A CYCLE-ACCURATE SIMULATION INFRASTRUCTURE FOR CACHE-COHERENT INTERCONNECT ARCHITECTURES

A CYCLE-ACCURATE SIMULATION INFRASTRUCTURE FOR CACHE-COHERENT INTERCONNECT ARCHITECTURES

BY

SALAH HESSIEN, MASc.

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

Master of Applied Science (2020)	McMaster University
(Electrical & Computer Engineering)	Hamilton, Ontario, Canada

TITLE:	A Cycle-Accurate Simulation Infrastructure for Cache-Coherent Interconnect Architectures
AUTHOR:	Salah Hessien MASc. (Electrical & Computer Engineering), McMaster University, Hamilton, Canada
SUPERVISOR:	Dr. Mohamed Hassan
NUMBER OF PAGES:	xviii, 130

Abstract

Maintaining exponential growth in performance of computing systems is no longer derived by the advancement in technological metrics such as clock speed and transistor scaling due to the saturation of Moore's law. Therefore, architectural innovations are a crucial solution in order to maintain this growth. However, these innovations highly demand for comprehensive simulation tools since they provide an infrastructure for evaluating and prototyping new design ideas. Thus, this thesis introduces CacheSim, an efficient, extensible, and cycle-accurate simulator for cache-coherent interconnect architecture. CacheSim enables researchers and computer architects to build reconfigurable simulation infrastructure for multi-core processor chips with a high degree of flexibility of controlling system's configuration parameters such as cache organization, coherence protocol models, and interconnect bus architecture as well as bus arbitration policies. The primary motivation behind developing CacheSim is to use it for architectural explorations, study new design ideas, and evaluate existing ones. Throughout this thesis, we make the following contributions. First, unlike existing state-of-art simulators, we develop a complete cache coherence solution for multi-level cache hierarchy memory systems that support modern interconnecting multi-core bus architectures. Second, CacheSim provides these capabilities to the end-user not only without the need to modify the source code, but also with a high degree of configuribility to control the simulator behaviour through a well-defined input interface. Third, We use this tool to design a novel predictable and coherent bus architecture that provides a considerably tighter latency bound compared to the state-of-art predictable coherent solutions. Finally, we thoroughly validate the simulator features using directed and continuous regression testing plan and code coverage to ensure the functional correctness of the simulator. We release CacheSim as an open-source for the research community to extend and use. We expect CacheSim to significantly accelerate the design and testing of novel research ideas in cache coherency and predictable interconnect architectures used in real-time systems.

To my wife, Maha Sayed, for her endless love and sacrifices.

&

My beloved parents for their support and encouragement.

Acknowledgements

First and foremost, all praise is due to Allah alone (God), the almighty, who has granted me countless blessings, knowledge, and power to accomplish this thesis.

I would like to thank my supervisor Prof. Mohamed Hassan for his role as a great advisor and all the support he gave me during my study journey. Indeed, his guidance, encouragement, and patient were crucial to my academic success. I could easily say that I am very proud to be a member of his Fanous research group and really appreciate the support and freedom he gave me from day one to explore a topic that I can employ my passion and capabilities in.

I am also thankful to McMaster School of Graduate Studies (SGS) for funding my master's studies.

Finally, I would like to thank my beloved family for all the support and encouragement they gave me during my study, especially my wife, Maha Sayed, for all the tremendous effort and sacrifices she gives to me. I am not exaggerating when I say this thesis would not exist without your encouragement to me. To my parents for supporting my decision to move into a different country, and for their love, encouragement, and prays of day and night make me able to get such success and honor.

Contents

List of Figures

List of Tables

Notation and Abbreviations

Notation

-

Abbreviations

AvgL	Average latency
CMP	Chip Multiprocessors
EEMBC	Embedded Microprocessor Benchmark Consortium
ET	Execution time
C2C	Cache-to-Cache transfer
FSM	Finite state machine
ILP	Instruction-level parallelism
IO	In-order execution
IoT	Internet-of-things

LLC	Last Level Cache
MSI	Modify-Shared-Invalid protocol
OOO	Out-of-order execution
RR	Round robin
rua	Request under analysis
TLP	Thread-level parallelism
WCL	Worst-Case latency
VnV	Verification and Validation
MG	Module Guide

Chapter 1

Introduction

The end of Moore's law should enable a new cusp of another Golden Age in Computer Architecture that encouraging computer scientists to invent new architectures to improve cost, performance, and security demands of modern embedded systems applications. This thesis will introduce potential solutions addressing some of these challenges.

1.1 Motivation

Semiconductor technology scaling has resulted in performance improvements of microprocessors over the past decades. The advancement of semiconductor manufacturing process has led to increasing both the frequency at which the processors run and the transistor's integration density. The increasing transistor bounty has led to innovations in microprocessors architecture design. Many of these innovations are meant to increase the amount of parallelism of instruction processors. For instance, the early advancement of microprocessor architectures relied on innovations like RISC, instruction-level parallelism (ILP), pipelined out-of-order execution cores, superscalar, and multilevel caches plus speculation to improve performance. Although the aforementioned architecture changes have boosted the performance of microprocessors beyond technology scaling, they can no longer deliver the high-performance demands of modern real-time embedded system applications such as those deployed in automotive, avionics, and Internet-of-things (IoT) [2, 3]. Therefore, mainstream chip-makers have turned their attention to thread-level parallelism (TLP) by designing chips with multi-cores known as Chip Multiprocessors (CMP) to keep track of the high-performance demand. Nonetheless, multi-core architectures bring their own challenges. One of the biggest challenges is the interference among various cores in the system while competing to access shared hardware resources such as memory buses, shared caches, and off-chip memories [4]. This interference results in system predictability issues and prevents the system analyzability since the execution time of a task on one core now depends on the run-time behavior of tasks running on other cores. Furthermore, cache coherence is another fundamental issue in multi-core platforms, as many of today's CMPs designs employ hierarchies of caches to mitigate latency and bandwidth gap between processor and memory speed. For instance, standard memory system designs assume each core has its own private L1 cache and a shared L2 cache among all or multiple cores as shown in Figure 1.1. Consequently, this can lead to incoherent sharing of data while processors access different versions of the same data present in their private cache. CMPs usually implement cache coherence protocols to resolve the coherency issue, which adds another level of complexity to the system.

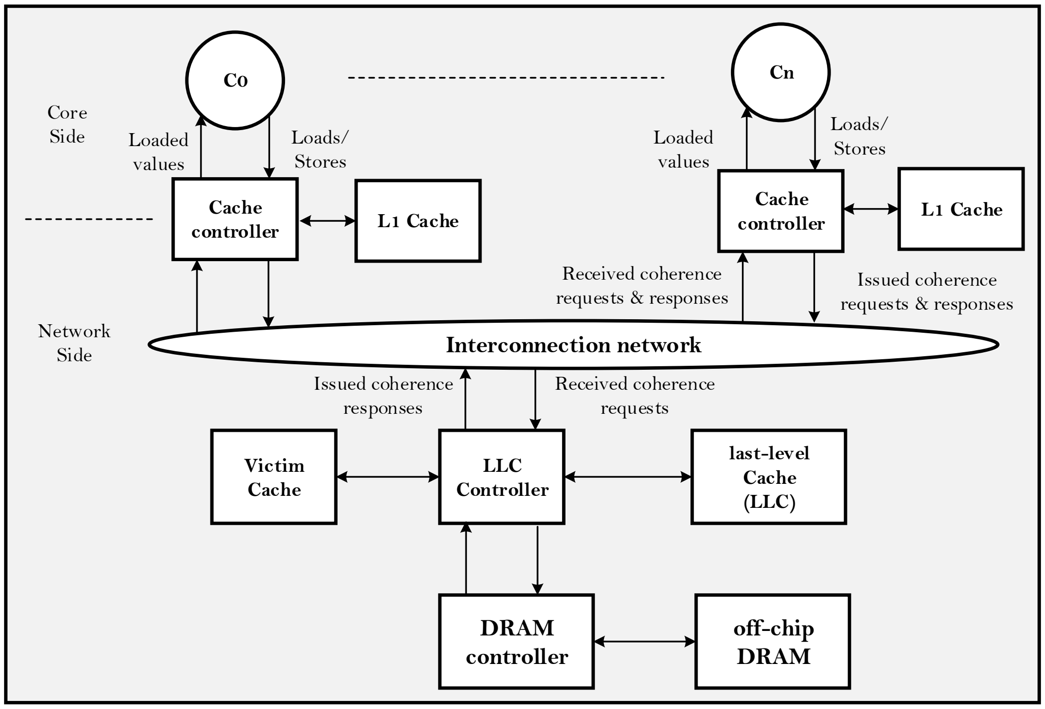


Figure 1.1: Multicore processor chip baseline system model

Moreover, maintaining exponential growth in technological metrics such as increasing clock rates and transistor counts became very challenging now. For instance, the newest Intel fabrication plant target for 10 nm technology node chip manufacturing was considerably delayed, delivered in 2019, five years after the previous generation technology of chips with 14 nm feature [5]. So, it becomes obvious now that we are on the cusp of a new Golden Age of computer architecture to track the high-performance demand of modern applications [6, 7]. Thus, in order to support these architecture innovations, computer system architects require accurate, extensible, and easy-to-use simulators to explore and prototype ideas.

In light of these challenges, this thesis focuses on 1) building an accurate and easy-to-use simulation tool that support simulating CMP environment with reasonably detailed microarchitecture modeling for cache hierarchy, coherence protocols, on-chip interconnect, and off-chip DRAM. The primary motivation behind developing this tool would be to use it for architectural explorations, studying new design ideas, and evaluating existing ones. 2) Then, we will use this tool to investigate the predictability and cache coherent memory access problems as those are now first-order design issues at the chip level in multi-core systems [1, 8, 9]. 3) Finally, we will show a practical use-case of the simulator where it can be used to develop a novel predictable and coherent memory access solution that provides considerably tighter latency bound compared to the existing state-of-the-art solutions [8].

1.2 Thesis Contributions

This thesis proposes CacheSim [10], an efficient, extensible, and cycle-accurate simulator for cache-coherent interconnect architectures. CacheSim is implemented in C++ using object-oriented programming concept with a high degree of configurablility to facilitate design space exploration of predictability and cache coherent memory issues raised in multi-core systems. The primary contributions of this simulator work are as follows:

- **Modularity:** CacheSim is based on modules and layers approach. Each module is coded in such a way that it can be constructed independently, and the encapsulated data is accessed through a well-defined interface. In this manner, changes to a particular block's behavior do not impact the other blocks in the system.
- **Expansibility:** CacheSim exploits the benefits of inheritance and polymorphism by providing virtual function interfaces, which minimize the amount of

code required to extend the functionality of each block.

- **Configurability:** CacheSim allows a high degree of configurability of system parameters such as configuration for number of cores, cache memory hierarchy, replacement policies, cache coherence protocols, bus arbitration policies, and DRAM configuration. These system specification parameters are grouped into a single XML document that allows the simulator to be fully configurable in advance by the user.

- **Reproducibility:** In order to investigate and study the impacts of different parameters' changes, it is crucial to allow reproducibility in simulations. CacheSim implements automated scripts that enable the system designer to regenerate any experiment setup based on a specific test-case configuration file for a specific workload.

- **Integrability:** CacheSim accepts trace-based benchmarks as an input. Besides, it also has an internal fixed DRAM latency model such that it can be run in standalone mode. This mode of operation is devised to improve simulation speed while focusing on evaluating the performance of new architectures with benchmarks by leveraging microarchitecture modeling details to improve simulation speed while maintaining reasonable functional correctness and performance accuracy. On the other hand, CacheSim employs a generalized interface that can be accessed by external memory simulators such as McSim [11] to send memory requests and employs an abstract DRAM interface for the DRAM device model so that the framework is not tied to any specific memory device type. It also provides an easy way to connect to gem5 [12] in order to run full system simulation.

- **Cache memory hierarchy:** CacheSim supports flexible cache organization at both L1 and L2 caches in terms of cache size, cache block size, associativity order, and replacement policy. A detailed discussion of supported features is covered in Chapter 4.

- **Memory Coherence:** Unlike existing state-of-the-art simulators [13, 14, 15, 16], CacheSim supports detailed cache coherence protocols for atomic, pipelined, or split-transaction buses. In CacheSim, we chose to implement coherence protocols that are commonly used in modern commodity multi-core architectures such as MSI, MESI, and MOESI [17, 18]. However, the modular nature of the simulator allows the user to plugin different protocols easily.

- **Interconnect arbitration:** Various arbiters are used to arbitrates coherence and data response messages generated by cache controller instances over the shared interconnect network. CacheSim supports both high-performance arbiters that favor system performance over other metrics such as fairness and predictability (e.g. First-Come-First-Serve (FCFS) [19, 20] and split-transaction [21, 22]), and predictable arbiters such as Time-division multiplexing (TDM) [23] and predictable MSI (PMSI) [1]. Moreover, CacheSim enables Cache-to-Cache data transfer feature between processing cores.

- **Latency Checkpointing:** CacheSim data logger module keeps track of all memory requests initiated by processors and records various performance metrics such as cache hit/miss metrics and latency components of each request at different checkpoints in the system. These metrics are potentially useful to support in order to verify and evaluate the performance new developed algorithms.

We then used CacheSim to explore the design space for improving predictability of accesses to shared data on multi-core systems. In this work, we propose PISCOT: a predictable and coherent bus architecture that (i) provides a considerably tighter latency bound compared to the state-of-the-art predictable coherent solutions (4× tighter bounds in a quad-core system) [1, 24, 25]. (ii) It does so with a negligible performance loss compared to conventional high-performance architecture coherence delays (less than 4% for SPLASH-3 benchmarks). This improves average performance by up to 5× (2.8× on average) compared to its predictable coherence counterpart. Finally, (iii) it achieves that without requiring any modifications to conventional coherence protocols [8].

1.3 Thesis Structure

The remaining of this thesis is organized as follows: Chapter 2 presents a background material for understanding this thesis. Chapter 3 discusses existing cache simulators and compares our solution against them. Chapter 4 describes the architecture design of CacheSim and the detailed implementation of the simulated hardware blocks. Then Chapter 5 proposes PISCOT, a predictable and coherent bus architecture that substantially reduces coherence delays while improving overall system performance. Chapter 6 evaluates and validates CacheSim features. Finally, Chapter 7 concludes the work and provides some guidelines for future research.

Chapter 2

Background

This chapter presents background materials about multi-core system architecture including caches, cache coherence, shared bus architectures, and off-chip DRAM. We start in Section 2.1 by presenting the system model that we consider through the thesis for multi-core systems. Section 2.2 explains what is a cache, how it is organized, and which parameters contribute to the cache's storage requirements. We also discuss details related to cache line replacement algorithms and victim caches. In Section 2.3, we define cache coherence invariants require to maintain data coherency on shared cache architectures and presents the big picture of cache coherence protocols including our table-driven methodology for presenting protocols with both stable and transient coherence states. Section 2.4 discusses shared bus architectures and different scheduling techniques used for arbitrating coherence and data response messages generated by CPU cores and shared memory controller over the shared bus. Finally, in Section 2.5 we give a brief overview of the off-chip DRAM and how to model its latency.

