache (L1) with the interconnection network and other caches (only *mandatoryQueue* remains unblocked); and third, unblocking all the queues when the operation finishes.

The principle for a three level cache hierarchy is the same but with the difference that now L1 and L2 together constitute the CPU's local private cache and every time L1 gets modified, that modification must be reflected in L2 as soon as possible.

Hence, in order to allow synchronization between L1 and L2 not only *mandatoryQueue* but also all the queues between L1 and L2 must remain unblocked. Except those, all the remaining queues should be blocked with the aim of avoiding interference from other cores. See figure 3.5 for reference.

3.2.6 Verification of the new cache hierarchy

Gem5 provides a script (*ruby_random_test.py*) that implements a random traffic generator connected to the whole memory hierarchy, it does not simulate all the full-system components but gets advantage of the gem5's engine simulator and trace capabilities to make a robust tester. Among all the available verification techniques for the new memory hierarchy, the ruby random tester is the best option because of the following reasons:

- **Ease of configuration**: The script allows us to easily specify for each test the number of processors (or testers injecting packets to the cache hierarchy), the total number of packets to be injected and the seed used by the random generation. With only few changes to the command line a completely different verification environment can be run.

- **No state explosion**: Since this method does not exhaustively explores all the reachable global states, it is not susceptible to the state explosion problem. This allows the verification of a system with 64 cores which is harder with other

verification techniques.

- **Detection of erroneous conditions**: Gem5 stops the verification process and outputs the information needed to debug whenever the data received is different from the expected (i.e. did not read the last data written into the block) or a probable deadlock was found. Although incoherent global states are not directly detected, those conditions will eventually end up in either a deadlock or data consistency error.

- **Debug support**: In case of error all the traces and debug features available in gem5 can be used in the tester.

- **No need to describe the FSM in other languages**: The protocol originally described in the gem5's syntax does not need to be re-described in other high-level language like Murϕ. This is also an error prone process because the protocols described in both languages might be different.

- **Micro-architecture verification**: Besides the coherence protocol, the memory hierarchy is composed by many micro-architecture elements with different behavior and latencies that must be verified too. The other techniques only verifies the coherence protocol.

- **Test with real workloads**: Since the memory hierarchy is embedded into a full system simulator it is also possible to run real workloads on it and verify the results.

Despite of the chosen chosen verification technique, the verification process for designs as complex as the L3 cache hierarchy is always a slow and challenging task. Many errors were found and fixed until getting the final version of the protocol shown

in figures 3.5, 3.6 and 3.7. The algorithm followed in this verification process is shown in figure 3.10

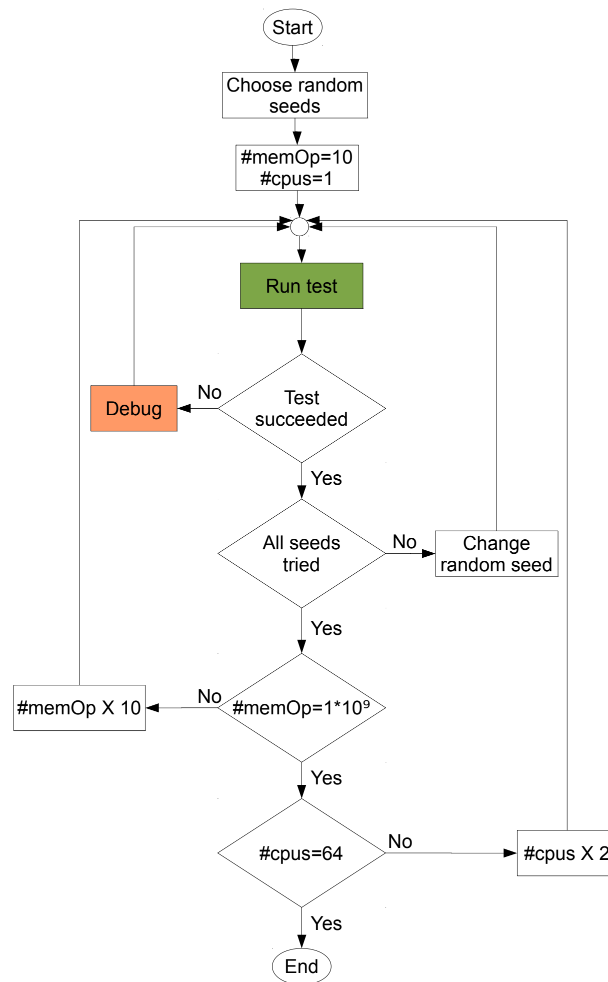

Figure 3.10: Algorithm used to debug and verify the new coherence protocol.

The verification process is an iterative algorithm that starts with one core and few memory operations. As the test results are successful, one can try with different seeds

for the random generator then, gradually increase the number of memory operations and finally, increase the number of cores. This process continues until reaching a reliable level of verification. The L3 cache hierarchy passed all the tests up to simulating 1×10^8 instructions and 64 cores with 5 different random seeds (≈ 6 simulation days per test).

3.2.7 Experiment design and performance results

The L3 cache hierarchy was implemented in gem5 and compared with a L2 baseline system. The specifications of both, the baseline and proposed systems, were chosen to resemble the Intel i7's architecture. I used CACTI[35] to get the latency for each cache configuration assuming a technology process of 32nm and a core frequency of 3GHz. The details of the cache configuration for each system are shown in the table 3.9.

Table 3.9: Specifications for the baseline and proposed hierarchy for the experiment.

Cache level	**Characteristics**	**Baseline**	**L3 hierarchy**
L1	Size	32KB I/32KB D	32KB I/32KB D
	Associativity	4-way I/8-way D	4-way I/8-way D
	Block size	64B	64B
	Latency (cycles)	1	1
L2	Size	2MB per core	256KB
	Associativity	16-way	8-way
	Block size	64B	64B
	Latency (cycles)	35	10
L3	Size	–	2MB per core
	Associativity	–	16-way
	Block size	–	64B
	Latecy (cycles)	–	35

I configured gem5 to simulate 16 out-of-order CPU's and ran the Parsec suite[4]

on it.

Figure 3.11 shows the performance of the L3 system normalized to the baseline. On average, the proposed system performs 29.67% better than the baseline. Although this value is within the expected range, it is necessary to prove that the main cause of performance improvement is the addition of an intermediate cache between the first and last-level caches. Figure 3.12 compares the accesses to the last-level cache (L2 for the baseline and L3 for the proposed system). What figure 3.12 implies is that 65.31% of the requests that would have gone to the LLC hit in the second level and were serviced faster than the baseline.

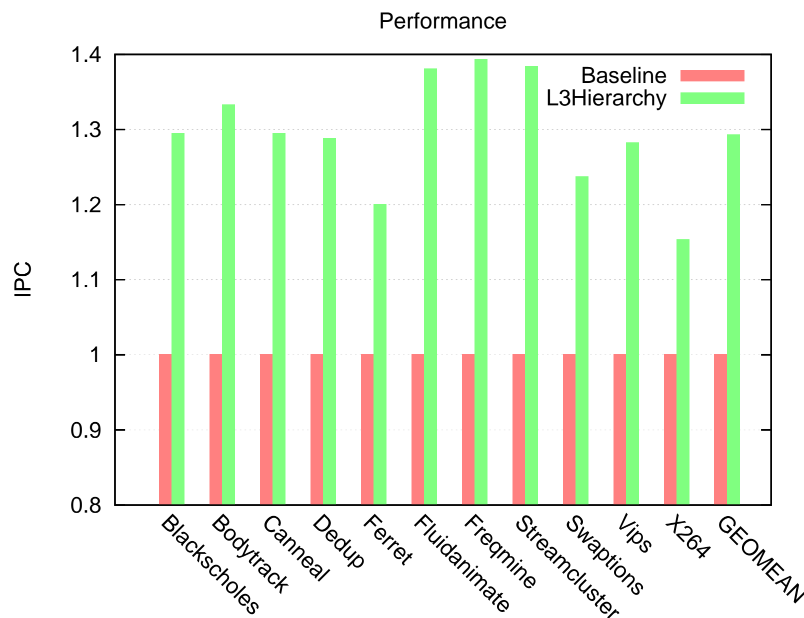

Figure 3.11: Performance improvement with the L3 cache hierarchy.