2.1 CMPs System Model

Figure 1.1 shows an example of a single multi-core processor chip and off-chip DRAM. The multi-core processor chips consist of multi single-threaded cores, and each core has its own private L1 cache and cache controller unit. Cache controller implements coherence protocol and acting as an interface between the processor core and shared interconnection network. The chip also includes a last-level cache (LLC) that is shared among all cores. Similar to the cache controller, the LLC controller maintains data coherency at the LLC cache. LLC controller is connected to off-chip DRAM through the DRAM controller and used to initiate memory requests whenever a core requests data that does not exist in the LLC or Victim cache. The Victim cache is used to hold L2 cache evicted cache lines.

2.2 The Cache

On-chip cache memories are used to overcome memory wall [26] problem. Memory system designers employ hierarchies of caches to mitigate the latency and bandwidth gap between processor and off-chip memory. Caches tend to be 10 to 100 times faster than off-chip DRAM [27], and they work perfectly to knock down the memory wall when the running applications exhibit *temporal locality* (i.e. they tend to reuse the same data close in time) or *spatial locality* (i.e. applications tend to use data that are located close to each other in memory). The cache memory is the first checkpoint the cache controller utilizes to decide if the CPU request is a *hit* in the cache or not. If the requested block state is valid and the tag bits stored in the cache match the tag in the original request, then it is a hit. Otherwise, it is a *miss*, and the controller needs to redirect the request to the next cache level or the off-chip DRAM if the miss occurred at the LLC level.

2.2.1 Definitions

A *Cache line* is the smallest amount of data that can be transferred between the upper level memory and the cache, usually represented in multiples of the machine words. Increasing cache line size explore the advantage of the spatial locality. Whenever the data requested by the processor is located in the cache, it is called a *hit*. Otherwise, it is a *miss*. The *mapping* is the techniques used to assign one upper level memory block to one cache line. There are two different policies govern the write operation to the cache. 1) *Write-through*: Whenever, a processor wants to write to a certain cache line, it update the value in both the cache and the main memory. The benefit of write-through to the main memory is to simplify the data coherency management at the expense of increasing the memory request traffic on the shared interconnect network. 2) *Write-back*: the data is written only into the cache in the write-back method. Then, the data is written back to the main memory only when the line is removed from the cache or another core request it. Write-back is the most commonly used technique to manage the write operation to the cache as it avoid useless back writings to the main memory. 3) *Miss penalty*: the time required to load the missing block from the next memory hierarchy level.

2.2.1.1 Different types of misses

A miss occurs when the data requested by the core does not exist in the cache: it must be then fetched from the upper-level cache or main memory. There are four types of misses:

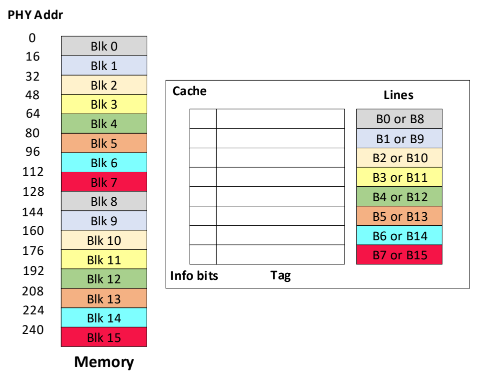
- **Compulsory misses:** It is also known as cold start misses. Theses misses occur when the data is referenced for the first time. Therefore, data must be brought from upper level memory into the cache.
- **Capacity misses:** These misses occur when the application working set exceeds the cache size, so cache need to evicts blocks (i.e. do replacement).
- **Conflict miss:** These misses result from the mapping as we discussed in the section above.
- **Coherence misses:** Theses misses occur in multi-core environment, the cache coherence protocol may invalidate a line as we will discuss in Section 2.3.1.

2.2.2 Cache Organization

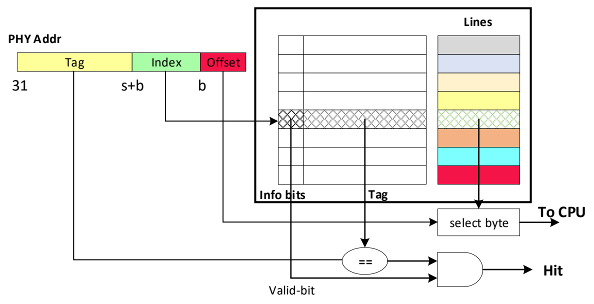
There are two methods used to map the date between upper level caches or main memory and lower level caches as presented below.

2.2.2.1 Direct-mapped cache

Figure 2.1a illustrates the direct-mapped mapping of 128 Bytes cache, where each line is composed of 4 words (i.e. 16 Bytes). The direct-mapped cache is the simplest approach where each main memory address maps to exactly one cache line. The mapping is done by dividing the physical address into three fields as shown in Figure 2.1b: 1) Byte Offset: is the first **b** bits (where $b = log_2(block\ size)$), these bits are used to select a specific word from the read cache line. 2) Cache Line Index: is the next **s** bits (where $s = log_2(number\ of\ lines)$), these bits are used to fetch cache



(a) Mapping



(b) Address decoding

Figure 2.1: Direct-mapped cache

line information stored in the cache. 3) Tag: is the remaining $(32 - s - b)$ bits of the PHY-address.

Direct mapping is a simple strategy but inefficient to deal with conflict miss situation when two memory requests are mapped to the same location in the cache. In this case, direct mapping triggers evictions and may significantly increase the miss rate if this conflict happens frequently. Therefore, it increases the miss penalty. A means to mitigate this issue is to allow a memory line to be mapped onto different cache lines. This solution is presented in the next section.

2.2.2.2 N-way set associative cache

A set associative cache permits data to be mapped into different locations inside the cache based on the associativity order. In set associative mapping, the cache is split into *sets* where each set is composed of N ways. The cache scheme for 2-way set associative cache is drawn on Figure 2.2, where a set is represented as the union of the red rectangles. The CPU memory request is first mapped to a set using direct map method, then it mapped on any of the N ways, thereby giving the possibility to

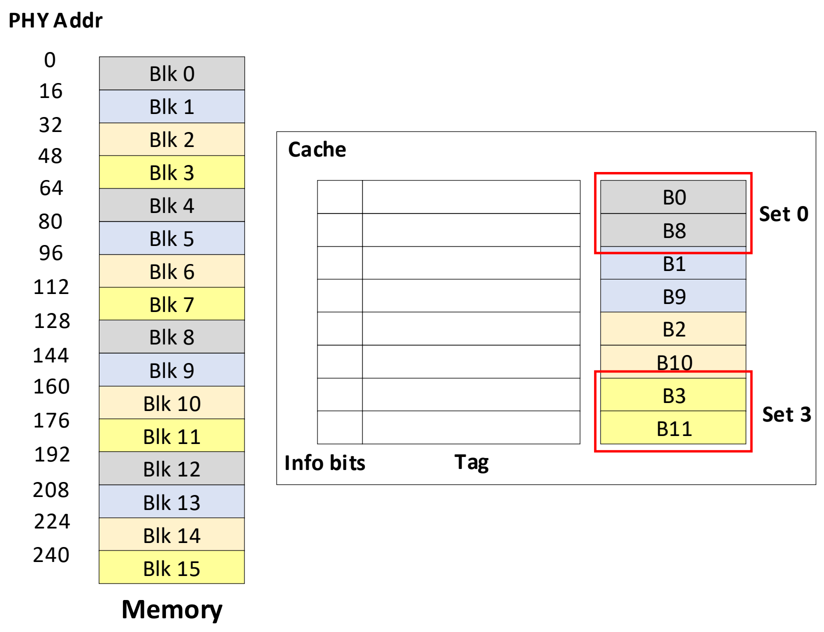


Figure 2.2: Set associative cache

decrease the number of conflict misses. Compared to directed-mapped caches, fully associative caches are expensive to implement because 1) there is no index field in the address anymore, the entire address must be used as the tag, increasing the total cache overhead. 2) Data could be anywhere in the cache, so we must check the tag of every cache block, therefore adding a lot of comparators. Cache associativity order is a trade-off between performance (i.e. low conflict misses) and hardware complexity.

2.2.3 Cache Replacement algorithms

We have seen in the previous section that different upper level memory lines are mapped to the same cache line. Thus, when a set of cache lines is full, an eviction event is triggered among the cache lines of this set, which is the set of an N-way set associative cache. The role of the replacement algorithm is to select the evicted cache line as optimally as possible. Through this section, we briefly explain the replacement policies supported by CacheSim.

- **Random (RAND):** The replacement policy selects a candidate cache line for eviction randomly. This algorithm is easy to implement as linear feedback shift register.

- **Least Recently Used (LRU):** The LRU algorithm evicts the least recently used line. The idea behind is to keep the data recently used which should be used soon (i.e. exploit temporal locality principle). This algorithm requires to keep track of when each cache line was used. It is thus very expensive to implement for large cache with great number of ways.

- **Most Recently Used (MRU):** In contrast to LRU, this algorithm discards the most recently cache line first. This algorithm is useful in situations where the older cache line is more likely to be accessed in the future.

- **First-In-First-Out (FIFO):** Using this algorithm, the cache behaves in the same way as a FIFO queue. It removes block in the order they were brought in the cache, thereby taken advantage of the locality principle in a simple way. FIFO yield a miss ratio $12 - 20\%$ higher than LRU in average [28].

- **Last-In-First-Out (LIFO):** The cache behaves in the same way as a stack when the LIFO is deployed. The cache evicts the block added most recently first without any consideration to how often or how many times it was accessed in the past.

- **Least Frequently Used (LFU):** This algorithm keeps track of the frequency of accesses of the cache lines and replaces the LFU one. The drawback of this method is that lines which have been accessed very frequently and that will not be needed in the future tend to remain in the cache. Usually an aging policy like LRU is preferred to use in this scenario to avoid cache pollution.

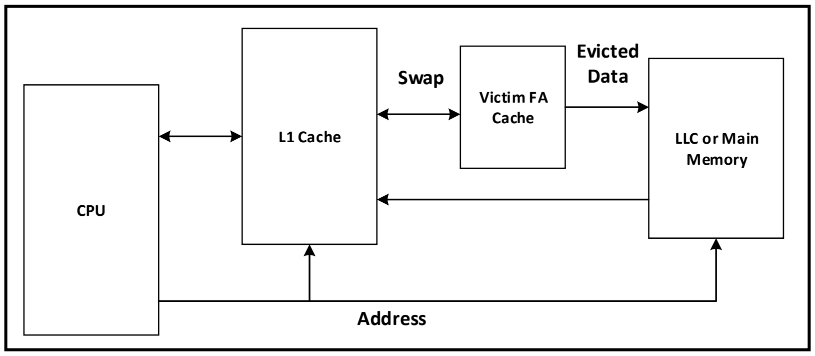


Figure 2.3: A Victim Cache Organization

2.2.4 Victim Cache

CacheSim supports victim cache architecture that was proposed by Norman P.Jouppi [29] to reduce the conflict misses of the direct-mapped cache without affecting its fast access time. Victim Cache is a fully associative cache, whose size is typically 4 to 16 cache lines, locating between the L1 cache and the next level of memory as shown in Figure 2.3. The data evicted from the L1 cache is placed into the victim cache. Victim Cache implements a FIFO replacement strategy. Upon data access, the following chain of events occurs: 1) The L1 cache is checked first; if it is a hit in the L1 cache, the block is fetched from the cache and returned to the CPU. 2) If it is a miss in the L1 cache but a hit in the victim cache, then the block in the victim cache is promoted to the L1 cache, and the L1 cache missed line is placed in the victim cache. 3) If the request is a miss in both the L1 cache and victim cache, then the requested block is fetched from the next memory level and placed into the L1 cache, and the L1 missed cache line is moved to the victim cache. We also can deploy the same concept of victim cache at the higher caches levels to reduce the amount of conflict misses at a low cost since the victim cache is very small compared to the cache size.

2.2.5 Inclusive vs. Exclusive Caches

The data in multi-level caches can be organized in various ways depending on whether the content of one cache level is present in the other cache levels. If the lower-level cache contents are a subset of the higher-level cache, then the lower-level cache is said to inclusive of the higher-level one. On the other hand, multi-level exclusion is the other extreme where the data that is present in the lower-level cache cannot be present in the higher-level cache. Non-inclusion caches lie in between the two. A block can be presented in both the L1 and L2 cache or one of them. CacheSim supports inclusion cache as it is the most commonly implemented policy in CMPs today. For instance, IBM Power4 CMP and Intel Xeons scalable processors. inclusive coherence protocol is simpler to implement than the other two protocols. However, the main drawback is that multi-level inclusion does not effectively exploit the total available cache capacity at the second cache level.

2.3 Cache Coherence

The possibility of incoherence arises only when there are multiple actors with access to caches and memory. In modern systems, these actors are processor cores, DMA engines, and external IO devices. Figure 2.4 shows once instance of data incoherent on the shared cache line **A** in a dual-core system. ① initially, the L2 cache has **A** with a value of 5. ② Core C_0 performs a read on A; hence, it obtains a local copy of **A** in its private L1 cache. ③ Then, C_0 performs a write operation that updates this local copy to 8. ④ Whem C_1 reads **A**, the L2 cache responds with the stale value of **A**, 5. This is because C_0 did not update the L2 cache with the new value of **A**. Therefore, when a cache block resides in multiple caches, and one of the cores modified its private cached version it must ensure that the values in the caches of other cores are updated to prevent outdated values from being used. To accomplish this, cache coherence protocols are used to resolve data coherence issue.

A coherence protocols avoid data incoherence by deploying a set of rules to ensure that cores access the correct version of data at all times. These rules are known as *coherence invariants* [30] and are defined as follow:

- *Single-Writer, Multiple-Read (SWMR) Invariant.* For any memory location **x**, at any given time **t**, there exists only a single actor that may write to **x** (and can also read it) or some number of cores that may only read **x**.
- *Data-Value Invariant.* The value of the memory location at the start of an epoch is the same as the value of the memory location at the end of its last read–write epoch.

The cache controller is the hardware component that implements the coherence protocol. General-purpose processors deploy different variants of coherence protocols, as we will discuss in Section 2.3.1. The vast majority of these protocols called "Invalidate protocols", which are designed explicitly to maintain coherence invariants.

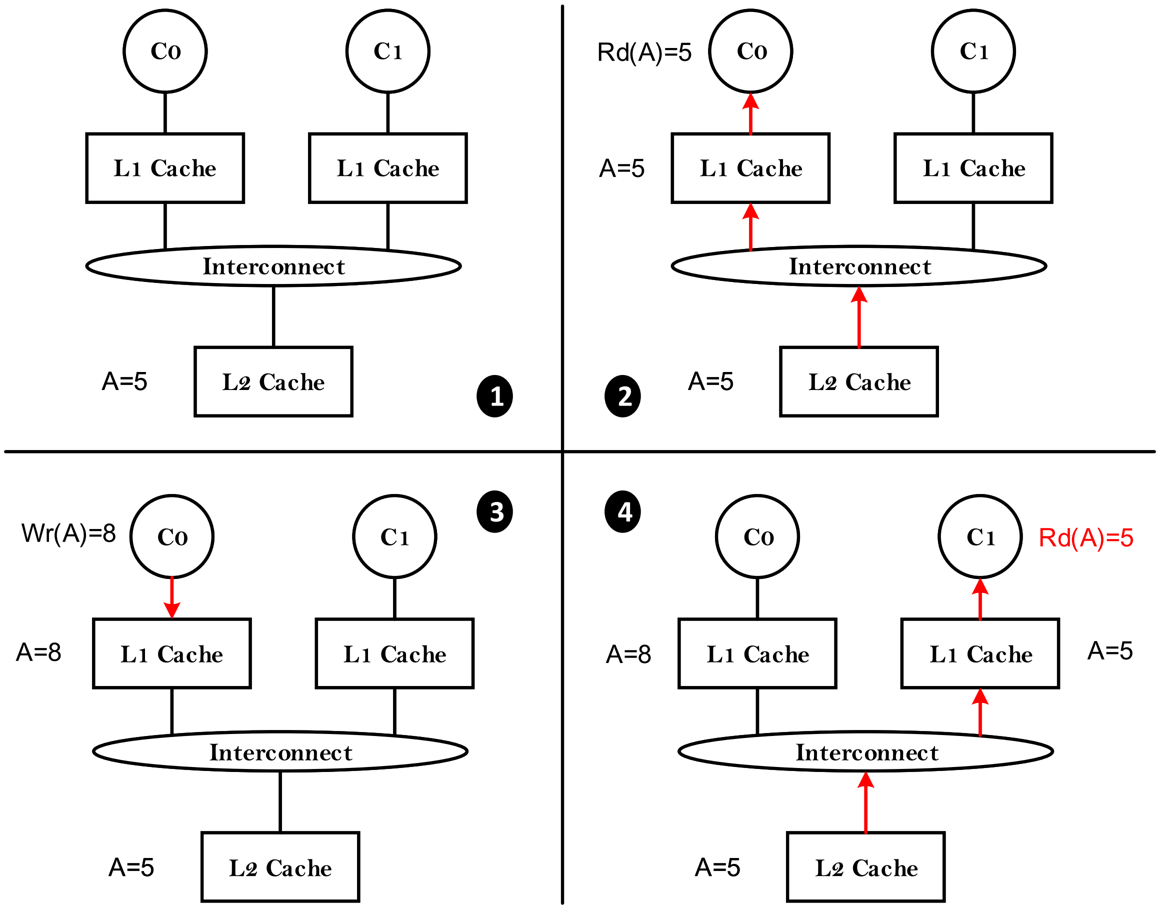


Figure 2.4: Multi-core cache coherence issue

For instance, if a core wants to read a memory location, it sends coherence messages to the other cores to obtain the current value of the memory location and to ensure that no other cores have cached copies of the same memory address in a read-write state. These coherence messages end any active read-write epoch and begin a read-only epoch. Moreover, if a core wants to write to a memory location, it sends another coherence messages to the other cores to obtain the current value of this memory address and to ensure that no other cores have cached version in read-only or read-write states. These coherence messages end any active read-write or read-only epoch and begin a new read-write epoch. Epoch's different permissions are encoded in the coherence state of the deployed protocol and stored in the cache tag for every block.

2.3.1 Cache Coherence Protocols

The cache coherence protocol enforces the coherence invariants through state machines at each cache controller and by exchanging coherence messages between controllers. Many coherence protocols use a subset of the classic five stable states $\{M, O, E, S, I\}$ introduced by Sweazey and Smith [31]. The most fundamental three states $\{M, S, I\}$ represent the minimum set that enables multiple cores to simultaneously hold a cache line in read-only (state S**hared**), or to denote that a single core holds a cache line in read-write (state M**odified**) and the other cores hold it in an invalid state (state **Invalid**). State O**wner** and E**xclusive** are used to implement coherence protocol optimizations. For example, state O achieves two benefits: 1) It allows the protocol satisfies a read request by accessing the cache of another processor core (the owner core) instead of accessing slower DRAM. 2) It eliminates the potentially unnecessary write-back to the next level cache or DRAM. Adding the exclusive E state enables the cache controller to upgrade the unshared cache line from shared S to modified M state saliently without the need for issuing coherence messages on the bus. The E state can thus eliminate half of the coherence transactions in this common scenario.

2.3.1.1 Transient States

The transition between stable states do not usually happen atomically, they are usually interrupted by other requests from other cores as request to the memory bus from different cores are allowed to interleave (i.e. there can be multiple pending requests at the same time) to increase system performance as we will discuss in Section 2.4. Thus, a cache line usually moves to one or multiple transient state(s) in its journey from one stable state to another. In this thesis, we encode these states using a notation XY^Z, which denotes that a cache line moves from stable state X to stable state Y, and the transition will not complete until an event of type Z occurs. The waiting event could be either a data or coherence message or both.

2.3.1.2 Coherence Transactions

A set of messages initiated by cache controller when Load/Store miss occurs in the private cache. Most protocols have a similar set of transactions. Table 2.1 lists a set of typical coherence transactions and the purpose of each transaction. We use the preface "Own" and "Other" to distinguish coherence transactions issued by a given cache controller versus those issued by other cache controllers. For example, OwnGetS() means that the cache controller receives a coherence transaction that was originally initiated by itself.

Table 2.1: Common Cache Controller Coherence Transactions

Transaction	**Goal of Requestor**
$GetS()$	obtain a cache line in Shared (read-only) state
$GetM()$	obtain a cache line in Modified (read-write) state
$Upg()$	upgrade a cache line from read-only to read-write state
$PutM()$	evict a cache line in Modified state

2.3.1.3 Cache-To-Cache Data Transfer

Cache-to-Cache (C2C) intervention optimization allows the cache controller to receive data from another cache controller if the later one owns it in either M, O, or state rather than receiving it from the higher-level memory. This intervention is faster than retrieving the cache line from higher-level cache. Moreover, cache-to-cache communication architectures reduce the congestion on the shared interconnect network.

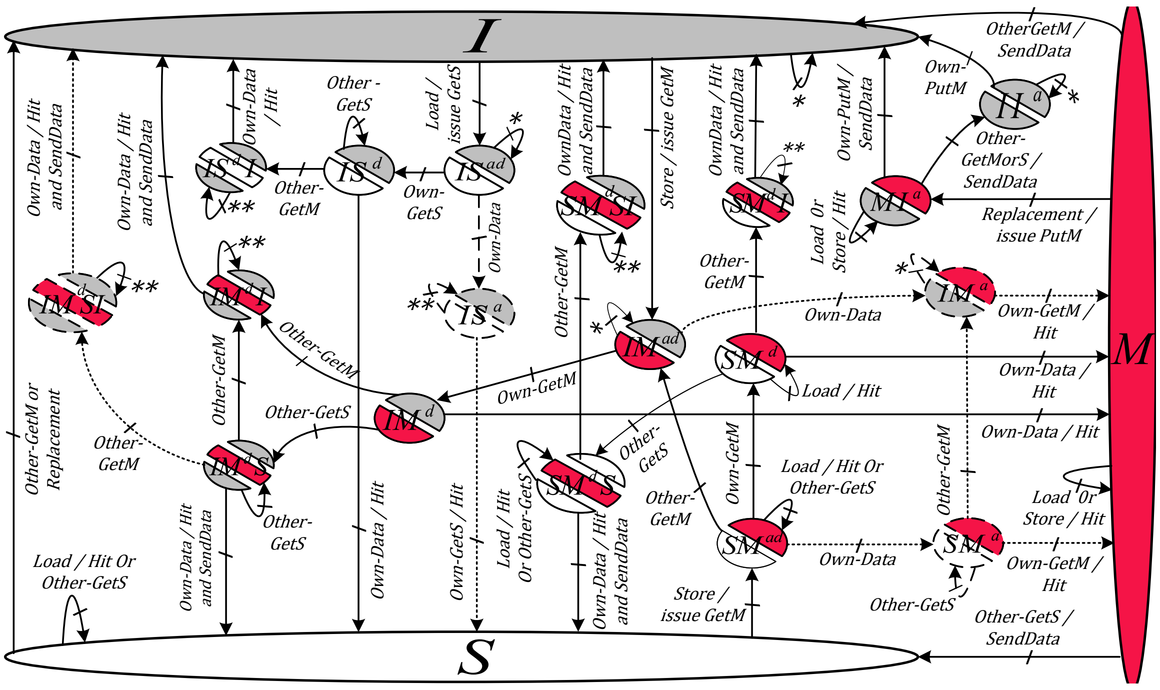
2.3.1.4 Snooping vs. Directory

There are two main classes of coherence protocols: snooping and directory. In snooping-bases protocols, memory requests initiated by the cache controller are broadcasted via a shared bus to all other cores on the bus. Snooping protocols rely on the interconnection network to deliver the broadcast messages consistently to all cores. On the other hand, in directory-based protocols, memory requests are transmitted over an arbitrary point-to-point network to a centralized directory node. The choice between snooping and directory protocols is a trade-off between scalability and complexity. In this thesis, we consider the implementation of the snooping-based protocols only and leave the directory based one for future work.

2.3.1.5 MSI Snooping-Based Protocol

Figure 2.5 delineates the complete state machine of the Modified-Shared-Invalid (MSI) protocol. The diagram includes all transient states and control actions that need to be taken by the cache controller. As with all MSI protocols, cores may perform loads operation (i.e. hit) in states S and M, while stores hit only in state M. On load and store misses, the cache controller issues coherence transactions by sending $GetS()$ and

$GetM()$ requests, respectively. The transient states IS^d, IM^d, SM^d, IS^dI, IM^dS, and IM^dSI indicate that the request message has been sent, but the data response has not yet been received. On the other hand, states IS^{ad}, IM^{ad}, SM^{ad} indicate that the cache controller is waiting for both to observe its coherence message and data response. Moreover, if the controller receives the requested data before observing its coherence message, it has to move to either IS^a, IM^a, or SM^a state and wait for its message to appear on the bus. This is necessary since these broadcasted messages are the contract between all cores guaranteeing that they all observe changes to cache lines in the same order; otherwise, data inconsistencies will exist among cores.



* = Other-GetM, Other-GetS, or Other-PutM. ** = Other-GetM or Other-GetS.

Figure 2.5: MSI Coherence Protocol FSM

We could specify coherence protocol specifications in tabular format, which is the approach that we will use to describe the state machines of other complicated coherence protocols in Chapter 4 instead of using state diagram. Tables 2.2 and 2.3 illustrate the specification of the cache controller at the private and shared memory sides, respectively. We express state machine transitions according to input events in the format "action/next state", and we may omit the "next state" if the next state is the same as the current state. If the state transition requires the controller to perform a specific action, we use "issue" or "SendData" to specify the controller actions. We also denote impossible or invalid transitions using "X", and it is important to implement such error-checking assertions in the state machines to detect these fault transitions when they occur. For instance, a cache controller should never receive a data message for a cache line that it has not requested (i.e., a block in I state). **"-"** indicates no action needs to be taken by the controller.

Table 2.2: MSI snooping protocol state table at cache controller side

State	**Core Event**			**Bus Event**						
	Load	**Store**	**Replace**	**OwnGetS**	**OwnGetM**	**OwnPutM**	**OtherGetS**	**OtherGetM**	**OtherPutM**	**Own data response**
I	*issue GetS/* IS^{ad}	*issue GetM/* IM^{ad}	*X*	*X*	*X*	*X*	-	-	-	*X*
IS^{ad}	*stall*	*stall*	*stall*	-/ IS^{d}	*X*	*X*	-	-	-	-/IS^{a}
IS^{d}	*stall*	*stall*	*stall*	*X*	*X*	*X*	-	-/$IS^{d}I$	-	*Hit/S*
IS^{a}	*stall*	*stall*	*stall*	*Hit/S*	*X*	*X*	-	-	*X*	*X*
$IS^{d}I$	*stall*	*stall*	*stall*	*X*	*X*	*X*	-	-	-	*Hit/I*
IM^{ad}	*stall*	*stall*	*stall*	*X*	-/ IM^{d}	*X*	-	-	-	-/IM^{a}
IM^{d}	*stall*	*stall*	*stall*	*X*	*X*	*X*	-/ $IM^{d}S$	-/ $IM^{d}I$	-	*Hit/M*
IM^{a}	*stall*	*stall*	*stall*	*X*	*Hit/ M*	*X*	-	-	-	*X*

$\boldsymbol{IM^dI}$	*stall*	*stall*	*stall*	*X*	*X*	*X*	-	-	-	*Hit SendData /I*
$\boldsymbol{IM^dS}$	*stall*	*stall*	*stall*	*X*	*X*	*X*	-	-/ IM^dSI	-	*Hit SendData /S*
$\boldsymbol{IM^dSI}$	*stall*	*stall*	*stall*	*X*	*X*	*X*	-	-	-	*Hit SendData /I*
$\boldsymbol{S}$	*Hit*	*issue GetM/* SM^{ad}	-/*I*	*X*	*X*	*X*	-	-/*I*	-	*X*
$\boldsymbol{SM^{ad}}$	*Hit*	*stall*	*stall*	*X*	-/ SM^d	*X*	–	-/ IM^{ad}	-	-/ SM^a
$\boldsymbol{SM^d}$	*Hit*	*stall*	*stall*	*X*	*X*	*X*	-/ SM^dS	-/ SM^dI	-	*Hit/ M*
$\boldsymbol{SM^a}$	*Hit*	*stall*	*stall*	*X*	*Hit/ M*	*X*	-	-/ IM^a	*X*	*X*
$\boldsymbol{SM^dI}$	*Hit*	*stall*	*stall*	*X*	*X*	*X*	-	-	-	*Hit SendData /I*
$\boldsymbol{SM^dS}$	*Hit*	*stall*	*stall*	*X*	*X*	*X*	-	-/ SM^dSI	-	*Hit SendData /S*
$\boldsymbol{SM^dSI}$	*Hit*	*stall*	*stall*	*X*	*X*	*X*	-	-	-	*Hit SendData /I*
$\boldsymbol{M}$	*Hit*	*Hit*	*issue PutM/* MI^a	*X*	*X*	*X*	*Send Data/ S*	*Send Data / I*	-	*X*
$\boldsymbol{MI^a}$	*Hit*	*Hit*	*stall*	*X*	*X*	*Send Data/ I*	*Send Data/* II^a	*Send Data /* II^a	-	*X*
$\boldsymbol{II^a}$	*stall*	*stall*	*stall*	*X*	*X*	-/*I*	-	-	-	*X*

Table 2.3: MSI snooping protocol state table at LLC controller side

State	**Bus Event**				
	GetS	***GetM***	***OwnerPutM***	***OtherPutM***	***Data***
IorS	*SendData*	*SendData*, *SetOwner*/ *M*	*X*	-	*X*
M	*ClearOwner*/ M^dIorS	*SetOwner* / *C2C* ? - : M^dM	*ClearOwner*/ M^dIorS	-	*StoreData*/ $IorS^a$
$\boldsymbol{M^dIorS}$	*stall*	*stall*	*stall*	-	*StoreData*/ *IorS*
$\boldsymbol{M^dM}$	*stall*	*stall*	*stall*	-	*StoreData*/ *M*
$\boldsymbol{IorS^a}$	*ClearOwner*/ *IorS*	*SetOwner*/ *M*	*ClearOwner*/ *IorS*	-	*X*

2.4 Interconnection Network

The complexity of the coherence protocols heavily relies on the interconnect network architecture. For instance, the proposed baseline MSI coherence protocol introduced in Figure 2.5 assumes interconnecting cores with a split-transaction bus. Indeed, a significant optimization can be done to simplify the design of the protocol when atomic buses are deployed at the interconnect network. However, atomic buses degrade performance considerably as shown in Figure 2.6 - ①. Interconnecting cores with an atomic bus prevents all other cores from utilizing the bus until the core that granted access to the request bus receives its data on the response bus. Consequently, most modern systems implement non-atomic buses for improved performance. Figure 2.6 - ② illustrates the operation of a pipelined, non-atomic bus. The key advantage is not having to wait for the data response before a subsequent request can be serialized on the request bus, and thus the bus can achieve much higher bandwidth compared to the atomic bus. Figure 2.6 - ③ shows the operation of the split-transaction bus. The main difference between the pipelined and split-transaction buses is that pipelined bus provides data responses in the same order as the requests, while the split-transaction bus can provide responses in an order different from the request bus.