Finally, in order to demonstrate the correct performance operation of the states dedicated to attend the prefetcher's requests, I ran one additional set of simulations with the prefetcher activated. Although this condition was previously verified, it

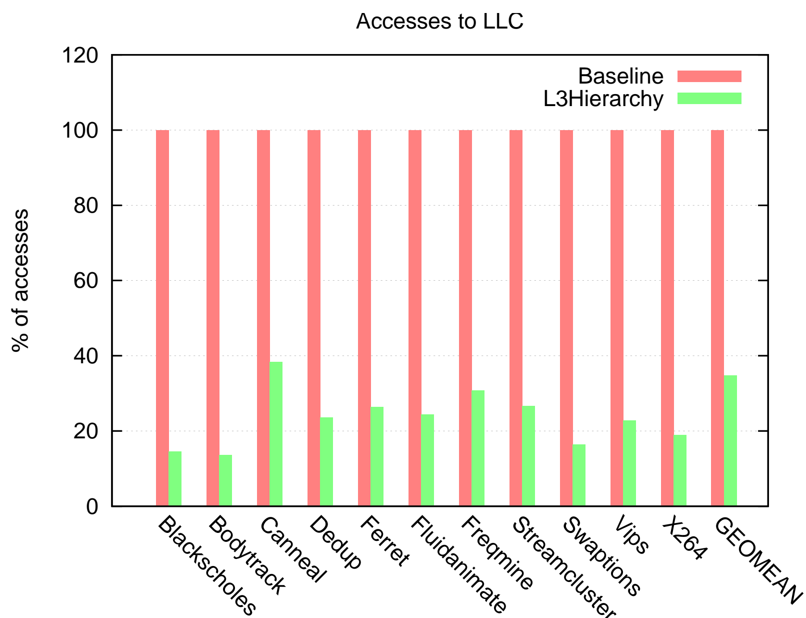

Figure 3.12: Reduction of accesses in LLC with the L3 cache hierarchy.

is worth noting that all the benchmarks executed correctly. Furthermore, figure 3.13 shows that the prefetcher does affect the performance if the system. It is not the goal of this work to analyse the prefetcher impact on the performance, but possible causes to the marginal improvement on the IPC are: wrong speculation, cache contamination and the number of prefetcher requests is negligible compared with the amount of loads and stores.

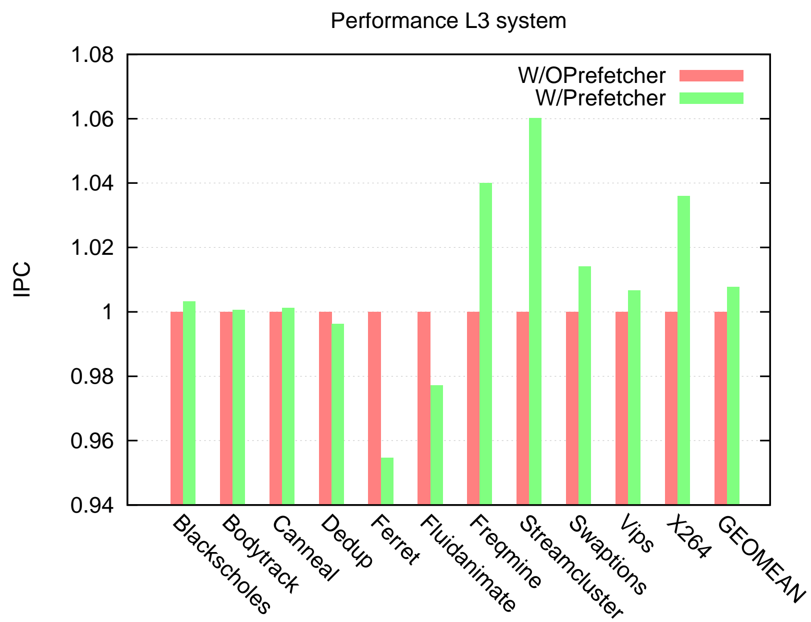

Figure 3.13: Performance comparison of a system with or without prefetcher.

4. STATISTICAL SAMPLING SIMULATION IN GEM5

The most reliable tools researchers in computer architecture have are the simulators which main goal is to give a first order approximation of the behaviour of systems at the initial design stages. Compared to the processors of decades ago, current designs have many additional structures that support a higher and more efficient performance. The increment on the real processor's complexity had its effects too on the simulators which, by trying to replicate the new features of the recently released systems, also increased its complexity considerably. However, the result of more complexity is not always the same: While it makes real hardware faster, it diminishes the performance of the simulators because the have more structures to simulate.

Today's simulators are typically thousands of times slower than the actual CPU. Full-system simulation, which may include many CPU's, memory transactions, peripherals and other system components increment the slowdown by one or two orders of magnitude[40]. Furthermore, multi-threaded applications tend to be longer than their single-threaded counterparts. Although it is true that current host machines are faster, that does not compensate the huge performance disparity between hardware and simulators. This results in prohibitively long simulation run-times (months or even years) just for programs that take some seconds to execute in real hardware.

Sampling simulation aims to reduce the simulation time with little effects in the results by only simulating some parts of the full program. Many sampling simulation techniques were proposed involving many trade-offs like accuracy, simulation time, disk usage, flexibility. However, there is no common agreement about which technique is the best, and it looks like the choice depends on the platform, simulator and

benchmarks of interest.

There are two aspects of which simulators should worry about in order to guarantee accuracy in the results[37]:

- **Correct memory image to execute the sample**: A normal program has several stages in which the system behaves in different ways. If the samples are not selected carefully enough, they may not fully represent the behaviour of the program.

- **Warm architecture state**: Current processors have many structures that help the to perform better, like caches, branch predictors or TLB's. If those structures don't have the same state right before the sample as they would in a normal non-sampling simulation, the obtained results will underestimate the performance.

Even though it is one of the most popular simulators, gem5 does not have a reliable platform for sampling simulation. It is only able to switch back and forth between two different CPU models for the same amount of time, but that does not result in a noticeable speed-up. Furthermore, if not managed correctly, it might introduce measurement errors.

Throughout this chapter I will explain the extension made to gem5 so that it supports sampling simulation. First I present a survey of the best accepted techniques on sampling simulation and their implications. Latter I will talk about the chosen methodology to implement and the maximum speed-up achievable in gem5. Then I cover some implementation details and finally, I present the results of the experiments.

4.1 Survey on sampling simulation techniques

One approach to shrink the simulation time is to make programs smaller to a point where their simulation is feasible. Actually, some benchmark suites like Parsec[4] have several input sets with different problem sizes and consequently different simulation times. Some of these sets are still too long to fully simulate and those that show an acceptable simulation time, spend to much time on the start up and shut down parts of the program. Moreover, not all the benchmark suites or programs willing to run in the simulator have such input sets.

When researchers first faced the extremely long simulation time, the solution they came up with was to skip the initial instructions of the program (in order to avoid variables and subroutine initialization) and then simulate an arbitrary long subset of instructions. One way to do it is to *fastforward* to a particular point in the execution and then start the cycle-detailed simulation from there. During the fast-forward process the simulator only needs to act at the functional level, where it cannot output any representative results but accelerates the simulation process.

The problem with fast-forwarding is that it serializes the simulation and invariably the researcher needed to wait for the fast-forward to advance from the beginning of the program to the point of interest. As an alternative, most simulators have the ability to execute the program until a given point and save the state into a *check-point* so that other simulations can restore it and start the detailed simulation from there. Nevertheless, some studies indicate that either fast-forwarding or check-pointing may fail to summarize the global behaviour of the program[34, 21].

4.1.1 Warm-up techniques

Another problem to take into account when starting the cycle-detailed simulation at the middle of the program is that all the architectural structures are empty,

hence a warm-up time is needed in order to fill the structures before start collecting representative results and avoid the *cold-start bias*. Some checkpoints might store the full architectural state of the simulator so no warm-up time is needed, but they consume more space on disk. Warming-up big structures like the caches is very time consuming, sometimes the time spent warming-up is much bigger than the time required to measure the samples, consuming most of the total simulation time. Therefore it is crucial to determine the optimal warm-up length[5] that reduces the simulation time without any accuracy sacrifice.

Haskins *et al.* followed that idea and presented one of the first formal attempts to reduce the warm-up length: Minimal Subset Evaluation (MSE)[17]. After characterizing each benchmark, MSE mathematically determines, for a given cache configuration, the warm-up length that will reproduce (with a probability p specified by the user) the simulated hardware state exactly as if cycle-accurate simulation was done instead of fast-forwarding. It is worth noting that MSE was initially designed for L1 caches but is actually flexible to any hardware configuration. However, there is also the possibility of bringing to the structures more blocks than those that will be needed in the sample, making the warm-up unnecessary long.

Further ideas exploit this condition by bringing to the structures only those blocks that will be used in the sample, rather than trying to replicate the exact state like if no fast-forwarding/check-pointing was made. Two years latter Haskins *et al.* presented the concept of Memory Reference Reuse Latency (MRRL)[18]. They claim that memory references that occurred closer to the starting point of the sample are more likely to be used in the sample, so they focused their efforts on ensuring that all the memory references used in the sample are in the cache at the end of the warm-up period. For a given sample in the benchmark, they analyse the trace to figure out

[5]Number of instructions to simulate in cycle-accurate mode before sampling

what memory references are required in the sample, then for each memory reference, they find the most recent reference to the same address before the beginning of the sample. Thus the minimal required sample time is defined by the reference that occurred earlier before the sample. BLRL[12] extended MRRL's work and achieved better results but the main idea is the same.

Since these techniques rely only on the memory dependencies of the program, it is compatible for any architecture; however, a previous analysis of the program is required and the resulting warm-up length varies depending on the behaviour of the program at that point. It is inconvenient for simulations with lots of samples where the user must either specify a different warm-up period for each sample or lose efficiency by using the largest warm-up period indistinctly for every sample. To overcome this problem, Lou proposed the Self-Monitored Adaptive Cache Warm-up technique or SMA[22], where instead of defining the warm-up length before the execution, the simulator constantly monitors the warm-up process of the caches and decides when the caches are warm enough to start sampling. At the beginning of the warm-up process all the caches blocks are initialized to the *cold-start* state. When a block is first accessed and data is brought from main memory, the block changes permanently its state to *valid*. The simulator then, monitors two aspects: The percentage of cache blocks in cold-start state and the percentage of memory accesses to cold blocks during a time interval. When any of the previous two numbers drops below a threshold, the cache is considered warm.