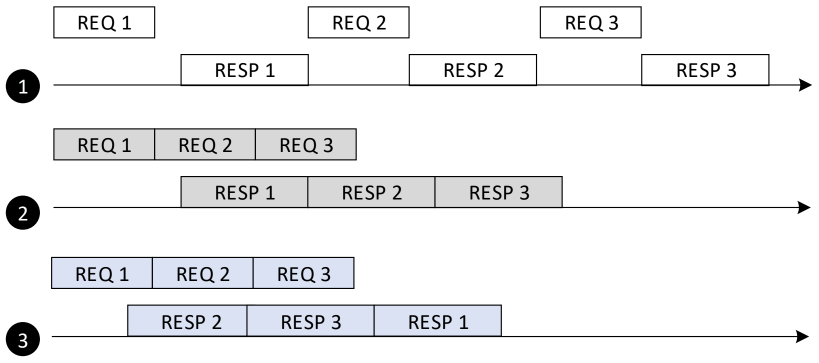


Figure 2.6: Interconnect bus architectures

Compared to a pipelined bus, the advantage of the split-transaction bus is that a low latency response does not have to wait for a long latency response to a prior request. For instance, if REQ 1 is for a cache line that is a miss in LLC cache, then the block has to be fetched from the off-chip DRAM, at the same time REQ 2 is for a cache line owned by LLC, then forcing RESP 2 to wait for RESP 1, as a pipeline bus would require, incurs a performance penalty.

2.4.1 Scheduling schemes

Bus arbiters decide which processing core would become the current master of the bus. A variety of scheduling schemes has been proposed by researchers for both performance-oriented and real-time platforms. Table 2.4 classifies these arbiters into three main categories, and we discuss them in details in the following subsections.

Table 2.4: Bus Arbitration existing approaches.

Approaches	Arbiter	Shared Data Support	Coherence Protocol	Predictability	Examples
COTS platforms baseline	High performance	✓✓	✓✓	✗	FCFS [19, 20], split-transaction [21, 22, 32, 33], priority-based [20, 34]
Traditional Real-Time Arbitration	Predictable by-design	✗(not data-aware)	✗	✓✓	TDM: [35, 36, 23], RR: [37], Harmonic RR (HRR): [38], weighted RR: [39]
Data-Aware Arbitration	builds on traditional arbitration	✓✓	✓(requires coherence modifications)	✓(with significant latency bounds)	PMSI [1], CARP [25], HourGlass [40], PENDULUM [24]

2.4.2 Commodity Performance-Oriented Arbitration

Arbitration among different requests in COTS platforms is usually realized using a high-performance arbiter that favors system performance over other metrics such as fairness and predictability. Such arbiter prioritizes requests based on their arrival time (age-based priority), where older requests are serviced before younger ones. A common example of such arbiter is the First-Come First-Serve (FCFS) scheme [19, 20]. Such arbitration is not predictable since it provides no latency guarantees upon accessing the shared memory. This is because one core can have a request that is pending (theoretically) forever, while other cores are saturating the queues. In addition to age-based arbitration, some COTS platforms also deploy another level of fixed-priority arbitration to give higher-priority for requests from a certain processor. This also entails no guarantees are granted to lower-priority requests. A final observation about COTS arbiters is that for cache coherent systems, the bus is usually implemented as a split-transaction interconnect to increase system performance by concurrently handling both coherent requests (messages) and data responses [30, 21, 22, 32, 33]. For instance, the ARM Corelink CCI550 dictates separate channels for snooping requests and their corresponding responses [32]. Similarly, the Intel's QPI designates different virtual channels to data and coherence messages [33].

2.4.3 Traditional Real-Time Arbitration

In multi-core real-time systems, access to the shared memory (e.g. the Last-Level Cache (LLC)) is managed through a predictable arbiter such as (TDM) [35, 36, 23], and Round Robin (RR) [37]. Considering the TDM arbitration example depicted in Figure 2.7, a request suffers a maximum latency of one TDM period before it is granted access to the bus. For a system with N cores, this is $N \cdot S$ cycles, where S is the slot width in cycles. This occurs when the requesting core just misses its own slot. We denote a core as Cx, where x is the core index. The GetM(B) from $C2$ in Figure 2.7 is an example of such a request, where it arrives to the private cache controller at timestamp t. Assuming that $C2$ just missed its own slot, it waits until $t + 150$ to gain access to the bus. Since the system in Figure 2.7 has three cores, this is equivalent to a one TDM period of 3 slots assuming that the slot width allows for only one memory transfer (one request) and is 50 cycles. Once granted access to the bus, the request conducts its memory transfer consuming an extra slot (50 cycles) and finishes at $t + 200$.

The big limitation of this analysis is that it only applies if cores do not share data. In the example in Figure 2.7, all the cores request to access different cache lines. Consequently, the shared memory is able to respond with the correct data in the request's same slot. Unfortunately, this does not apply if cores are allowed to share data. It has been shown by [1] that shared data can lead to unpredictable behavior even when deploying a predictable arbitration such as TDM.

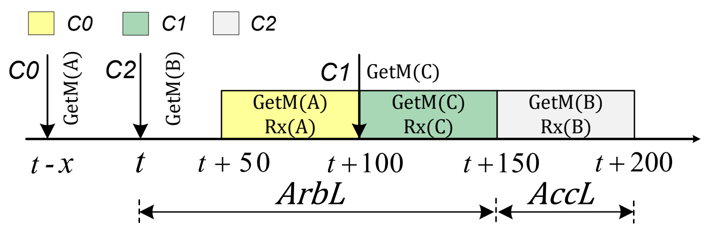


Figure 2.7: Traditional TDM arbitration with no shared data.

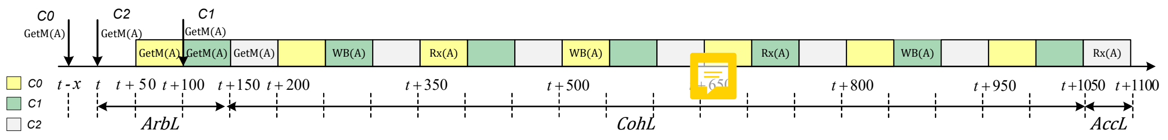


Figure 2.8: TDM-based coherence approach [1]. Initially, C1 owns A in the M state.

2.4.4 Coherent Shared-Data Aware Predictable Arbitration

To guarantee predictability while allowing coherent sharing of data, several recent arbitration solutions have been proposed [1, 24, 25, 9]. All these solutions assume a variant of the TDM arbitration scheme and propose coherence protocol as well as architectural changes to support predictability. Despite showing that coherence can lead to significant performance improvements in data-sharing real-time systems, they incur significant WCL bounds. To illustrate this drawback, Figure 2.8 delineates the TDM behavior for the same system in Figure 2.7 but with assuming that cores can share data, and hence, they issue requests to the same cache line, A. The example follows the protocol guidelines from PMSI [1].

It is clear from Figure 2.8 the significant added latency due to the coherence interference on the shared data. The request under analysis (GetM(A) from $C2$) in this case has to wait for every other core to receive the data of cache line A, conduct the store operation, and then write it back to the shared memory. Since the slot width of the TDM allows for only one memory transfer, and every core gets one slot per TDM period, every core now requires two TDM periods to conduct the aforementioned operation. As a result, $C2$'s GetM(A) request waits until timestamp $t + 1050$ in Figure 2.8 before it can start receiving its requested data. Formally, for a system with N cores and a TDM arbitration with shared data, a request has to wait for up to $(2 \cdot N^2 + 2 \cdot N) \cdot S$ before it can start transferring its requested data [1]. The other existing solutions while supporting systems with mixed criticalities [24, 25], this comes at the expense of incurring even larger WCL than PMSI if all cores have the same criticality. The DISCO solution in [9] improves the WCL bounds by requiring a special handling of writes compared to reads.

It is worth noting that in Figure 2.8 it might seem that there are many idle slots, and thus, this large latency can be completely avoided using a work-conserving schedule. However, this is not true since there can be requests from other cores in the system that utilize these slots. They are not shown in Figure 2.8 for simplicity. For example, $C0$ receives its requested data at timestamp $t + 400$. Thus, it can issue another request afterwards in its coming slots. Clearly, in an out-of-order architecture, more pending memory requests can also co-exist in the system.

2.5 Off-chip DRAM

Dynamic Random Access Memory (DRAM) is a three-dimensional array of memory cells consisting of multiple banks, and each bank consists of several rows and columns. A DRAM module consists of one or more ranks such that each rank consists of multiple banks. Multi-rank DRAM is used to form a wide interface to increase the amount of data that can be transferred in one access; this is known as the memory granularity. DRAM accesses are controlled by the memory controller (MC) that arbitrates among different requestors. MC also generates memory access commands and translates the request's physical address into channel, rank, bank, row, and column addresses. There are four main types of commands generated by MC: RAS, CAS, PRE, and REF. The RAS commands use the row address to index a particular row in a bank and place the data into the row buffer. The row buffer is a temporary buffer that holds the open row data for further reads and writes operations. The CAS command reads or writes a specific portion of data in the row buffer. The pre-charge command PRE is used to write back the row buffer into the memory cells. DRAM cells must be refreshed periodically in order to retain the stored values. MC uses REF commands to reference the DRAM. The performance of the main memory subsystem is highly dependant on the arbitration scheme deployed at the MC and on the DRAM device technology [11]. Several existing DRAM device models can be used to model the complex internal behaviour of the DRAM device depending on the required accuracy and simulation speed. For example, a simple fixed-latency [41] or non-cycle accurate models [42] can be used for fast DRAM simulation, while more sophisticated device simulators such are Ramulator [43] can be used when the model accuracy is the primary concern.

Chapter 3

Related Work

We used a survey of cache simulators published by Brais et. al [44] in 2020 to compare CacheSim against a subset of these simulators. The survey provides a detailed discussion on 28 CPU cache simulators, including popular or recent simulators. We compared our proposed solution against a subset of 13 simulators given in Table 3.1, the other remaining simulators either have similar characteristics or were commercialized. In the following sections, We compare these simulators in three major design characteristics: 1) Support for different cache configurations such as cache size, cache block size, associativity, and replacement algorithms. 2) CMPs architecture support in terms of the configurable number of processing cores and cache hierarchy levels, shared memory hierarchy, coherence protocols, and cache inclusiveness. 3) Interconnect and bus arbitration.

3.1 Cache Configurability

According to Table 3.1, all simulators, including our solution, support configurable cache parameters such as cache size, cache block size, and associativity. Also, many simulators such as gem5, Cachegrind [13], and Dinero IV [14] allow different configurations for the replacement policy. Other simulators, such as MacSim [16], use a fixed replacement policy (LRU, in this case).

Table 3.1: Cache simulators features comparison

Simulator	**Design feature**								
	PS	CO	N_{core}	Cohr	Replc	BW	Arb	IE	N_{lvl}
gem5 [12]	✓	✓	✓	✓	✓	✓	✓	✓	✓
Cachegrind [13]	✓	✓	**F**	✗	✓	✗	✗	**F**	**F**
Dinero IV [14]	✓	✓	**F**	✗	✓	✗	✗	**F**	✓
CASPER [15]	✓	✓	✓	✗	✓	✗	✗	**F**	✓
CMP$im [45]	✓	✓	✓	**?**	✓	**?**	**?**	✓	✓
drcachesim [46]	✓	✓	✓	✗	✓	✗	✗	**F**	**F**
MARSSx86 [47]	✓	✓	✓	✓	**F**	✓	✓	**F**	✓
McSimA+ [48]	✓	✓	✓	**F**	**F**	✓	**F**	✓	✓
MacSim [16]	✓	✓	✓	✗	**F**	✓	✓	✓	✓
vCSIMx86 [49]	✓	✓	✓	**F**	✓	✗	✗	✓	✓
SMPCache [50]	✓	✓	✓	✓	✓	✓	✓	**F**	✓
Multi2Sim [51]	✓	✓	✓	**F**	✓	✓	✓	**F**	✓
ZSim [52]	✓	✓	✓	**F**	✓	**F**	**F**	**F**	✓
CacheSim	✓	✓	✓	✓	✓	✓	✓	**F**	**F**
Notes	1) **PS** = private and shared caches, **CO** = cache organization including cache size, cache block size, and associativity, $\boldsymbol{N_{core}}$ = number of processing cores, **Cohr** = cache coherence, **RP** = replacement policy, **BW** = bus bandwidth, **Arb** = bus arbitration, **IE** = inclusive v/s exclusive caches. $\boldsymbol{N_{lvl}}$ = number of cache levels 2) ✓= configurable to some extend, **F** = fixed and simple, ✗= not simulated or considered, **?** = unknown.								