4.1.2 Sampling simulation for single-threaded programs

Simpoint[34] is perhaps, one of the first and most accepted sampling simulation techniques and its goal is to identify subsets of instructions called simulation points that, when simulated and combined accurately, represent the behaviour of the pro-

gram. Simpoint divides the program in to several Basic Block Vectors (BBV) that represent the code blocks executed in a given interval of time, then the magic of Simpoint yields on classifying the BBV's into clusters and choosing a representative for each cluster that better approximates the behaviour of the full cluster. Then the simulation results of each representative (or simulation point) are weighted appropriately to get an approximation of the program's foot print. With the identification of the simulation points the user saves lots of simulation time because, assuming that the micro-architecture is warm at the beginning of each simpoint, the simulation can run from beginning to end and simulate in cycle accurate mode only those instructions blocks indicated by the simpoints. Furthermore, if there are enough computational resources, the user can run in parallel as many simulations as simpoints, after that, it is just matter of gathering the results.

Another approach that does not require a prior analysis of the programs or storage of several simpoints is the Statistical Sampling Simulation, perhaps better represented by the SMARTS framework[42, 43]. SMARTS does periodic sampling of a large number of very small slices of execution throughout all the program simulation. Then, it uses statistics theory to find a measure of variability among the samples and determine the optimal number of samples that captures the program's behaviour within a confidence level. One major difference of smarts is that while it does functional simulation and fast-forwards in between samples, it keeps simulating the micro-architectural state of big structures like the caches. The advantage of this *functional warming* is less *detailed warm-up* time because the large structures never cool-down. In other words, for a given sampling period of T instructions, $T-(W+U)$ instructions will be fast-forwarded (functional warm-up) and only $W+U$ instructions are executed in cycle-accurate mode, from which W represents the detailed warm-up period and U is the sample.

It is worth noting that SMARTS is an iterative algorithm that usually converges fast. At the end of each simulation it outputs the confidence interval of the results for a given confidence level. If any of both parameters is not within user's expectations, there is need to run another simulation with a different sampling period which will allow us to get the optimal number of samples to satisfy the confidence requirements.

In the SMARTS approach the functional warming dominates the total simulation time, for SPEC2000 benchmarks the functional warming occupies hours of simulation while the cycle-detailed simulation requires minutes to complete. As an extension of their work with SMARTS, the authors presented TurboSMARTS[41] as a solution to alleviate the functional warming bottleneck. Right at the beginning of each sample they drop checkpoints that contain the state of the functionally warmed micro-architecture. As a result, the simulator only needs to restore a checkpoint, execute the detailed warm-up and sample, and repeat this process for the following sample. Note that after eliminating the functional warming, the total simulation time depends on the variability of the program (number of sample units) and not on the program length.

Storing the state of large structures on each checkpoint can be very costly in terms of disk space, specially when de caches are big. The problem intensifies in programs that require a large amount of samples, in some cases the size of the checkpoint set was in the order of tens of terabytes[41], however after some compressing the TurboSMARTS authors could reduce it to tens of gigabytes.

Even sizes of gigabytes are prohibitively big when dealing with benchmark suites that have many programs in it, furthermore, loading and uncompressing the checkpoints consumes significant amount of time. Following the ideas of MRRL or BLRL, Van Biesbrouck *et al.*[36] proposes a way to reduce the checkpoint size by storing only data that will be needed in the sample unit. Furthermore, they also reduce the

detailed warm-up time by storing the state of other micro-architecture structures. In particular they present two techniques, Touched Memory Image (TMI), which stores only the words of memory to be accessed in the sample and Memory Hierarchy State (MHS) which recreates the state of the major micro-architecture components (TLB's, BTB, register file, etc). Although originally conceived for Simpoint, these methods also are valid for frameworks like TurboSmarts. Figure 4.1 shows an illustration of the most accepted simulation sampling techniques for single-threaded programs.

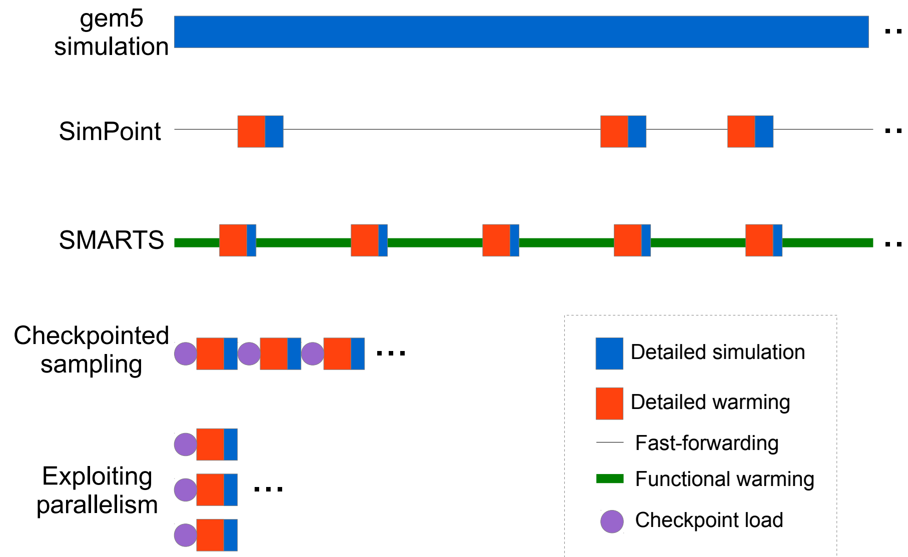

Figure 4.1: Sampling techniques for single-threaded programs.

Van Biesbrouck[38] goes beyond the sampling simulation of single-threaded uniprocessor systems stepping into the Simultaneous Multi-Threading (SMT) simulation area; the main questions his work tries to answer is: where to sample?. Every program has its own behaviour and variability that require it to be sampled at

different points or with different frequencies than others. When two or more programs execute concurrently in the same processor the solution is not trivial and cannot be treated as the superposition of all threads because they share and compete for the same hardware resources. Van Biesbrouck proposes to analyse with Simpoint separately each program in order to find its phases, then a *co-phase matrix* will store the IPC and length of all the combinations of individual thread phases to approximate the IPC and estimate the next sampling point.

4.1.3 Sampling simulation for multi-threaded programs

All the previous methods were successfully tested in uniprocessor simulators, nevertheless, due inter-thread communication and synchronization, the properties of multi-threaded applications are hard to characterize and many of the sampling simulation methods may not hold for many-cores simulations.

Alameldeen[1] identified as a potential problem the differences between performance estimates of multiple runs of the same workload, and states that the variability seen in some simulations can lead to incorrect architectural conclusions. He proposes to do hypothesis and confidence interval tests over several runs of the same workload. This assumption is rarely considered in architectural simulation studies, specially because the simulators are deterministic and always output the same result for the same workload and system configuration, but this is not the case for sampled simulation. If two threads are competing for a shared element in memory right before the sampling unit begins, incomplete warm-up can cause one thread, that originally would lose the race, to win access to the shared element and lock it. Unlike in single thread simulations, these imperceptible changes affect the performance of many threads and can lead to a totally different behaviour of the system during the sample.

Later on, Alameldeen published a paper in which he argues that the IPC is not

a reliable metric of the performance of multiprocessor workloads[2]. Unlike single-processor workloads, due to synchronization mechanisms like idle loops, spin locks, or barriers, small timing variations can result in very different execution paths. Even though all these synchronization mechanisms change IPC, they have little effect on the amount of useful work done by the program. After providing some examples where the IPC is not necessary related to the performance of the system, Alameldeem proposes to use work-related metrics[6] to better measure the performance of the system. However, this makes things more complex because every program will have its own metric and would be hard to compare them fairly.

On another analysis regarding the special considerations for sampling multi-threaded applications, Carlson *et al.* pointed out the periodicity of the applications and the importance of correct sample selection to avoid aliasing, specially in cases where threads cannot be assumed to run independently[7]. Furthermore, they noted that monitoring per thread *non-idle IPC* (non-spinning) and simulating the inter-thread communications (like data sharing or barriers) while fast-forwarding, can help to increase the sample accuracy.

After all these studies that reveal the complexity of doing sampling simulation with multi-threaded applications, Hardavellas introduces SIMFLEX[16], a new simulation framework able to run multi-threaded programs that also implements the SMARTS technique. For single-threaded applications the sample unit selection is an accepted and straight-forward procedure, it is not the same case with multiprocessor programs that consist of multiple instructions streams with non-determinism among them. This makes hard to find a metric that approximates the relative progress of different processors. Following the Alameldeen's idea of measuring work-related metrics rather than IPC, the most important metric for multiprocessor systems is

[6]like number of transactions, processed pixels, or compressed data blocks per unit time

the total program run time, thus they focus on the execution along the critical path of the program.

In multi-threaded programs it is common to see some threads waiting in a barrier for other thread to move on, the program cannot execute faster if the latter thread is not completed. The critical execution path goes through parts of different threads and defines the fastest execution time of a program, processors off of the critical path do not contribute to the determination of overall execution time. To sample only on the processor where the critical path is currently in, ensures that the relative progress on each path is representative of the program. However, being able to sample according the critical path requires a previous analysis of the programs. Furthermore, due to the variability in multi-threaded programs that Alameldeen described, small changes on the system under testing can dramatically change the the critical path during simulation.

In a further work, Wenisch presents an extension to SIMFLEX for multi-threaded applications with an approach similar to TurboSMARTS[40]. That is, rather than fast-forwarding and functionally warming, they created *flex points* that store the contents of the micro-architecture and thus, avoid warming the structure up. Since SIMFLEX was first conceived as a server simulation framework, they tested throughput applications on it. Usually the performance of those applications is reported in terms of transactions per second. However, due to the amount of time it takes to simulate and the high coefficient of variation the transactions show, the transaction rate was not the best option for the simulation framework. Instead, they use as a metric the number of user-mode instructions per transaction, which is proportional to transaction throughput but with lower variance.

The main problem in sampled simulation for multi-threaded workloads is to make sure that all threads are aligned at the beginning of each sampling unit.

BarrierPoint[8] exploits the fact that global synchronization barriers represent a common point in time for all threads, and therefore are safe points for checkpointing. BarrierPoint applies to multithreaded applications the same methodology that SimPoint applies to their single-threaded counterparts with one big difference: the simulation points are no longer defined by fixed instruction blocks, but by all the instructions in between two global barriers. Thus, BarrierPoint collects data signatures to determine the most representative inter-barrier regions.

All the sampling simulations previously proposed had to functionally simulate the program at least once, for example, they had to run a functional simulation to reach the sampling units and either switch to detailed simulation or create a checkpoint. If the samples are few but representative the functional simulation represents a bottleneck in the simulator performance. To overcome this problem Falcón *et al.*[14] propose a technique that combines dynamic sampling with virtualization and allows to run applications over emulated hardware at near-native speed. Thus, instead of fast-forwarding, they save time by running the program in virtual hardware. The virtualization tool constantly monitors the system's metrics of interest (instructions executed, exceptions, memory requests, etc.) and whenever it detects a significant change in the behaviour of those metrics, it communicates with a cycle-accurate simulator which starts the warm-up and sampling process. It is worth noting that unlike SimPoint or SMARTS this technique does not specify the simpoints or sampling period before the simulation begins, it makes on-the-flight decisions of where to sample based on the program behaviour. However there is a high correlation with the parts of the program that SimPoint would have chosen.

Due to its highly non-deterministic behaviour and inter-thread communication nature, sampled simulation on multi-threaded applications is specially hard to achieve. When SMARTS and SimPoint were first presented for single-threaded applications,

the research community quickly accepted them and all the research focused on just optimizing the same original idea. On the other hand, several techniques for multi-threaded applications have been proposed, all of them achieving great speed-ups but also proposing substantially different approaches and metrics. Today, there is no common agreement between researchers and it seems that the election of the sampling simulation technique for multi-threaded applications depends on the variable under study, the simulation platform and the benchmark suite to be used.

4.2 Potential speed-up in gem5

Not all of the sampling techniques described in the prior section are adequate to be implemented in gem5, specifically we are looking for a low-overhead solution compatible with any kind of program. To better determine the optimal sampling technique for gem5, it is necessary to go through gem5's main features and know what capabilities for sampling already supports.

All the objects within a memory system are connected to each other by ports that transmit requests and responses. In particular, there are three types of accesses supported by the ports:

- **Timing**: The most detailed type of access, it simulates realistic timing and models the queuing delay and resource contention. When a timing request is successfully sent after some time the device that sent the request will get a response. Timing and atomic accesses cannot coexist in the same memory system.
- **Atomic**: Atomic accesses are faster but less detailed than timing accesses, for that reason they are useful for fast-forwarding and warming up caches. When an atomic access is sent, the response is provided immediately. Atomic and timing accesses cannot coexist in the same memory system.

- **Functional**: These accesses are mainly used when a remote debugger is attached to the simulator and not for simulation purposes, thus, this type of access will not be considered in further discussions in this work. However, it is worth noting that the functional accesses occur immediately and can coexist with atomic or timing accesses in the same memory system.

The gem5 simulator supports two different memory system models: *Classic* and *Ruby*. The Classic memory supports atomic and timing accesses and is a faster than Ruby, which makes it advantageous when one needs to fast-forward to a given part of the execution. It maintains coherence through an abstract snooping protocol and only requires small modifications in the python script to create and arbitrary memory hierarchy, however more deep modifications to the model require significant effort. It relatively lacks of accuracy because it does not model transient states and protocol contention as accurately as Ruby.

On the other hand, Ruby sacrifices simulation speed and provides an infrastructure to accurately model cache coherence and network related features in the memory system, hence, it is only compatible with timing accesses. Ruby supports a domain specific language called SLICC (Specification Language for Implementing Cache Coherence) that allows the definition modification of the cache hierarchy and coherence protocol in a relatively easy way.

Gem5 supports also four different types of CPU's that vary on simulation detail and execution time:

- **Atomic**: The fastest and pure functional model, ideal for cases where simulation time is a constrain and no detail is needed (like fast-forwarding or warm-up periods). This is the only CPU that uses atomic memory accesses.Among other features, it holds the architected state, sets up fetch requests, advances the PC,

implements functions for read/write memory and handles pre-execute setups and post-execute actions.

- **Timing**: It is also a functional model but unlike the atomic model, it uses timing memory accesses only. In addition to the features present in the atomic model, it also does other actions like stalling the execution on cache access and waiting for the memory system to respond before proceeding.

- **Detailed-O3**: This is a time accurate out-of-order CPU model. Simulates six pipeline stages and other auxiliary structures like the branch predictor, functional units, reorder buffer or load/store queue.

- **InOrder**: This model has almost the same features that the O3 model, being the big difference that it only simulates an in-order pipeline.

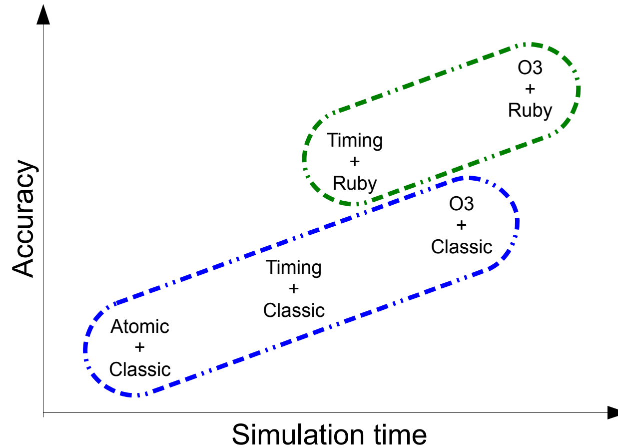

Figure 4.2: Summary of all possible combination of models in gem5.

Figure 4.2[7] depicts all the possible combinations of models in gem5, the dashed shapes signal the combination of models that are compatible with the same type of memory access. For example, since the Classic memory model is compatible with all kinds of memory accesses, it can be used with the all three CPU models. On the other hand, Ruby does not support atomic accesses, hence it is only compatible with the timing and O3 model.

At the moment of this work, gem5 supports fast-forwarding through an application using a fast CPU model, when the simulation reaches a specific number of simulated instructions it switches the CPU model to the most detailed one and continues until its end. This option only allow us to save some time in the first stages of the simulation process, however, the remaining simulation time is still large and it does not switch back to the fast model once the region of interest of the application has passed.

Checkpointing is well supported in gem5, it even provides a warm-up period after which the statistics are cleared and the sample begins. It is also possible to exploit parallelism by restoring multiple checkpoints at the same and run each simulation in different threads. Nevertheless, the goal is to find a low overhead sampling simulation technique and the amount of memory consumed by the checkpoints discards this as an option.

There is an option that makes gem5 switch back and forth between two CPU models with a period specified by the used and 50% of the instructions are simulated in each model. Although this option reduces the simulation time, it does not exploit all the available acceleration, provide a warm-up period after each switch or let the user choose the size of the sampling unit.

[7]The Functional will not be considered in this work because it is only useful for debugging purposes and we are interested in cycle-accurate simulation memory model. InOrder CPU model is also out of scope of this work because most of the modern processors support out-of-order execution.

However, with few modifications to the simulator, the user could be able to specify the period, warm-up and sample size and do periodic sampling like in SMARTS.

Note that SimPoint and all the derived techniques like BarrierPoint can also be implemented with this approach. The programs would need to be analysed off-line and specify the beginning of each simulation point rather than a sampling period.

Before explaining the implementation details, it is important to know the potential speed-up this technique may achieve. Since the goal is to get samples of the most detailed model, according to figure 4.2 the only choice we have is to fast-forward using Timing+Ruby and sample with O3+Ruby. If we consider a perfect simulation framework where there is no need to warm-up the and the number of instructions in the samples is negligible compared with the total number of instructions, then the resulting simulation time will be dominated by the time spent fast-forwarding between samples. Figure 4.3 shows a comparison of the simulation times for the simsmall input set of the Parsec benchmark suite. It is possible to get an maximum average speed up of 3.29×, which is not as big as other works present but is the best we can get given the overhead and accuracy constraints.