3.2 CMPs architectures support

3.2.1 Multi-threaded workloads

An essential aspect of a simulator is to support multi-threaded workloads as many of today's modern embedded system applications rely on TLP to improve performance. Dinero IV and Cachegrind can run only single-threaded workloads. Usually, two approaches can be used to support multi-thread execution. 1) Full system simulators (FS), such as gem5 [12] and MARSSx86 [47], rely on the underlying OS to schedule threads and map them into different processing cores. 2) Trace-based simulators including CASPER [15] and vCSIMx86 [49] use benchmarks for multi-thread applications as an input to support TLP simulations. A full system simulation approach is beneficial for design space exploration that requires heavy I/O activities or extensive OS kernel function support. However, these simulators are relatively slower than trace-driven simulators [48]. Moreover, involving OS and its underlying running software stack make it challenging to isolate the impact of architecture changes from the interaction between hardware and software. Therefore, FS simulators are not the best choice compared to the trace-based class if the research targets only architectural aspects. Thus, we consider the trace-based approach as the primary mode of operation in our simulator. The primary motivation behind developing CacheSim is to use it for architectural change explorations, study new design ideas, and evaluate existing ones. We have no intention to position our simulator as a FS simulator. Nevertheless, CacheSim is designed in an extensible and integrable way through generic interfaces such that it can be integrated easily with any full-system simulator to support FS simulations if needed.

3.2.2 Multi-level cache hierarchy

Current CMPs employ a complex cache hierarchy to mitigates latency and bandwidth gap between processors and memory speed. Consequently, it is vital to support multi-level cache hierarchy with both private and shared caches. Some simulators such as CMP$im [45], McSimA+ [48], and zSim [52] support a configurable cache hierarchy, while others (e.g. Cachegrind [13] and drcachesim [46]) support a fixed hierarchy of private L1 cache and a shared L2 cache among all cores. This architecture is the most common hierarchy deployed in CMPs design more than a decade ago and still common for low-power processors. CacheSim supports the common hierarchy (fixed hierarchy, in this case).

3.2.3 Cache coherence protocols and cache inclusiveness

According to Table 3.1, many of the existing simulators offer limited coherence support. For instance, Cachegrind, Dinero IV, CASPER, drcachesim, and MACSim do not support cache coherence into their design, and thus the correct execution and timing reports for multi-threaded applications are not guaranteed due to the lack of cache coherence protocols. McSimA+ does not support snooping-based coherence protocols. It focuses on GPUs and NoC architectures; therefore, it supports only directory-based coherency. SMPCache supports cache coherence but is not available as open-source. CMP$im is not an open-source, and according to the survey [44] its coherence support is unknown. The two remaining FS simulators that support coherency are gem5 and MARSSx86. gem5 version 20.0+ [53] includes two cache systems models: the classic caches and Ruby caches. The classic caches have single hard-coded hierarchical MOESI coherence protocols. On the other hand, Rudy caches enable configurable cache system model. However, it requires the user to define coherence protocols using a domain-specific language called specification language for implementing cache coherence (SLICC). Since it requires the user to define coherence specifications and learn a new coding language, it is not a pluggable solution. Also, gem5 is not the best simulator to use for architecture improvements, as we discussed in subsection 3.2.1. On the other hand, MARSSx86 supports both MESI and MOESI protocols. However, it only supports round-robin arbitration at the interconnect network. Clearly, there is a potential need to have a light-weight simulation tool with detailed coherency and bus arbitration models to enable researchers exploring new designs for CMPs architectures, and this is where our simulator work steps in.

Finally, Cache's inclusiveness is tied with coherency. Most simulators, including our proposed solution, support the inclusive model. Exceptions like gem5 allows configuring exclusive caches as well.

3.3 Interconnect and bus arbitration

We have covered most of the current work related to interconnect and bus arbiters in the background chapter in Section 2.4.1. Hence, we focus here on the simulators' flexibility of bus arbitration and interconnect model. Similar to the cache coherence observation that we made in the previous section, many of the existing simulators do not support flexible arbitration policies at the interconnect network. For example, cache-only simulators like drcachesim do not contain a model for the interconnect. Zsim and McSimA+ support only a round-robin arbitration policy. Exception again for FS simulators like gem5 do support detailed specifications for the interconnect.

Chapter 4

CacheSim Framework

CacheSim is a cycle-based detailed cache-coherent interconnect simulator designed to supports holistic coherency models for modern interconnecting multi-core bus architectures. Decomposing a system into modules is a commonly accepted approach to developing software. CacheSim advocates a decomposition based on the principle of information hiding [54]. This principle supports design for change, because the ”secrets” that each module hides represent likely future changes, especially during initial development as the solution space is explored. Therefore, design for change is valuable to consider when developing software. CacheSim is written in C++ using object-oriented programming concepts to support a modular, expansible, configurable, and integrable implementation of the simulator. Throughout this chapter, we introduce the high-level architecture of the CacheSim in Section 4.1. Then, we discuss the detailed implementation of simulated hardware components in Section 4.2 including the uses hierarchy between modules. We defer the evaluation and validation discussion of CacheSim to Chapter 6.

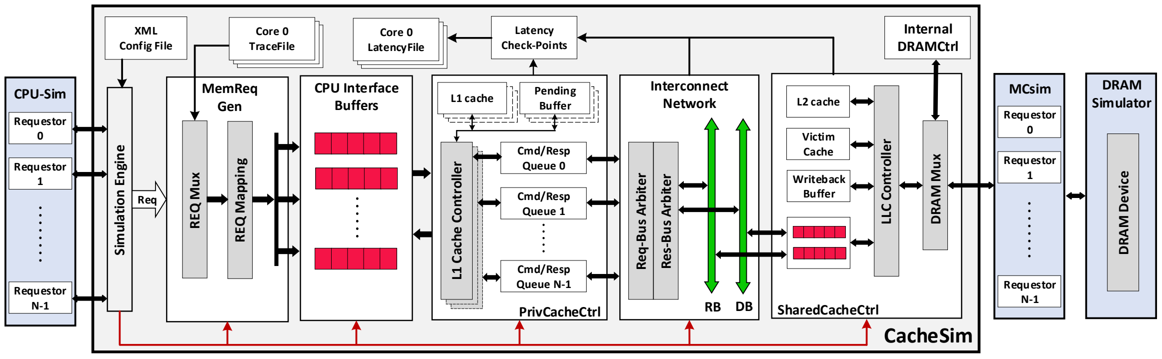


Figure 4.1: CacheSim high-level architecture and major blocks

4.1 CacheSim High-Level Architecture

Figure 4.1 depicts the high-level architecture of CacheSim. CacheSim is decomposed into seven subsystems. 1) ***CacheSim*** is the main constituent of the simulator which responsible for creating instances of the various classes based on the user configuration. The simulator accepts two types of inputs from the user, i) high-level configuration parameters such as benchmarks trace files path, output results path, and other debug configurations are given using command-line arguments. ii) test-case specific parameters such as configurations for each subsystem component are provided in a format of XML document. We advocate this technique to allow a high degree of configurability of the running experiments. Besides, to allow reproducibility in simulation, which is a key feature for architectural change explorations and study new design ideas. Moreover, CacheSim provides a generalized I/O interface to facilitate integrability with external simulators such as CPU model simulator (e.g. gem5 [12]) and/or DRAM memory system simulator (e.g. MCsim [11] and Ramulator [43]) to support full system simulations. 2) ***CpuCoreGen*** supports trace-based simulation mode. This mode allows running multi-thread simulation through benchmarks trace files given as an input to the simulator. *CpuCoreGen* sends memory requests and receives data acknowledgments from its corresponding cache controller subsystem through a set of CPU interface buffers. The benefit of including those buffers is to support asynchronous operation between subsystems and also to facilitate modularity and OOO operations. 3) ***PrivCacheCtrl*** is the main constituent of the L1 cache system of each processing core, which includes i) cache controller that implements top-level controller of L1 cache in addition to cache coherence protocol. ii) cache memory device, including implementation for all replacement policies discussed in Section 2.2.3, iii) pending write-back buffer, which is used to save pending write-back messages that need to be sent on the interconnect. iv) a set of bus interface buffers to facilitate asynchronous operation between cache controller and interconnect network. 4) ***Interconnect network*** considers the implementation of the interconnect including both unified and split bus architectures in addition to bus arbitration policies that are discussed in Section 2.4.1 as well as the proposed PISCOT solution that is presented in Chapter 5. *Interconnect network* connects both the L1 cache system of each processing core and shared L2 cache systems through the bus interconnect. Again, the communication between these components is facilitated through a well-defined interface to support modularity and design for change principle. 5) Similar to *PrivCacheCtrl* system, ***SharedCacheCtrl*** implements i) top-level controller of L2 cache and the cache coherence protocol at shared memory side.should be ," and for similar following points as well ii) L2 cache memory device, including replacement algorithms. iii) pending write-back buffer which holds shared cache controller write-back messages. iv) a set of bus interface buffers to facilitate asynchronous operation between modules. *SharedCacheCtrl* has another interface with the *DRAMCtrl* system, and the interface is implemented using a set of FIFOs as well. However, it is not shown in Figure 4.1 for simplicity. It is worth mentioning that both L1 cache memory and L2 cache memory systems are implemented using an abstract class to allow code reusability. We have decided to implement the victim cache at the L2 cache system. However, it can be instantiated inside the L1 cache system easily if required. 6) ***DRAMCtrl*** supports fixed DRAM latency model approach that is discussed in Section 2.5. This module aims to allow for DRAM simulation without the need to interface CacheSim with an external DRAM simulator. However, as discussed, the simulator provides a clear interface that easily allows it to be extended with any DRAM simulator. 7) ***Latency Check-points*** module serves as a data logger, which keeps track of all memory requests generated in the system and records various performance metrics such as cache hit and miss rates and latency components at different points in the system. These metrics will be considered when we validate the simulator features in Chapter 6.

4.2 CacheSim Functional Hardware Blocks

Throughout this section, we discuss the detailed implementation of CacheSim's hardware blocks and their interactions according to the class diagram in Figure 4.2. We use the filled diamond shape to represent the composition relationship between modules, hollow triangle shape to represent inheritance, and solid triangle shape to represent association relationship. For the sake of readability, we use the small boxes as connectors to the original class names.

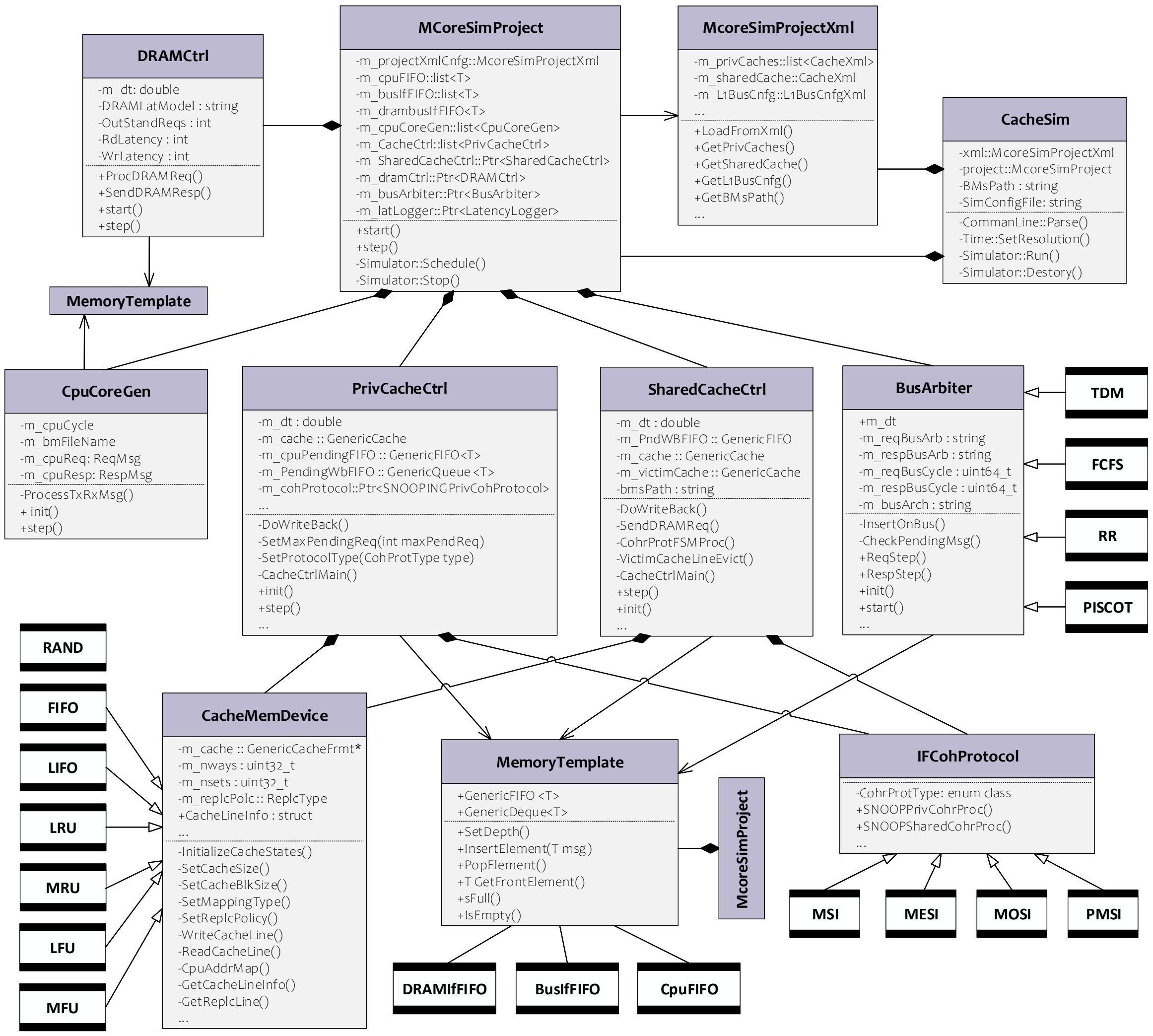


Figure 4.2: CacheSim class diagram representing the main functional blocks in the simulator

4.2.1 CacheSim Top-Level Node

CacheSim is the top-level node of the simulator hierarchy. It creates a configurable infrastructure of the multi-core system environment based on the test-case configuration parameters provided in the input XML document. In Code 4.1, we present `CacheSim` main function. `CacheSim` first resolves user's command-line configurations using `CommandLine` class, next it acquires the test-case specific configuration parameters from the XML document with the aid of `MCoreSimProjectXml` class, then it instantiates the `MCoreSimProject` class to construct the different system components that have been discussed in Section 4.1. `CacheSim` utilizes two external classes from ns-3 [55] library. i) `ns3::Time` which manages the virtual time in real world units. For instance, `CacheSim` uses this class to specify simulation time resolution m_dt. ii) `ns3::Simulator` which controls the scheduling of simulation events spawn by different system components. Finally, `CacheSim` calls `Start()` method to start simulation engines. `Start()` is an external method that is implemented inside `MCoreSimProject` class and illustrated in Code 4.3.

```
NS_LOG_COMPONENT_DEFINE ("CacheSim");
int main (int argc, char *argv[])
{
  // simulator output Latency trace Files
  string LatTracePath        = "LatTracePath";
  // simulator input files
  string SimConfigFile = "test_cfg.xml"; // testcase cnfg file
  string BMsPath = "BMs/tests/"; // BMs Path
  bool LogGenEn = true; // enable log file dump
  // command line parser
  CommandLine cmd;
  // adding a call to sim configurable parameters
  cmd.AddValue("CfgFile", "simulator cfg file", SimConfigFile);
  cmd.AddValue("BMsPath", "BMs file(s) path", BMsPath);
  cmd.AddValue("LogGenEn", "enable flag for log files", LogGenEn);
  // parse user commands
  cmd.Parse (argc, argv);
```

```
    // read Xml Configuration File
    TiXmlDocument doc(SimConfigFile.c_str());
    MCoreSimProjectXml xml;
    xml.LoadFromXml          ( doc              );
    xml.SetBMsPath           ( BMsPath          );
    xml.SetLatTracePath      ( LatTracePath     );
    // setup simulation environment
    MCoreSimProject project  ( xml              );
    // set simulation clock resolution to 1 ns
    Time::SetResolution (Time::NS); // MS, US, PS
    // lunch simulator
    project.Start();
    Simulator::Run();
    // clean up once done
    Simulator::Destroy();
    return 0;
}
```