With this approach caches are kept warm and as a consequence, the required warm-up time is expected to be small, this resembles the functional warming of SMARTS. However, the cache image at the beginning of the warm-up is not the same as it would be if no sampling were happening. The reason after that are the differences between the CPU models used to sample and fast-forward.

The CPU model used in the fast-forward stage only does functional simulation of the instructions. On the other hand the model used in the samples should simulate the pipeline stages, out-of-order execution and speculative instructions. All the store-instructions within a speculative branch will not be committed until the branch resolution. However, that is not the case with load-instructions which are injected

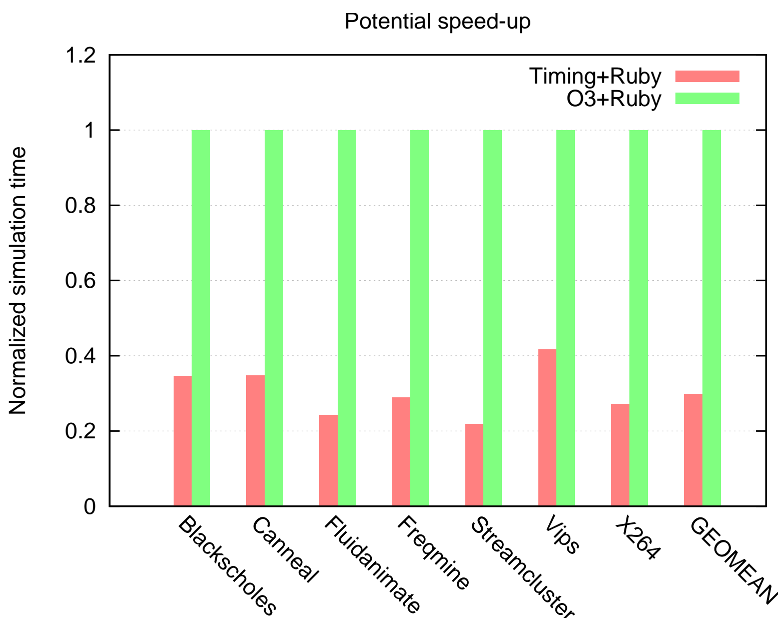

Figure 4.3: Potential speed-up in the Parsec suite.

into the memory hierarchy regardless the speculation is right or not.

Pierce[28] studied how the execution of speculative instructions pollutes the cache and increases the memory bus traffic. Armstrong[3] analysed the effects of what he called the wrong-path events: events generated in an out-of-order machine when instructions following a mispredicted path are speculatively executed before the branch resolution. In [26] the authors showed that rather than diminishing the performance due to cache pollution, speculative execution prefetches blocks that will be used later by future instructions. Actually, they concluded that not considering the effects of speculative instructions in the memory hierarchy can underestimate the performance of the system.

Even though the memory hierarchy is simulated all the time, since the functional CPU model does not simulate speculative paths, there will be some differences in the cache state at the beginning of the warm-up period. Thus, the cache hierarchy also

needs to be warmed-up before the sample. However, warming-up the caches with speculative instructions is much faster than warming-up an empty cache hierarchy.

4.3 Switching CPU models

I extended the gem5's capability to switch between CPU models so it can appropriately perform the chosen sampling simulation methodology. In specific the user is only required to specify the two CPU models to switch between, as well as the number of instructions that compose the desired sampling period T, warm-up length W and sample unit size U. Thus, after $T-(W+U)$ instructions the simulator stops and switches the CPU models, it continues simulating in detailed mode and starts measuring the sample after W instructions. After U instructions it again stops the simulation and switch back to the first CPU model. This sampling process is only active in the Region of Interest (ROI) of the application, other parts of the program are just fast-forwarded.

Switching CPU models is not straight forward, it requires a deep understanding of the simulated micro-architecture and the draining process, actually is in this part where I faced most of the bugs.

Figure 4.4 shows the four main steps in the switching process, in particular when it switches from the O3 to Timing models. The first step is to stall and drain all the internal structures of the CPU in the same way they would have been drained due to erroneous speculation. One of the structures that take longer to drain is the load/store queue because it needs to wait for the memory system to respond those requests. If the system is not drained, the simulator keeps ticking to allow the memory system attend the pending requests. Deadlocks can occur here if an specific condition avoids any structure to drain.

Once the CPUs are drained they are switched out by disconnecting the cache

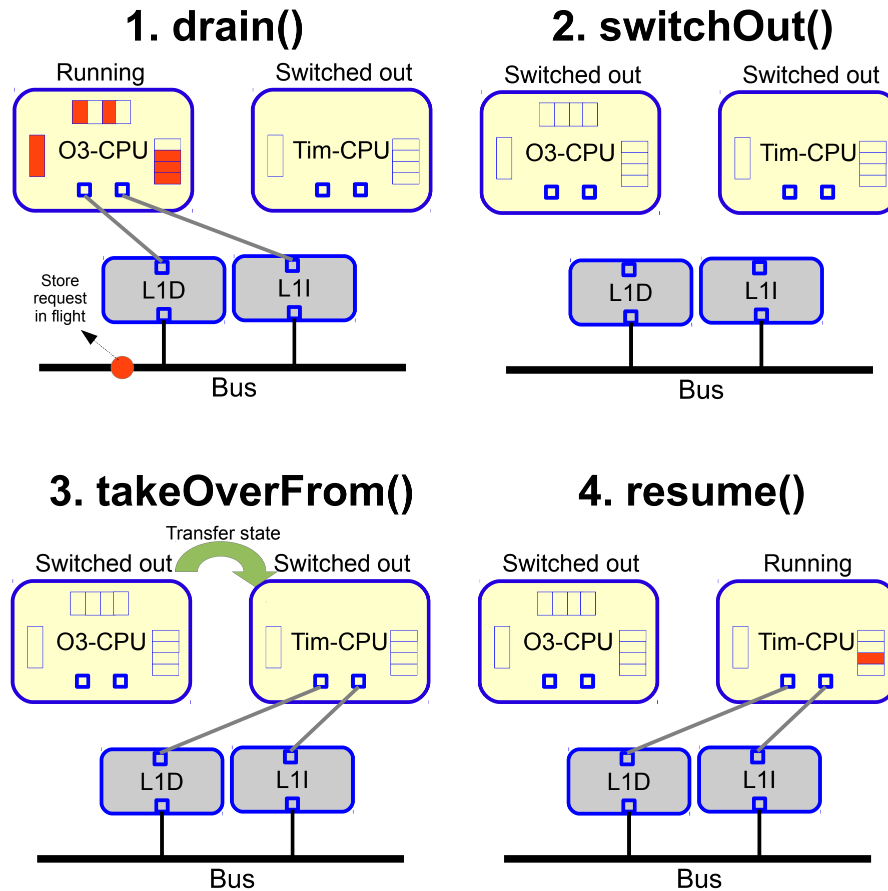

Figure 4.4: Process of switching CPU models.

ports, then the state (specially the PC and register file) is migrated from one model to other and the cache ports are attached to the second CPU model. At this point the simulation is resumed. As mentioned before, the caches are never switched out and hold their state at all times.

At the end of the simulation the system prints out the statistics of interest for each core as well as the mean of the statistics gathered throughout all the samples. It also shows the number of sample units gathered, the variation coefficient and the

confidence interval among the samples with confidence levels of 95% and 99%. If the resulting confidence interval is not the desired for a given confidence level, the user should run again the simulation with a bigger sample size defined by the equation 4.1.

$$n \geq \left(\frac{z \cdot \hat{V}_{cpi}}{\epsilon} \right)^2 \tag{4.1}$$

Where, n is the minimum sample size to required to satisfy the the confidence interval constraints. $\hat{V}_{cpi}$ is the variation coefficient calculated in the previous simulation. z is the $100[1-(\alpha/2)]$ percentile of the standard normal distribution for a given confidence level, for confidence levels of 95% and 99%, $z = 1.96$ and $z = 3$ respectively. ϵ is the percentage of the sample's mean that represent the desired confidence interval, in other words, the desired confidence interval of a variable X is defined as $\pm\epsilon \cdot \bar{X}$. In practice, the required sample size n can typically be found after one test sample.

The variation coefficient $\hat{V}_{cpi}$ is different for every program, and thus, there can be programs that can be represented with a small number of samples while others require to sample many times more. Figures 4.5 and 4.6 show how the CPI varies for blackscholes and vips during the simulation of the Region-of-Interest. On one side, blackscholes is an example of and extremely homogeneous application which show small variation on the CPI. On the other side, vips is a very variable program which requires more samples to represent its behaviour.

4.4 Optimal period and sample size

The starting point of this methodology is based in the SMARTS[42] work where relatively small sample units (U) and warm-up times (W) are proposed[8]. Several tests were done in order to determine the optimal value of U and W, hoping to

[8] 1000 and 4000 instructions, respectively

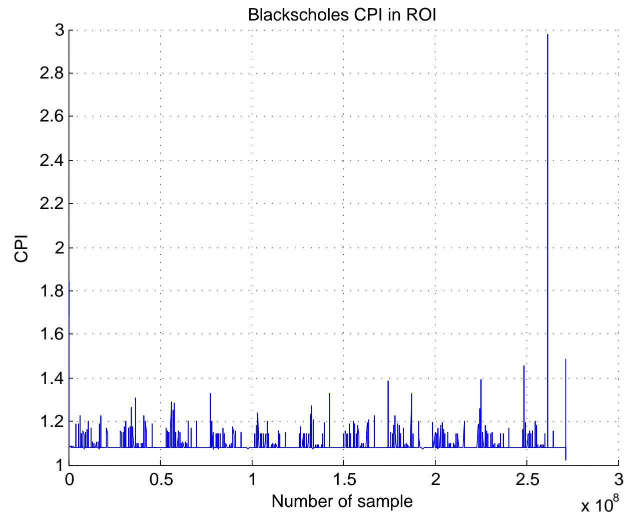

Figure 4.5: CPI of blackscholes throughout the ROI.

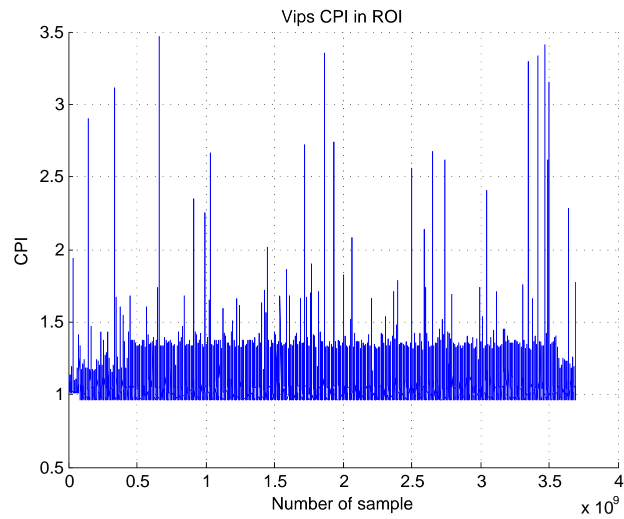

Figure 4.6: CPI of vips throughout the ROI.

find a point in which all the benchmarks perform with the least CPI error. I swept the benchmarks through three different warm-up times and three sample unit sizes

that I considered appropriate for a 16-core machine. The results of the tests with sample sizes of 32k, 64k and 128k instructions are shown in figures 4.7, 4.8 and 4.9, respectively. Among all, there are three benchmarks (Streamcluster, Vips and X264) that show an irregular behaviour and their CPI error does not have a straight forward relation with U or W, besides those, the CPI error for the remaining benchmarks is below 2%.

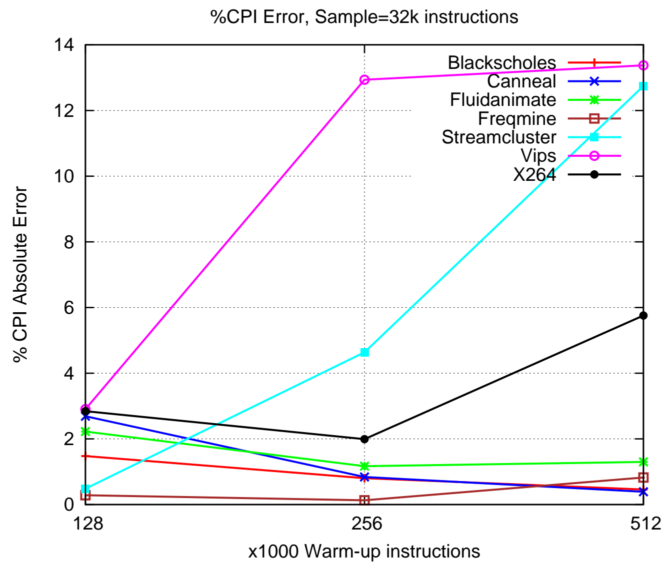

Figure 4.7: CPI percentage error for W when U=32k.

Going further on the search, I chose two of the conflicting benchmarks (Streamcluster & X264) and ran the simmedium input set for several sample sizes. I selected a small enough sampling period to sample at least 100 times and reduce as much as possible sampling errors. Also, in order to discard the chances of bias due to cold start, I set the warm-up time to 512000 instructions. The plot in figure 4.10 shows the changes in the measured CPI error while figure 4.11 plots the variation coefficient of the CPI.

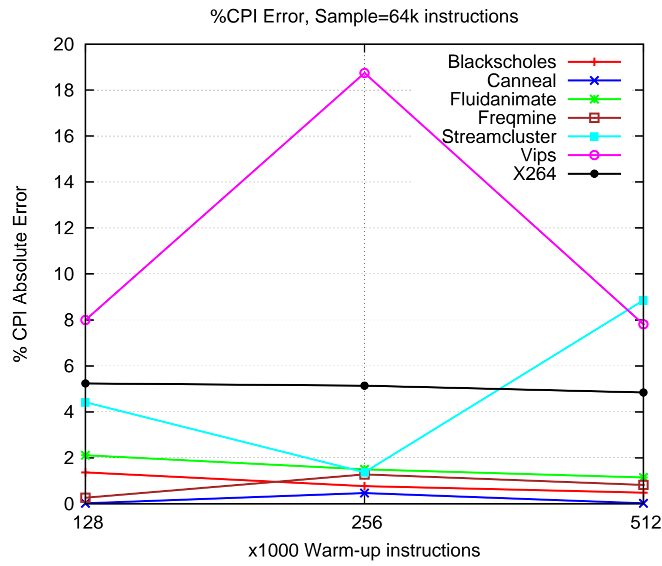

Figure 4.8: CPI percentage error for W when U=64k.

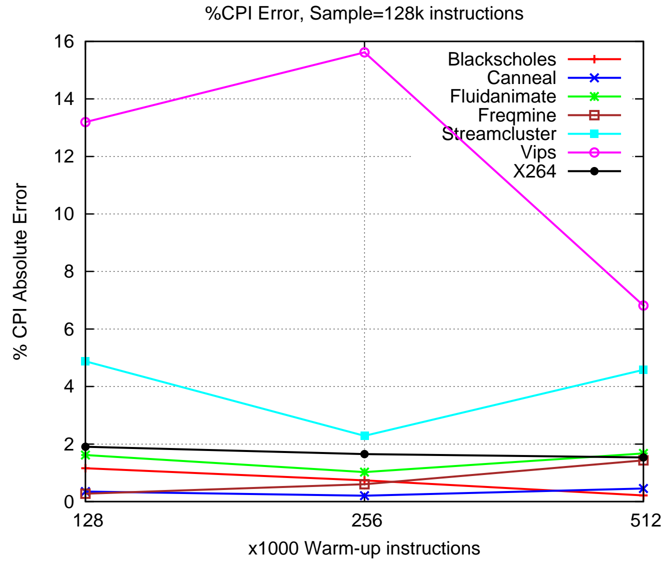

Figure 4.9: CPI percentage error for W when U=128k.

The results in figure 4.11 are expected. The bigger the sample size, more instructions are accounted and particular variations in the sample get averaged by the

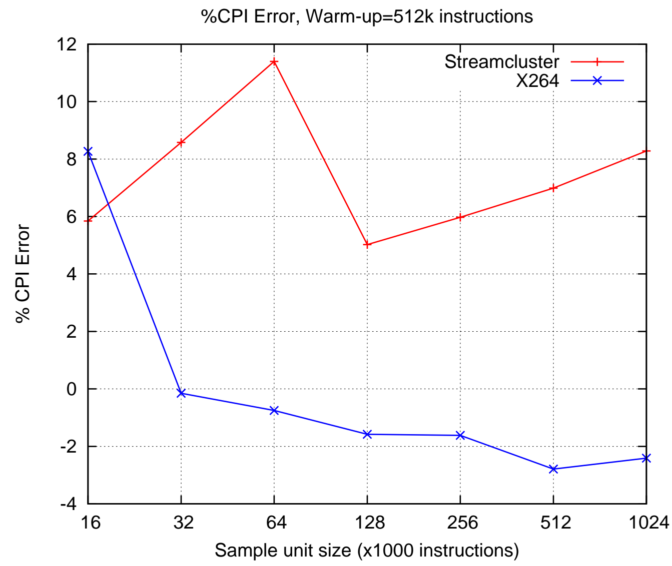

Figure 4.10: % CPI error, W=512k.

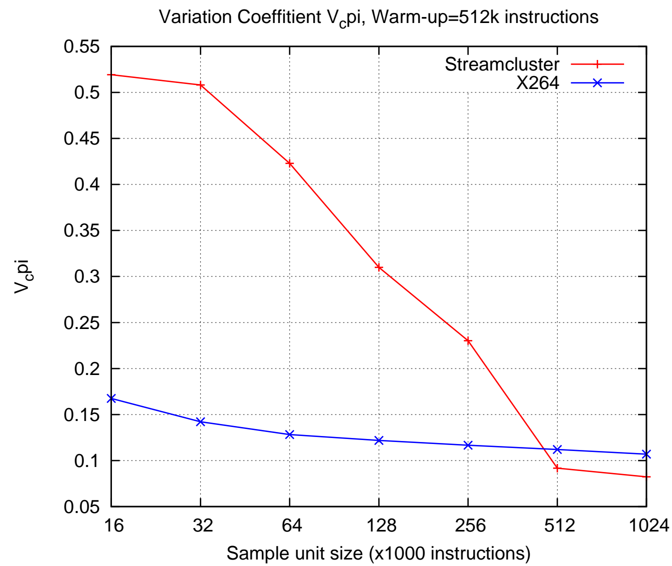

Figure 4.11: Variation coefficient, W=512k.

rest of the sample. In other words, big samples are more homogeneous and thus, the variation coefficient decreases. As the homogeneity in the samples becomes ev-

ery time more evident, the sampled CPI is expected to approach the application's real CPI. Nevertheless, figure 4.10 shows a totally different behaviour. From both plots we can conclude that somehow, the sampling technique introduces bias in the measurements.