Code 4.1: CacheSim main function

4.2.2 MCoreSimProject Class

As mentioned before, `MCoreSimProject` class creates a configurable simulation infrastructure based on the input configuration. `MCoreSimProject` instantiates all system components and controls their internal behaviour using a group of `Set()` and `Get()` function calls. Code 4.2 shows an example of constructing *CpuCoreGen*, *CpuIfFIFO*, and *PrivCacheCtrl* subsystems based on input XML configurations. These subsystems are created using `m_cpuCoreGen`, `m_cpuFIFO`, and `m_cpuCacheCtrl` objects, respectively. `MCoreSimProject` access system configuration parameters using `m_projectXmlCfg` object of type `MCoreSimProjectXml` that is passed to it during the instantiation. `MCoreSimProject` class knows how many processing cores and their cache controller configurations by accessing xmlPrivCaches configuration list from `MCoreSimProjectXml` class using `GetPrivCaches()` method. `MCoreSimProject` instantiates the remaining system components such as *SharedCacheCtrl*, *Interconnect network*, *DRAMCtrl*, and *LatencyLogger* using the same concept. Those components are accessible through `m_busArbiter`, `m_dramCtrl`, and `m_latLogger` objects. respectively. Clearly, this design approach augments CacheSim with a generalized template for building simulation hierarchy, leading to improved configurability and extensibility.

```
MCoreSimProject::MCoreSimProject(MCoreSimProjectXml projectXmlCfg) {
  // Set project xml cnfg
  m_projectXmlCfg = projectXmlCfg;
  // Get all cpu configurations from xml
  list<CacheXml> xmlPrivCaches = projectXmlCfg.GetPrivCaches();
  // iterate over each core
  for (list<CacheXml>::iterator it = xmlPrivateCaches.begin();
        it != xmlPrivateCaches.end();it++) {
     CacheXml PrivateCacheXml = *it;
     // instantiate cpu interface FIFO
     Ptr<CpuFIFO> newCpuFIFO = CreateObject<CpuFIFO> ();
     m_cpuFIFO.push_back ( newCpuFIFO );
     // instantiate trace-based core model
     Ptr<CpuCoreGen> newCpu = CreateObject<CpuCoreGen> (newCpuFIFO);
     m_cpuCoreGen.push_back ( newCpu );
     // instantiate cache ctrl Bus IF FIFOs
     Ptr<BusIfFIFO> newBusIfFIFO = CreateObject<BusIfFIFO> ();
```

```
        // instantiate cache ctrl
        double ctrlClkPeriod = PrivateCacheXml.GetCtrlClkNanoSec();
        Ptr<PrivCacheCtrl> newCacheCtrl = CreateObject<PrivCacheCtrl>
                                            (newBusIfFIFO, newCpuFIFO);
        m_cpuCacheCtrl.push_back( newCacheCtrl );
        // pass config params using set methods
        ...
    }
    // instantiate other system components such as
    // SharedMemCtrl, DRAMCtrl, BusArbiter, ..., etc.
```

Code 4.2: CacheSim's configurable simulation infrastructure

Once simulation infrastructure is created and all instances are wired together, then `MCoreSimProject::Start()` method is used to initialize the internal states of the hardware blocks. This is done by invoking `init()` function call of each subsystem as shown in Code 4.3. In addition, `MCoreSimProject::Start()` uses `Schedule()` method from `ns3::Simulator` class to register a callback attached with `step()` function every m_dt second.

```
// start simulation engines
void MCoreSimProject::Start() {
  for(list<Ptr<CpuCoreGen> >::iterator it = m_cpuCoreGens.begin();
      it != m_cpuCoreGens.end(); it++) {
    (*it)->init();  // start cpuGen init function
  }
  for(list<Ptr<PrivCacheCtrl> >::iterator it = m_CacheCtrl.begin();
      it != m_CacheCtrl.end(); it++) {
        (*it)->init(); // start PrivCacheCtrl init function
  }
  for(list<Ptr<LatLogger> >::iterator it = m_latLogger.begin();
      it !=  m_latLogger.end(); it++) {
        (*it)->init(); // start latency logger init function
  }
  m_SharedCacheCtrl->init(); // start SharedCacheCtrl
  m_dramCtrl->init(); // start DRAMCtrl
  m_busArbiter->init(); // start BusArbiter
  // schedule CacheSim step() callback
  Simulator::Schedule(Seconds(0.0), &Step, this);
  Simulator::Stop(MilliSeconds(m_totalTimeInSeconds));
}
```

Code 4.3: Initialize internal states and run simulation

4.2.3 CacheSim test case configuration file

CacheSim accepts test case configurations parameters in XML document format. In code 4.4, we show an example of a test case configuration file for quad-core system running at 1 GHz with out-of-order pipeline (NPendReq = 8), 16 kB direct-mapped L1 per-core private cache, and a 1 MB 8-ways set-associative L2 shared cache across all cores. The XML document is formatted in such a way to support 1) the expansibility feature of the simulator. For instance, the configuration parameters for the processing cores subsystem are grouped together into a list of data structure `privateCaches` that can be expanded or shrunk based on the requirement of the running experiments. A user can utilize the same configuration file to run a dual-core simulation by just modifying this list to accommodate the configuration for two cores instead of four and set *nCores* parameter value to 2. 2) a high degree of configurability for the system parameters. For example, a user can specify different configurations for cache organization, interconnect architecture, or replacement policy by just changing the configuration in the XML document without the need to modify the source code. 3) building an automation framework for unit and continuous regression testing as discussed in Chapter 6.

```xml
<CacheSim
  nCores="4"
  CohProtocol="MOESI"
  ... >
  <!--L1 bus configuration params -->
  <InterConnect>
    <L1BusCnfg
      BusArch="split"
      BusArb="PISCOT"
      ReqBusLat="4"
      ... >
  <privateCaches>
    <privateCache <!-- core 0 and Privcachectrl cnfg params -->
      NPendReq="8"
      ReplcPolc= "RANDOM"
      cacheSize="16384"
      mapping="DirectMap"
      ...>
    </privateCache>
    <!-- Other cores configuration params -->
    ...
  <sharedCaches>
    <sharedCache <!--L2 cache configuration params -->
```

```
      ReplcPolc= "LRU"
      cacheSize="1048576"
      nways="8"
      ... >
    </sharedCache>
  </sharedCaches>
  <DRAMCnfg  <!--DRAMCtrl configuration params -->
       MEMMODLE="FIXEDLat"
       MEMLATENCY="250"
       ...>
  </DRAMCnfg>
</CacheSim>
```

Code 4.4: Test case input configuration file

4.2.4 MCoreSimProjectXml parser

This class converts the user data given in the input XML documents into various data structures using `LoadFromXml()` function. The simulation parameters for L1 cache for each processing core are stored into a list of `CacheXml`. `L1BusCnfgXml` class contains the interconnect and bus arbitration configuration parameters. Similarly, other data structures that are not shown in Code 4.5 is used to store the remaining system configuration parameters. `MCoreSimProjectXml` also provides a set of `Get()` methods to allow `MCoreSimProject` class to acquire these configurations.

```
class MCoreSimProjectXml {
private:
  int  m_totalTimeInSeconds;
  int  m_runTillSimEnd;
  int  m_busClkNanoSec;
  int  m_nCores;
  // more code
public:
  // load input configurations from XML doc
  void LoadFromXml (TiXmlHandle root)}
  // return L1 cache configuration params
  list<CacheXml> GetPrivCaches();
  // return shared cache configuration params
  CacheXml GetSharedCache();
  // return interconnect configuration params
  L1BusCnfgXml GetL1BusCnfg();
  // return BMs files path
  string GetBMsPath ();
  // more methods
  ...
};
```

Code 4.5: XML parsing class

4.2.5 CpuCoreGen Class

`CpuCoreGen` class reads memory requests from input trace files, process the trace information, and issue memory requests at the specified time-stamp using `ProcessTxRxBuf()` method. In case of full system simulation mode, this function processes the memory request from an external input interface instead of reading it from the trace file. However, this feature is not implemented and is left for future work. The memory request is sent to the cache controller subsystem using transmitter `m_txFIFO` queue. `ProcessTxRxBuf()` calls `InsertElement(T msg)` method from `m_cpuFIFO->m_txFIFO` to insert the message. Similarly, `CpuCoreGen` uses `m_cpuFIFO->m_rxFIFO` queue to receive data acknowledgment from the cache controller. Code 4.6 shows a sample of `CpuCoreGen` interface. The `init()` method is accessible by the upper layer (i.e. `MCoreSimProject`) to initialize the module and schedule `Step()` callback every clock cycle. `Step()` is used to invoke `ProcessTxRxBuf` to process CPU memory requests.

```
class CpuCoreGen  {
  private:
    int  m_coreId; // cpu core id
    double m_dt; // cpu clk period
    std::ifstream m_bmTrace; // benchmark file stream
     Ptr<CpuFIFO> m_cpuFIFO; // a pointer to cpu I/F FIFO
    // process memory requests
    void ProcessTxRxBuf();
  public:
    // initialize core generator
    void init();
    // callback to run ProcessTxRxBuf every cycle,
    static void Step(Ptr<CpuCoreGen> cpuCoreGen);
     // other methods
     ...
  };
  // step function
  void CpuCoreGen::Step(Ptr<CpuCoreGen> cpuCoreGen) {
    cpuCoreGen->ProcessTxRxBuf();
  }
```

Code 4.6: `CpuCoreGen` class interface

4.2.6 Private Cache Controller

PrivCacheCtrl class implements private cache controller hardware logic of each CPU core (i.e. *PrivCacheCtrl* in Figure 4.1). Code 4.7 depicts PrivCacheCtrl class interface. We use Figure 4.3 to show queues structure and connection flow inside PrivCacheCtrl class. PrivCacheCtrl has two interfaces with the top-level module (i.e. MCoreSimProject in this case). 1) m_cpuFIFO to communicate with the CpuCoreGen module. 2) m_busIfFIFO to communicate with the interconnect BusArbiter. Besides, it has access to the L1 cache memory device which is implemented using GenericCache class. PrivCacheCtrl instantiates two internal buffers, m_cpuPendingFIFO and m_PendingWbFIFO, to keep track of the outstanding CPU memory requests and pending write-back data responses, respectively. The cache controller's main objectives are to i) serve incoming CPU memory requests by checking it first in its private L1 cache and issue coherence transactions on the bus if the requests are misses in the cache. ii) monitor the received coherence messages on interconnect to maintain the data coherency across its L1 cache. iii) do a write-back of their owned cache lines when other cores request them. PrivCacheCtrl first invokes m_cohProtocol micro-controller module to determine the control actions that need to be taken to achieve these objectives. Then, it uses the top-level controller CacheCtrlMain() methods to execute these actions. m_cohProtocol periodically monitors both the CPU interface FIFOs (i.e. m_cpuFIFO) and interconnect interface FIFOs (i.e. m_busIfFIFO) to decide the list of required actions using cache coherence protocol FSM and other control logic. PrivCacheCtrl is controlled by the top-level through a set of Get() methods. For instance, MCoreSimProject uses SetProtocolType() to configuration the coherence protocol type. MCoreSimProject also invokes `init()` at the beginning of the simulation to initialize the controller internals, including the cache memory and FSM initial states. The `init()` also calls `Step()` to schedule callback events for `CacheCtrlMain()`.

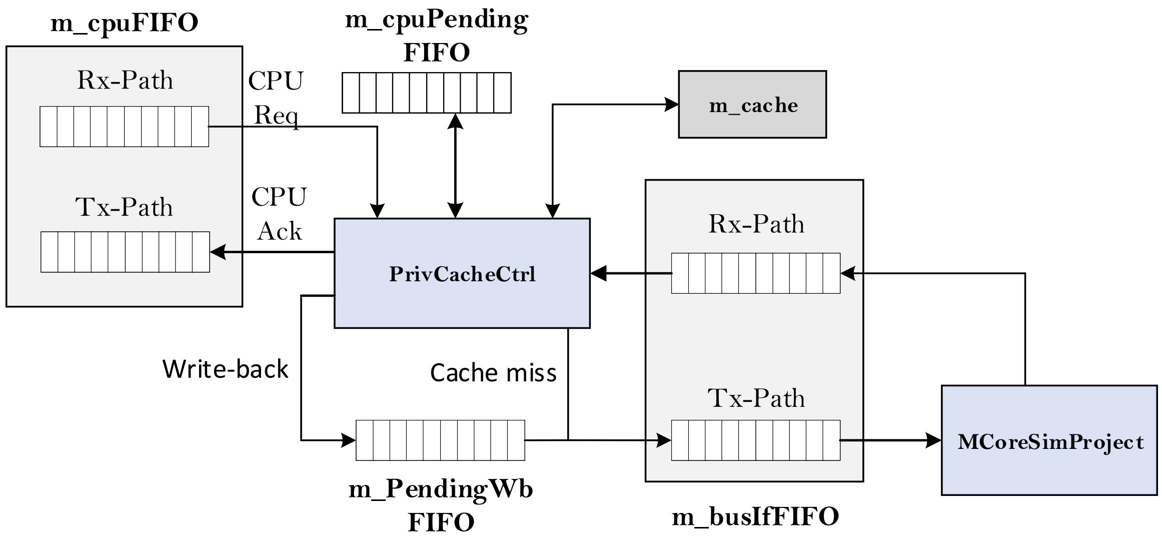


Figure 4.3: PrivCacheCtrl queues structure

```
class PrivCacheCtrl : public ns3::Object {
  private:
    Ptr<CpuFIFO>   m_cpuFIFO; // pointer to cpu I/F FIFOs
    Ptr<BusIfFIFO> m_busIfFIFO; // pointer to bus I/F FIFOs
    Ptr<GenericCache> m_cache; // pointer to cache memory
    // internal buffers
    Ptr<GenericFIFO <PendingMsg >> m_cpuPendingFIFO;
    GenericQueue <BusIfFIFO::BusReqMsg> m_PendingWbFIFO;
    // pointer to cache coherence ctrl
    Ptr<ns3::SNOOPPrivCohProtocol> m_cohProtocol;
    // process write back msgs
    bool DoWriteBack (uint64_t addr, uint16_t wbCoreId,
                      uint64_t msgId, bool dualTrans);
    // cachectrl main function
  void CacheCtrlMain ();
public:
  // set maximum number of pending cpu requests
  void SetMaxPendingReq (int maxPendingReq);
  // set coherence protocol type
  void SetProtocolType (CohProtType ptype);
  // initialization function
  void init();
  // callback to run CacheCtrlMain()
  static void Step(Ptr<PrivCacheCtrl> privCacheCtrl);
  // other methods
  ...
};
```

Code 4.7: PrivCacheCtrl class interface

4.2.7 Shared Cache Controller

`SharedCacheCtrl` class works similar to `PrivCacheCtrl`. Code 4.8 shows a simplified interface of the `SharedCacheCtrl` class. We use Figure 4.4 to illustrate queues structure and connection flow inside `SharedCacheCtrl` class. The class has two external interfaces with the top-level.i) `m_busIfFIFO` represents L1 interconnect interface that is used for receiving memory requests from the private cache controllers. ii) `m_dramBusIfFIFO` represents DRAM interface that is used for sending and receiving request to the DRAM controller. `SharedCacheCtrl` has access to three internal memory components: Shared L2 cache memory (i.e. `m_cache`), victim cache (i.e. `m_victimCache`), and write-back (i.e. `m_PndWBFIFO`) buffer. `SharedCacheCtrl` uses

`DoWriteBack()` methods to send messages on the L1 interconnect bus. These messages can be either data or coherence messages such as invalidation or exclusive response. `CacheCtrlMain()` is the main control function which is responsible for observing incoming messages on the external interfaces (DRAM and L1 bus interfaces), runs coherence protocol FSM using `CohProtFSMProc()` method, and decides what actions need to be taken. These actions can be either one or multiple decisions such as sending data or coherence message to the requestor on the L1 bus, sending memory request to DRAM controller using `SendDRAMReq()`, or evicting a specific cache line from victim cache using `VictimCacheLineEvict()`. Similar to `PrivCacheCtrl`, the `init()` and `Step()` methods are used by the top-level module to initialize and schedule callback events. respectively.