The most probable source of bias is the branch prediction mechanism. To prove it, I simulated only one sample in detailed mode and compared it with exactly the same section of the detailed trace (without switching CPU models). Results showed big difference in the hit rate of the branch target buffer. Undoubtedly, the switching process and warm-up time change the interaction between the treads and also the program phases under measurement.

4.5 Sampling process

The goal of this section is to find the sampling parameters that guarantee the sampled CPI to be within a confidence interval of 5% of the real application CPI. We start by proposing an arbitrary sampling period of $T = 4 \times 10^7$ instructions, a sample unit size equal to $U = 64000$ instructions and a warm-up interval of $W = 256000$ instructions. Table 4.1 shows a summary of the simulations after the first sampling round. From left to right, the second column shows the number of sampling units collected with a given T, the next column to the right contains the percentage of error from the sampled CPI. The forth column from left to right shows the confidence interval of the sample with confidence level of 99%. *Nopt* refers to, according to the variation coefficient of the sample, the minimum number of sample units to get a confidence interval equal or less than 5%. Finally, the rightmost column indicates whether another sampling round is needed to get the desired confidence interval or not.

In the first round the only benchmarks that satisfied the confidence interval re-

Table 4.1: Summary of the 1st sampling round

Benchmark	N	% CPI Error	% CI/99%	Nopt	Additional round
Blackscholes	30	0.631155	0.8098	1→30	No
Canneal	19	0.637552	6.13159	39	Yes
Fluidanimate	103	1.531162	2.56686	62	No
Freqmine	325	0.657079	7.47832	725	Yes
Streamcluster	156	4.918889	9.27065	1216	Yes
Vips	290	11.448302	7.6273	657	Yes
X264	33	3.433865	12.1576	144	Yes

quirement are blackscholes and fluidanimate. Blackscholes is an example of an extremely regular program, that is why the sampling technique calculated that only one sample is enough to fully represent the program. Actually, that assumption is true for blackscholes, but in order to be statistically correct, the central limit theorem requires to have at least a sample size of 30[9].

Table 4.2: Summary of the 2nd sampling round

Benchmark	N	% CPI Error	% CI/99%	Nopt	Additional round
Canneal	39	0.947153	4.180664	27→30	No
Freqmine	725	1.161099	5.00128	725	No
Streamcluster	1216	5.930528	6.36437	1998	Yes
Vips	661	2.912759	5.01486	662	No
X264	144	0.748005	5.01912	144	No

After the second sampling round, streamcluster is the only program that had a confidence interval greater than 5% and needs a third simulation round, which results are shown in table 4.3.

The resulting CPI error for streamcluster is greater than the confidence interval, this is due to the bias problem explained in the previous section. Since, according to

Table 4.3: Summary of the 3rd sampling round

Benchmark	**N**	**% CPI Error**	**% CI/99%**	**Nopt**	**Additional round**
Streamcluster	1998	7.940398	4.61347	1701	No

the nature and number of sample units, the confidence interval is already less than 5%, there is no statistical justification to sample more frequently aiming to reduce the error. The same phenomena happened in the first round of vips, but it got fixed in the next round. According to my experience running several tests, streamcluster and vips are the most likely programs to have this bias problem and happen to be also the programs with more irregular behaviour in figures 4.7, 4.8 and 4.9.

4.6 Results

Eight benchmarks from the Parsec suite were run in a 16-core simulation environment using the simmedium input set[9]. Figure 4.12 shows the CPI obtained for each benchmark after running the simulations with the minimal sample size required to get a confidence interval of 5%. Actually, the error bars located at the top of the bars represent the $(0.95 * CPI_{det}, 1.05 * CPI_{det})$ interval, where CPI_{det} is the CPI measured from a pure detailed simulation.

As expected, due to the previously explained bias problem, streamcluster is not in between such interval. On the other hand, figure 4.13 shows the percentage of the CPI error along with the confidence interval resulting from the sample size and the variation of the sample units. As table 4.3 first mentioned, the error in streamcluster is bigger than its confidence interval. Nevertheless, even with the great contribution of streamcluster, the average CPI error along all the benchmarks is 2.267%.

Figure 4.14 shows the reduction in simulation time that implies doing sampling

[9]The remaining benchmarks had compatibility issues with one of the CPU models, independent from the implementation of the sampling simulation framework

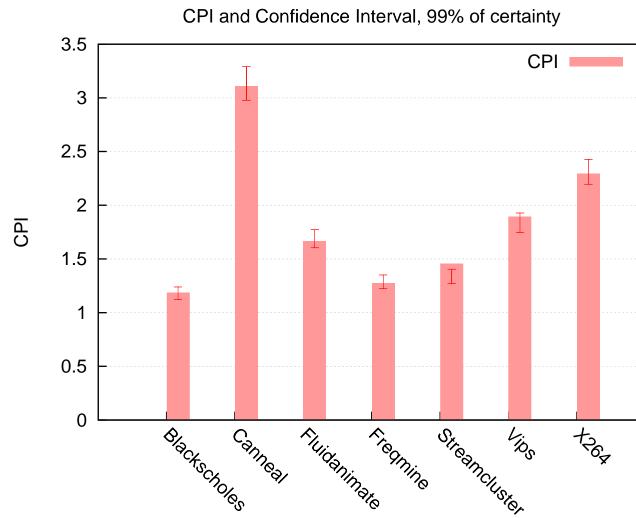

Figure 4.12: Sampled CPI and interval of ± 5% of the pure detailed CPI

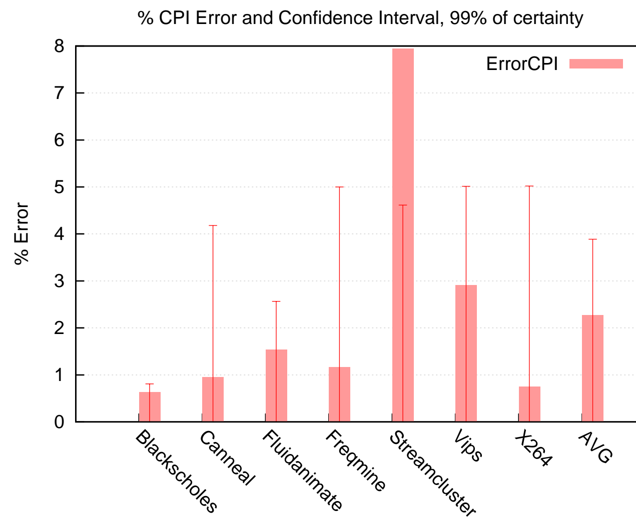

Figure 4.13: Percentage CPI error and confidence intervals with confidence level of 99%

simulation against the purely detailed simulation. The time offered is very close to that offered by the timing simulation, which is the upper bound in acceleration. On average, compared with the detailed simulations, the timing and detailed simulation

take 30.35% and 35.63% of the time, respectively. This means that if the ROI's of these eight benchmarks were simulated one right after the other, the sampled simulation would finish 8.22 days before than the detailed and just 16.68 hours after the timing.

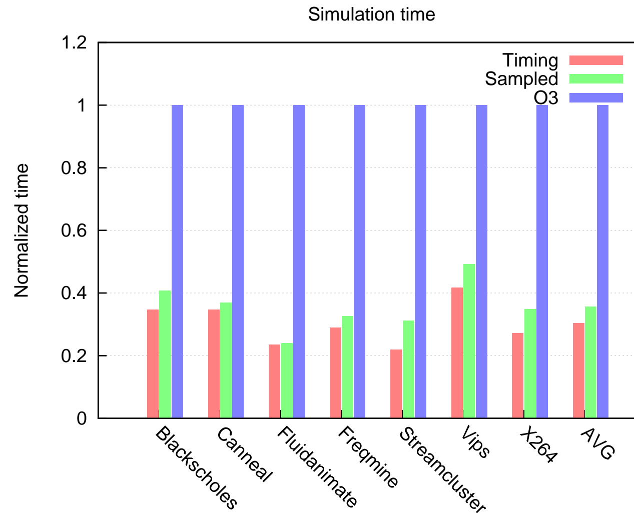

Figure 4.14: Speed-up of sampling compared with the pure timing and detailed simulations

Table 4.4 shows the percentage of instructions simulated in each of the stages existent in sampling simulation. Note that great majority of the instructions are fast-forwarded and, except for streamcluster, less than 1% of the instructions are actually sampled.

Finally, table 4.5 aims to help to replicate this work or give a starting point to those willing to use sampling simulation with the simmedium Parsec's input set. The table shows the maximum recommendable sampling period to achieve a confidence interval of 5% using sample units of 64000 instructions and warm-up periods of 256000 instructions in a system simulating 16 cores.