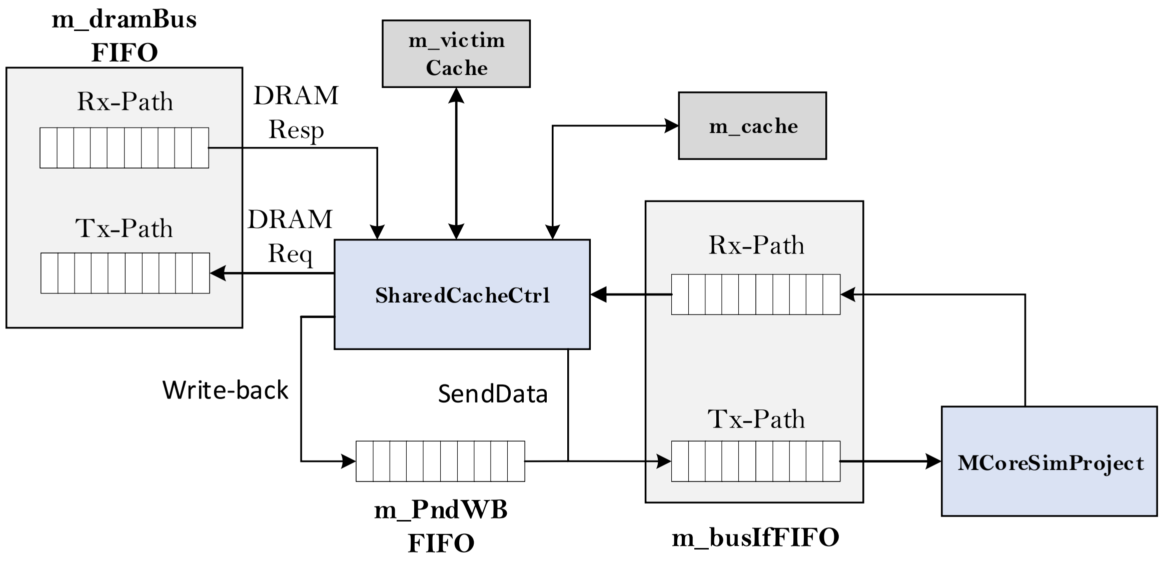


Figure 4.4: SharedCacheCtrl queues structure

```
class SharedCacheCtrl : public ns3::Object {
  private:
    // send data type
    enum SendDataType {
        DataOnly = 0,
        ExclOnly,
        DataPlsExcl,
        CoreInv
    };
    // external interfaces
    Ptr<BusIfFIFO> m_busIfFIFO; // bus I/F FIFOs
    Ptr<DRAMIfFIFO> m_dramBusIfFIFO; // DRAM I/F FIFOs
    // internal memories
    GenericFIFO <BusIfFIFO::BusReqMsg> m_PndWBFIFO; // wb buffer
    Ptr<GenericCache> m_cache; // L2 cache memory
    Ptr<VictimCache> m_victimCache; // victim cache
    // process write-back messages
    bool DoWriteBack (uint64_t cl_idx, uint16_t wbCoreId,
                      uint64_t msgId , SendDataType type);
    // controller main function
    void CacheCtrlMain ();
    // sharedctrl coherence FSM
    void CohProtFSMProc (SNOOPSharedEventType eventType,
                         SNOOPSharedOwnerState owner, int state);
    // send DRAMReq
    bool SendDRAMReq (uint64_t msgId, uint64_t addr,
                      DRAMIfFIFO::DRAM_REQ type);
    // victim cache eviction
    void VictimCacheLineEvict (uint32_t victimWayIdx);
    // other methods
    ...
  public:
    // initialization function
    void init();
    // callback function
    static void Step(Ptr<SharedCacheCtrl> sharedCacheCtrl);
    // other methods
    ...
```

Code 4.8: `SharedCacheCtrl` class interface

4.2.8 Cache Coherence Protocol FSM

Cache coherence protocol is used to maintain coherence of the shared data stored in the private cache hierarchies of the multi-core system. Unlike existing state-of-the-art simulators [13, 14, 15, 16], CacheSim supports detailed coherence models including both stable and transient states to support an independent operation of the underlying interconnect network. Our coherency models also consider the cache inclusivity feature, as it is intertwined with the protocol operation. To the best of our knowledge, existing coherence models either completely ignore the implementation of the coherence transient states assuming that the underlying interconnect is atomic or implementing partial transient states that alter the modularity of the interconnect. Private cache coherence protocols are implemented using finite state machines (FSM) which can be defined formally as a tuple (P, Σ, T, Q, E, C), where:

- $P \in$ `State<T>` is the present state
- $T : P \times \Sigma \rightarrow P$ is the transition function
- $\Sigma \in$ `SNOOPPrivEventList` is the finite set of input
- $Q \in$ `State<T>` is the next state

– $E \in$ `SNOOPPrivCtrlAction` is the event output

– $C \in$ `SNOOPPrivCohTrans` is the condition output

The state definition (i.e. P, and Q) and transition function T are protocol dependant, while Σ, E, and C are almost common among all snooping coherence protocols. Σ represents the input event to the coherence protocol, which can be generated by the processor or interconnect as discussed before. E is the list of actions the cache controller needs to perform. Finally, C is the coherence transaction to be sent on the bus as discussed in Subsection 2.3.1.2. Table 4.1 shows how state transition Q, event output E, and C are computed based on the input values P, and Σ for MESI coherence protocol which can be represented as follow:

– transition: $Q := T : P \times \Sigma \rightarrow$ `State<T>`

– output: $out := Q, C_{action}, C_{tr}$ where

$$C_{action} := (E : P \times \Sigma \rightarrow \texttt{SNOOPPrivCtrlAction}),$$

$$C_{tr} := (C : P \times \Sigma \rightarrow \texttt{SNOOPPrivCohTrans})$$

The same analysis can be used to model the coherence protocol at the shared cache controller side using state transition Table 4.2 for MESI. Compared to MSI protocol that we discussed in Section 2.3.1.5, MESI introduces few extra states to the protocol to optimize the performance of the read-modify-write scenario. The protocol introduces a stable exclusive state E to enable the cache controller to upgrade cache line permission from read-only to read-write without the need to issue $GetM()$ transaction on the bus. The cache controller does this upgrade only if there are no other sharers to the requested cache line. Therefore, E state is introduced to communicate this information to the controller. Whenever the cache controller issues GetS() transaction to acquire read-only permission for a certain cache line, the shared memory controller checks the line's state in its L2 cache. If it has the line in I state, then the controller responds with exclusive notification to the requestor core as shown in Table 4.1 **Own Excl()**. Once the cache controller observes this message, it updates the line state from IS^d to IE^d, then moves to E once it receives the date response. The transient states IE^dS, IE^dSI, and IE^dI are introduced to support non-atomic bus architectures as discussed in Section 2.4. The E state is considered an ownership state in our simulator. Therefore, the transient EI^a state is introduced to handle block replacements in state E.

Invalidate() coherence message is also a new modification that is added to support cache inclusivity feature when the shared cache controller decides to evict a cache line from its L2 cache. Upon receiving the invalidation message, the private cache controllers need to invalidate their cached versions, and only the owner should write-back the data to L2 cache. The transient states $EorMI^a$ and $IEorM^d$ are added to support the write-back operation for pending data response cache lines.

Table 4.1: MESI snooping protocol state table at cache controller side

State	**Core Event**			**Bus Event**								
	Load	**Store**	**Replace**	**OwnGetS**	**OwnGetM**	**OwnPutM**	**OtherGetS**	**OtherGetM**	**OtherPutM**	**Own data response**	**Invalidate**	**Own Excl**
I	*issue GetS/* IS^{ad}	*issue GetM/* IM^{ad}	X	X	X	X	-	-	-	X	-	X
IS^{ad}	*stall*	*stall*	*stall*	-/IS^d	X	X	-	-	-	-/IS^a	-	X
IS^d	*stall*	*stall*	*stall*	X	X	X	-	-/ IS^dI	-	*Hit/S*	-/ IS^dI	-/ IE^d

$\boldsymbol{IE^d}$	*stall*	*stall*	*stall*	*X*	*X*	*X*	IE^dS	-/ IE^dI	-	*Hit/E*	*issue PutM/* $IEorM^dI$	*X*
$\boldsymbol{IE^dS}$	*stall*	*stall*	*stall*	*X*	*X*	*X*	-	-/ IE^dSI	-	*Hit SendData /S*	-/ IE^dSI	*X*
$\boldsymbol{IE^dSI}$	*stall*	*stall*	*stall*	*X*	*X*	*X*	-	-	-	*Hit SendData /I*	-	*X*
$\boldsymbol{IE^dI}$	*stall*	*stall*	*stall*	*X*	*X*	*X*	-	-	-	*Hit SendData /I*	-	*X*
$\boldsymbol{IS^a}$	*stall*	*stall*	*stall*	*Hit/S*	*X*	*X*	-	-	*X*	*X*	*X*	*X*
$\boldsymbol{IS^dI}$	*stall*	*stall*	*stall*	*X*	*X*	*X*	-	-	-	*Hit/I*	-	*X*
$\boldsymbol{IE^a}$	*stall*	*stall*	*stall*	*Hit/E*	*X*	*X*	-	-	*X*	*X*	*X*	*X*
$\boldsymbol{IM^{ad}}$	*stall*	*stall*	*stall*	*X*	-/IM^d	*X*	-	-	-	-/IM^a	-	*X*
$\boldsymbol{IM^d}$	*stall*	*stall*	*stall*	*X*	*X*	*X*	-/ IM^dS	-/ IM^dI	-	*Hit/M*	*issue PutM/* $IEorM^dI$	*X*
$\boldsymbol{IM^a}$	*stall*	*stall*	*stall*	*X*	*Hit/M*	*X*	-	-	-	*X*	*X*	*X*
$\boldsymbol{IM^dI}$	*stall*	*stall*	*stall*	*X*	*X*	*X*	-	-	-	*Hit SendData /I*	-	*X*
$\boldsymbol{IM^dS}$	*stall*	*stall*	*stall*	*X*	*X*	*X*	-	-/ IM^dSI	-	*Hit SendData /S*	-/ IM^dSI	*X*
$\boldsymbol{IM^dSI}$	*stall*	*stall*	*stall*	*X*	*X*	*X*	-	-	-	*Hit SendData /I*	-	*X*
$\boldsymbol{S}$	*Hit*	*issue GetM/* SM^{ad}	-/*I*	*X*	*X*	*X*	-	-/*I*	-	*X*	-/*I*	*X*
$\boldsymbol{SM^{ad}}$	*Hit*	*stall*	*stall*	*X*	- /SM^d	*X*	-	-/ IM^{ad}	-	-/SM^a	-/IM^{ad}	*X*
$\boldsymbol{SM^d}$	*Hit*	*stall*	*stall*	*X*	*X*	*X*	-/ SM^dS	-/ SM^dI	-	*Hit/ M*	*issue PutM/* $IEorM^dI$	*X*
$\boldsymbol{SM^a}$	*Hit*	*stall*	*stall*	*X*	*Hit/M*	*X*	-	- /IM^a	*X*	*X*	*X*	*X*
$\boldsymbol{SM^dI}$	*Hit*	*stall*	*stall*	*X*	*X*	*X*	-	-	-	*Hit SendData /I*	-	*X*

SM^{d}S	*Hit*	*stall*	*stall*	*X*	*X*	*X*	-	-/ *SM^{d}SI*	-	*Hit SendData /S*	-/ *SM^{d}SI*	*X*
SM^{d}SI	*Hit*	*stall*	*stall*	*X*	*X*	*X*	-	-	-	*Hit SendData /I*	-	*X*
E	*Hit*	*Hit/ M*	*issue PutM/ EI^a*	*X*	*X*	*X*	*Send Data/ S*	*Send Data / I*	-	*X*	*issue PutM/ EorMI^a*	*X*
M	*Hit*	*Hit*	*issue PutM/ MI^a*	*X*	*X*	*X*	*Send Data/ S*	*Send Data / I*	-	*X*	*issue PutM/ EorMI^a*	*X*
MI^a	*Hit*	*Hit*	*stall*	*X*	*X*	*Send Data/ I*	*Send Data/ II^a*	*Send Data / II^a*	-	*X*	-/ *EorMI^a*	*X*
EI^a	*Hit*	*stall*	*stall*	*X*	*X*	*Send Data/ I*	*Send Data/ II^a*	*Send Data / II^a*	-	*X*	-/ *EorMI^a*	*X*
II^a	*stall*	*stall*	*stall*	*X*	*X*	-/*I*	-	-	-	*X*	-	*X*
EorMI^a	*stall*	*stall*	*stall*	*X*	*X*	*Send Data/ I*	-	-	*X*	*X*	*X*	*X*
IEorM^d	*stall*	*stall*	*stall*	*X*	*X*	–	-	-	*X*	*Hit Send Data/I*	*X*	*X*

Table 4.2: MESI snooping protocol state table at LLC controller side

State	**Bus Event**					**Ctrl Event**	
	GetS	***GetM***	***Owner PutM***	***Other PutM***	***Data***	***Replc***	***DRAM Req***
I	*SendExecData, SetOwner/ EorM*	*SendData, SetOwner/ EorM*	*X*	-	*X*	-	*DRAM^d*
S	*SendData*	*SendData, SetOwner/ EorM*	*X*	-	*X*	*issue Inv / I*	*DRAM^d*
EorM	*ClearOwner/ EorM^{d}S*	*SetOwner / C2C ? - : EorM^{d}EorM*	*ClearOwner/ EorM^{d}I*	-	*StoreData/ EorM^a*	*issue Inv / EorM^{d}I*	*DRAM^d*
EorM^{d}S	*stall*	*stall*	*X*	-	*StoreData/ S*	*stall*	*DRAM^d*

EorM*^{d}*EorM	*stall*	*stall*	*stall*	-	*StoreData/ EorM*	*stall*	$DRAM^d$
EorM*^{d}*I	*SendExecResp / -*	-	*X*	-	*StoreData/ I*	*stall*	$DRAM^d$
***EorM*^{a}**	*clearOwner/ S*	*SetOwner/ EorM*	*ClearOwner/ I*	-	*X*	*X*	$DRAM^d$
***DRAM*^{d}**	*stall*	*stall*	*X*	*X*	*I*	*X*	*stall*

In addition to MSI and MESI, CacheSim also supports MOESI coherence protocol where Tables 4.3 and 4.4 illustrate coherence protocol FSM at the private cache controller and shared cache controller, respectively. As discussed in Section 2.3.1, MOSEI introduced O state to optimize the performance of the multi-core systems that support cache-to-cache (C2C) data transfer where it eliminates the unnecessary write-back to the shared memory. Similar to EI^a state, the OI^a handles the replacement of cache line in O state. The transient OM^a state handles upgrades from O to M state. The transient states IE^dO, IM^dO, and SM^dO are introduced to support non-atomic bus architectures. Moreover, O^dI and OM^aI states are added to support pending invalidations.