Table 4.4: Percentage of instructions spent in each stage

Benchmark	Fast-forwarding(%)	Warm-up(%)	Sample(%)
Blackscholes	99.2261	0.6190	0.1547
Canneal	98.3710	1.3032	0.3258
Fluidanimate	99.2499	0.6000	0.1500
Freqmine	98.2229	1.4217	0.3554
Streamcluster	89.8353	8.1318	2.0329
Vips	98.1814	1.4549	0.3637
X264	96.3892	2.8887	0.7221

Table 4.5: Recommended sampling period for W=256000 and U=64000 instructions

Benchmark	ROI Length	N	Period Length (T)
Blackscholes	1240612937	30	41353764
Canneal	766130081	39	19644361
Fluidanimate	4393967919	103	42659882
Freqmine	13055140728	725	18007090
Streamcluster	6289991367	1998	3148143
Vips	11648247326	662	17595539
X264	1276155295	144	8862189

5. CONCLUSIONS

Full-system cycle-accurate simulators are the most reliable tool in computer architecture research. As simulators implement current micro-architecture techniques they become more complex and the simulation time increases. This work proposes a solution to make the simulators more alike to current hardware designs and decrease the simulation overhead at the same time. In particular, a third level cache hierarchy as well as statistical sampling simulation were implemented in a current full-system simulator in order to make the simulation more accurate with less runtime.

A new cache hierarchy for multi-core systems along with its corresponding coherence protocol were presented. The protocol was validated and performed 1 billion random memory accesses with 64 cpu's. Data taken from cycle-accurate simulators show that a system with the proposed three level cache improves the performance around of 30% compared with a baseline system with only two cache levels.

Statistical sampling was used to speed-up the simulation of multi-threaded benchmarks. Results show an average measured CPI error of less then 2.5% and a speed-up of around 3x compared to the time needed to run a detailed simulation of the entire benchmark. In most of the cases, it was necessary to sample lees than 1% of the instructions to get the desired confidence interval. This work presents a table with the minimum number of samples (and sampling period) required to get results within a confidence interval of 5% and confidence level of 99%.

Apparently, sampling simulation is not the most appropriate technique for some multi-threaded programs. Small changes in the system environment can cause some threads to win races over others, changing the behaviour of the program and introducing bias in the measurements. Unfortunately the switching and warm-up process

modify the threads interaction and underestimate the measurements.

The drawback of SMARTS is that the time spent in the fast-forward stage limits the maximum achievable speed-up. Solutions like using virtual hardware or co-simulating different parts of the system promise to shrink the fast-forward time without requiring excessive disk usage.

REFERENCES

[1] A.R. Alameldeen and D.A. Wood. Variability in architectural simulations of multi-threaded workloads. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 7–18, Feb 2003.

[2] A.R. Alameldeen and D.A. Wood. Ipc considered harmful for multiprocessor workloads. *Micro, IEEE*, 26(4):8–17, July 2006.

[3] D.N. Armstrong, Hyesoon Kim, O. Mutlu, and Y.N. Patt. Wrong path events: Exploiting unusual and illegal program behavior for early misprediction detection and recovery. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 119–128, Dec 2004.

[4] 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*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[6] Jason F Cantin, Mikko H Lipasti, and James E Smith. Dynamic verification of cache coherence protocols. In *High Performance Memory Systems*, pages 25–42. Springer, 2004.

[7] T.E. Carlson, W. Heirman, and L. Eeckhout. Sampled simulation of multi-threaded applications. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 2–12, April 2013.

[8] T.E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout. Barrierpoint: Sampled simulation of multi-threaded applications. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 2–12, March 2014.

[9] Jay Devore. *Probability and Statistics for Engineering and the Sciences*. Cengage Learning, 2011.

[10] David L Dill. The mur ϕ verification system. In *Computer Aided Verification*, pages 390–393. Springer, 1996.

[11] D.L. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang. Protocol verification as a hardware design aid. In *Computer Design: VLSI in Computers and Processors, 1992. ICCD '92. Proceedings, IEEE 1992 International Conference on*, pages 522–525, Oct 1992.

[12] Lieven Eeckhout, Yue Luo, Koen De Bosschere, and Lizy K John. Blrl: Accurate and efficient warmup for sampled processor simulation. *The Computer Journal*, 48(4):451–459, 2005.

[13] E. A. Emerson and A. P. Sistla. Utilizing symmetry when model-checking under fairness assumptions: An automata-theoretic approach. *ACM Trans. Program. Lang. Syst.*, 19(4):617–638, July 1997.

[14] A. Falcon, P. Faraboschi, and D. Ortega. Combining simulation and virtualization through dynamic sampling. In *Performance Analysis of Systems Software,*

2007. ISPASS 2007. IEEE International Symposium on, pages 72–83, April 2007.

[15] Limin Han, Jianfeng An, Deyuan Gao, Xiaoya Fan, Xianglong Ren, and Tao Yao. A survey on cache coherence for tiled many-core processor. In *Signal Processing, Communication and Computing (ICSPCC), 2012 IEEE International Conference on*, pages 114–118, Aug 2012.

[16] Nikolaos Hardavellas, Stephen Somogyi, Thomas F. Wenisch, Roland E. Wunderlich, Shelley Chen, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *SIGMETRICS Perform. Eval. Rev.*, 31(4):31–34, March 2004.

[17] Jr. Haskins, J.W. and K. Skadron. Minimal subset evaluation: rapid warm-up for simulated hardware state. In *Computer Design, 2001. ICCD 2001. Proceedings. 2001 International Conference on*, pages 32–39, 2001.

[18] Jr. Haskins, J.W. and K. Skadron. Memory reference reuse latency: Accelerated warmup for sampled microarchitecture simulation. In *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on*, pages 195–203, March 2003.

[19] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[20] C Norris Ip and David L Dill. Verifying systems with replicated components in murϕ. In *Computer aided verification*, pages 147–158. Springer, 1996.

[21] Thierry Lafage and André Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the

data stream. In *Workload characterization of emerging computer applications*, pages 145–163. Springer, 2001.

[22] Yue Luo, L.K. John, and L. Eeckhout. Self-monitored adaptive cache warm-up for microprocessor simulation. In *Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004. 16th Symposium on*, pages 10–17, Oct 2004.

[23] KL McMillan and James Schwalbe. Formal verification of the gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 111–134, 1992.

[24] A. Meixner and D.J. Sorin. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. *Dependable and Secure Computing, IEEE Transactions on*, 6(1):18–31, Jan 2009.

[25] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan 1998.

[26] O. Mutlu, Hyesoon Kim, D.N. Armstrong, and Y.N. Patt. An analysis of the performance impact of wrong-path memory references on out-of-order and runahead execution processors. *Computers, IEEE Transactions on*, 54(12):1556–1571, Dec 2005.

[27] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. pages 348–354, 1984.

[28] J. Pierce and T. Mudge. The effect of speculative execution on cache performance. In *Parallel Processing Symposium, 1994. Proceedings., Eighth International*, pages 172–179, Apr 1994.

[29] Fong Pong and Michel Dubois. A new approach for the verification of cache coherence protocols. *Parallel and Distributed Systems, IEEE Transactions on*, 6(8):773–787, Aug 1995.

[30] Fong Pong and Michel Dubois. Verification techniques for cache coherence protocols. *ACM Comput. Surv.*, 29(1):82–126, March 1997.

[31] R. Rodrigues, I. Koren, and S. Kundu. A mechanism to verify cache coherence transactions in multicore systems. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2012 IEEE International Symposium on*, pages 211–216, Oct 2012.

[32] A. Ros, M.E. Acacio, and J.M. Garcia. A direct coherence protocol for many-core chip multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 21(12):1779–1792, Dec 2010.

[33] John P. Shen and Mikko H. Lipasti. *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[34] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. pages 45–57, 2002.

[35] S Thoziyoor, N Muralimanohar, JH Ahn, and NP Jouppi. Cacti 5.3. *HP Laboratories, Palo Alto, CA*, 2008.

[36] Michael Van, Brad Calder, and Lieven Eeckhout. Efficient sampling startup for simpoint. 2006.

[37] Michael Van Biesbrouck, Lieven Eeckhout, and Brad Calder. Efficient sampling startup for sampled processor simulation. In *High Performance Embedded Architectures and Compilers*, pages 47–67. Springer, 2005.

[38] Michael Van Biesbrouck, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on - ISPASS*, pages 45–56, 2004.

[39] D. Vantrease, M.H. Lipasti, and N. Binkert. Atomic coherence: Leveraging nanophotonics to build race-free cache coherence protocols. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 132–143, Feb 2011.

[40] T.F. Wenisch, R.E. Wunderlich, M. Ferdman, A Ailamaki, B. Falsafi, and J.C. Hoe. Simflex: Statistical sampling of computer system simulation. *Micro, IEEE*, 26(4):18–31, July 2006.

[41] Thomas F. Wenisch, Roland E. Wunderlich, Babak Falsafi, and James C. Hoe. Turbosmarts: Accurate microarchitecture simulation sampling in minutes. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, pages 408–409, New York, NY, USA, 2005. ACM.

[42] R.E. Wunderlich, T.F. Wenisch, B. Falsafi, and J.C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 84–95, June 2003.

[43] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. Statistical sampling of microarchitecture simulation. volume 16, pages 197–224, New York, NY, USA, July 2006. ACM.