Table 4.3: MOESI snooping protocol state table at cache controller side

State	**Core Event**			**Bus Event**								
	Load	**Store**	**Replace**	**OwnGetS**	**OwnGetM**	**OwnPutM**	**OtherGetS**	**OtherGetM**	**OtherPutM**	**Own data response**	**Invalidate**	**Own Excl**
I	*issue GetS/* IS^{ad}	*issue GetM/* IM^{ad}	*X*	*X*	*X*	*X*	-	-	-	*X*	-	*X*
***IS*^{ad}**	*stall*	*stall*	*stall*	-/IS^d	*X*	*X*	-	-	-	-/IS^a	-	*X*
***IS*^{d}**	*stall*	*stall*	*stall*	*X*	*X*	*X*	-	-/ IS^dI	-	*Hit/S*	-/ IS^dI	-/ IE^d
***IE*^{d}**	*stall*	*stall*	*stall*	*X*	*X*	*X*	-/ IE^dO	-/ IE^dI	-	*Hit/E*	*issue PutM/* $IEorM^dI$	*X*

IE^dO	*stall*	*stall*	*stall*	*X*	*X*	*X*	-	-/ IE^dI	-	*Hit SendData / O*	*issue PutM/* O^dI	*X*
IE^dI	*stall*	*stall*	*stall*	*X*	*X*	*X*	-	-	-	*Hit SendData /I*	-	*X*
IS^a	*stall*	*stall*	*stall*	*Hit/S*	*X*	*X*	-	-	*X*	*X*	*X*	*X*
IS^dI	*stall*	*stall*	*stall*	*X*	*X*	*X*	-	-	-	*Hit/I*	-	*X*
IE^a	*stall*	*stall*	*stall*	*Hit/E*	*X*	*X*	-	-	*X*	*X*	*X*	*X*
IM^{ad}	*stall*	*stall*	*stall*	*X*	-/IM^d	*X*	-	-	-	-/IM^a	-	*X*
IM^d	*stall*	*stall*	*stall*	*X*	*X*	*X*	-/ IM^dO	-/ IM^dI	-	*Hit/M*	*issue PutM/* $IEorM^dI$	*X*
IM^a	*stall*	*stall*	*stall*	*X*	*Hit/M*	*X*	-	-	-	*X*	*X*	*X*
IM^dI	*stall*	*stall*	*stall*	*X*	*X*	*X*	-	-	-	*Hit SendData /I*	-	*X*
IM^dO	*stall*	*stall*	*stall*	*X*	*X*	*X*	-	-/ IM^dI	-	*Hit SendData /O*	*issue PutM/* O^dI	*X*
S	*Hit*	*issue GetM/* SM^{ad}	-/*I*	*X*	*X*	*X*	-	-/*I*	-	*X*	-/ *I*	*X*
SM^{ad}	*Hit*	*stall*	*stall*	*X*	- /SM^d	*X*	-	-/ IM^{ad}	-	-/SM^a	-/IM^{ad}	*X*
SM^d	*Hit*	*stall*	*stall*	*X*	*X*	*X*	-/ SM^dO	-/ SM^dI	-	*Hit/ M*	*issue PutM/* $IEorM^dI$	*X*
SM^a	*Hit*	*stall*	*stall*	*X*	*Hit/M*	*X*	-	- /IM^a	*X*	*X*	*X*	*X*
SM^dI	*Hit*	*stall*	*stall*	*X*	*X*	*X*	-	-	-	*Hit SendData /I*	-	*X*
SM^dO	*Hit*	*stall*	*stall*	*X*	*X*	*X*	-	-/ SM^dI	-	*Hit SendData /O*	*issue PutM/* O^dI	*X*
E	*Hit*	*Hit/ M*	*issue PutM/* EI^a	*X*	*X*	*X*	*Send Data/ O*	*Send Data / I*	-	*X*	*issue PutM/* $EorMI^a$	*X*
O	*Hit*	*issue GetM/* OM^a	*issue PutM/* OI^a	*X*	*X*	*X*	*Send Data*	*Send Data / I*	-	*X*	*issue PutM/* $EorMI^a$	*X*

OM^a	*Hit*	*stall*	*stall*	*X*	*Hit/M*	*X*	*Send Data*	*Send Data/ IM^{ad}*	-	*X*	*issue PutM/ OM^aI*	*X*
M	*Hit*	*Hit*	*issue PutM/ MI^a*	*X*	*X*	*X*	*Send Data/ O*	*Send Data / I*	-	*X*	*issue PutM/ $EorMI^a$*	*X*
MI^a	*Hit*	*Hit*	*stall*	*X*	*X*	*Send Data/ I*	*Send Data/ OI^a*	*Send Data / II^a*	-	*X*	*-/ $EorMI^a$*	*X*
OI^a	*Hit*	*Hit*	*stall*	*X*	*X*	*Send Data/ I*	*Send Data*	*Send Data / II^a*	-	*X*	*-/ $EorMI^a$*	*X*
EI^a	*Hit*	*stall*	*stall*	*X*	*X*	*Send Data/ I*	*Send Data/ OI^a*	*Send Data / II^a*	-	*X*	*-/ $EorMI^a$*	*X*
II^a	*stall*	*stall*	*stall*	*X*	*X*	*-/I*	-	-	-	*X*	-	*X*
$EorMI^a$	*stall*	*stall*	*stall*	*X*	*X*	*Send Data/ I*	-	-	*X*	*X*	*X*	*X*
$IEorM^d$	*stall*	*stall*	*stall*	*X*	*X*	–	-	-	*X*	*Hit Send Data/I*	*X*	*X*
O^dI	*stall*	*stall*	*stall*	*X*	*X*	–	-	-	*X*	*Hit Send Data/I*	*X*	*X*
OM^aI	*stall*	*stall*	*stall*	*X*	*Hit Send Data/I*	*X*	-	-	*X*	*X*	*X*	*X*

Table 4.4: MOESI snooping protocol state table at LLC controller side

State	**Bus Event**					**Ctrl Event**	
	GetS	***GetM***	***Owner PutM***	***Other PutM***	***Data***	***Replc***	***DRAM Req***
I	*SendExecData, SetOwner/ EorM*	*SendData, SetOwner/ EorM*	*X*	-	*X*	-	*$DRAM^d$*
S	*SendData*	*SendData, SetOwner/ EorM*	*X*	-	*X*	*issue Inv / I*	*$DRAM^d$*
EorM	*O*	–	*ClearOwner/ $EorM^dI$*	-	*StoreData/ $EorM^a$*	*issue Inv / $EorM^dI$*	*$DRAM^d$*

O	-	*SetOwner/ EorM*	O^dS	-	O^a	*issue Inv / O^dS*	$DRAM^d$
$\boldsymbol{O^dS}$	*stall*	*stall*	*X*	-	*StoreData/ S*	*stall*	$DRAM^d$
$\boldsymbol{EorM^dI}$	*SendExecResp / -*	*stall*	*X*	-	*StoreData/ I*	*stall*	$DRAM^d$
$\boldsymbol{EorM^a}$	*X*	*X*	*ClearOwner/ I*	*X*	*X*	*X*	$DRAM^d$
$\boldsymbol{O^a}$	*X*	*X*	*ClearOwner/ S*	*X*	*X*	*X*	$DRAM^d$
$\boldsymbol{DRAM^d}$	*stall*	*stall*	*X*	*X*	*I*	*X*	*stall*

In order to provide an extensible implementation for cache coherence protocols, CacheSim models these protocols as a virtual interface to give users the flexibility to implement and integrate new protocols into the simulator with minimal effort. Code 4.9 shows the definition of `IFCohProtocol` class which includes the definition of private and shared cache coherence state machines using `SNOOPPrivCohrProc()` and `SNOOPSharedCohrProc()` methods. These virtual functions are type-casted dynamically based on the coherence protocol type `CohProtType` provided in the XML document as shown in Code 4.10.

```
class IFCohProtocol {
 public:
   // private cache coherence FSM processing
   virtual void SNOOPPrivCohrProc (
                       SNOOPPrivEventType   eventType,
                       SNOOPPrivEventList   eventList,
                       int                  &cacheState,
                       SNOOPPrivCohTrans    &trans2Issue,
                       SNOOPPrivCtrlAction  &ctrlAction,
                       bool                 cache2Cache
   ) {}
```

```
    // shared cache coherence FSM processing
    virtual void SNOOPSharedCohrProc (
                     SNOOPSharedEventType  eventType,
                     SNOOPSharedEventList  eventList,
                     int                   &cacheState,
                     SNOOPSharedOwnerState &ownerState,
                     SNOOPSharedCtrlAction &ctrlAction,
                     bool                  cache2Cache,
    ){}
    // other methods
    ...
};
```

Code 4.9: `IFCohProtocol` class interface

```
// Coherence FSM function call
void SNOOPPrivCohProtocol::CohProtocolFSMProcessing () {
  // Check protocol type
  IFCohProtocol *ptr;
  switch (m_pType) {
  case CohProtType::SNOOP_PMSI:
    ptr = new PMSI;  break;
  case CohProtType::SNOOP_MSI:
    ptr = new MSI;   break;
  case CohProtType::SNOOP_MESI:
    ptr = new MESI;  break;
  case CohProtType::SNOOP_MOESI:
    ptr = new MOESI; break;
  default:
    std::cout << "unknown snooping protocol type" << std::endl;
    exit(0);
  }
  // dynamic binding
  ptr->SNOOPPrivCohrProc (
     m_processEvent, m_eventList, m_currEventNextState,
     m_currEventTrans2Issue, m_ctrlAction,  m_cache2Cache
  );
  delete ptr;
}
```

Code 4.10: Coherence protocol dynamic binding

The actual implementations of the underlying coherence protocols are coded into separate classes. Each class implements a single protocol to support the design for change principle discussed in the introduction section (i.e. one secret per module). Code 4.11 shows `MSI` class which includes state definition and coherence protocol FSM implementation.

```
class MSI : public ns3::Object, public ns3::IFCohProtocol {
public:
// MSI protocol states encoding (Private Cache Side)
enum class SNOOP_MSIPrivCacheState {
  I = CohProtType::SNOOP_MSI,
  IS_ad,
  IS_d,
  IS_a,
  IS_d_I,
  IM_ad,
  // other states
  ...
  };
  // MSI protocol states encoding  (Memory Controller Side)
  enum class SNOOP_MSISharedCacheState {
    IorS = CohProtType::SNOOP_MSI,
    M,
    M_d_M,
    M_d_IorS,
    IorSorM_a,
    DRAM_d
  };
  // coherence protocol FSM implementations
  void MSI::SNOOPPrivCohrProc (...);
  void MSI::SNOOPSharedCohrProc (...);
  // other methods
  ...
};
```

Code 4.11: `MSI` class interface

4.2.9 Generic Cache Memory

To support a broad range of cache configurations, CacheSim has a general interface to model cache memory based on the input configurations. `GenericCache` class allocates memory to store cached data, provides methods to load, store, and replace data from a specific cache line inside this memory. The mapping between data stored in a certain cache-level and data stored in the upper-level is defined based on the input configuration parameters. `WriteCacheLine()` and `ReadCacheLine()` functions are used to read/write data into the cache memory. `CpuAddrMap()` function is used to map physical memory request to cache line/set index. `GetReplacementLine()` function is responsible to replace old cache blocks with the new one according to a certain policy. CacheSim implements all replacement algorithms that we discussed in Subsection 2.2.3.

4.2.10 Bus Arbiters

`BusArbiter` class arbitrates coherence and data response messages generated by PrivCacheCtrl and SharedCacheCtrl instances over the shared interconnect bus. We assume a system model like the one illustrated in Figure 4.5 where the interconnect is modeled as two split bi-directional buses that operate independently. The request bus is responsible for broadcasting coherence transactions (CohrTrans), while the data response bus is used to carry over the data response (DataResp) messages. BusArbiter class implements two methods `ReqStep()` and `RespStep()` that work independently to arbitrate these messages on the interconnect. The request and response bus latencies, in addition to the deployed arbitration schemes, are fully configurable by the top-level module. CacheSim supports both predictable and high-performance arbiters that we discussed in Section 2.4.1 and Chapter 5. Code 4.12 depicts `BusArbiter` class interface. `BusArbiter` has access to all cache controller's bus interface buffers (i.e. `m_busIfFIFO` and `m_sharedCacheBusIfFIFO`) in addition to the interconnect FIFOs `m_interConnectFIFO`. The top-level module (i.e. `MCoreSimProject`) sets a reference to these buffers during the construction of the `busArbiter` such that it can read and write them. The request and response arbiters use `SendMemCohrMsg()` and `SendData()`, respectively, to send the data on the bus. `MCoreSimProject` uses a set of `Get()` functions to configure `busArbiter` parameters. For instance, `SetBusArchitecture()` is used to set the bus architecture. CacheSim supports both unified and split architectures. `SetBusArbitration` sets the arbitration schemes used by `ReqStep()` and `RespStep()`. `MCoreSimProject` also invokes `init()` at the beginning of the simulation in order to initialize the arbiter and schedule callback events for `ReqStep()` and `RespStep()` using `Step()` method.

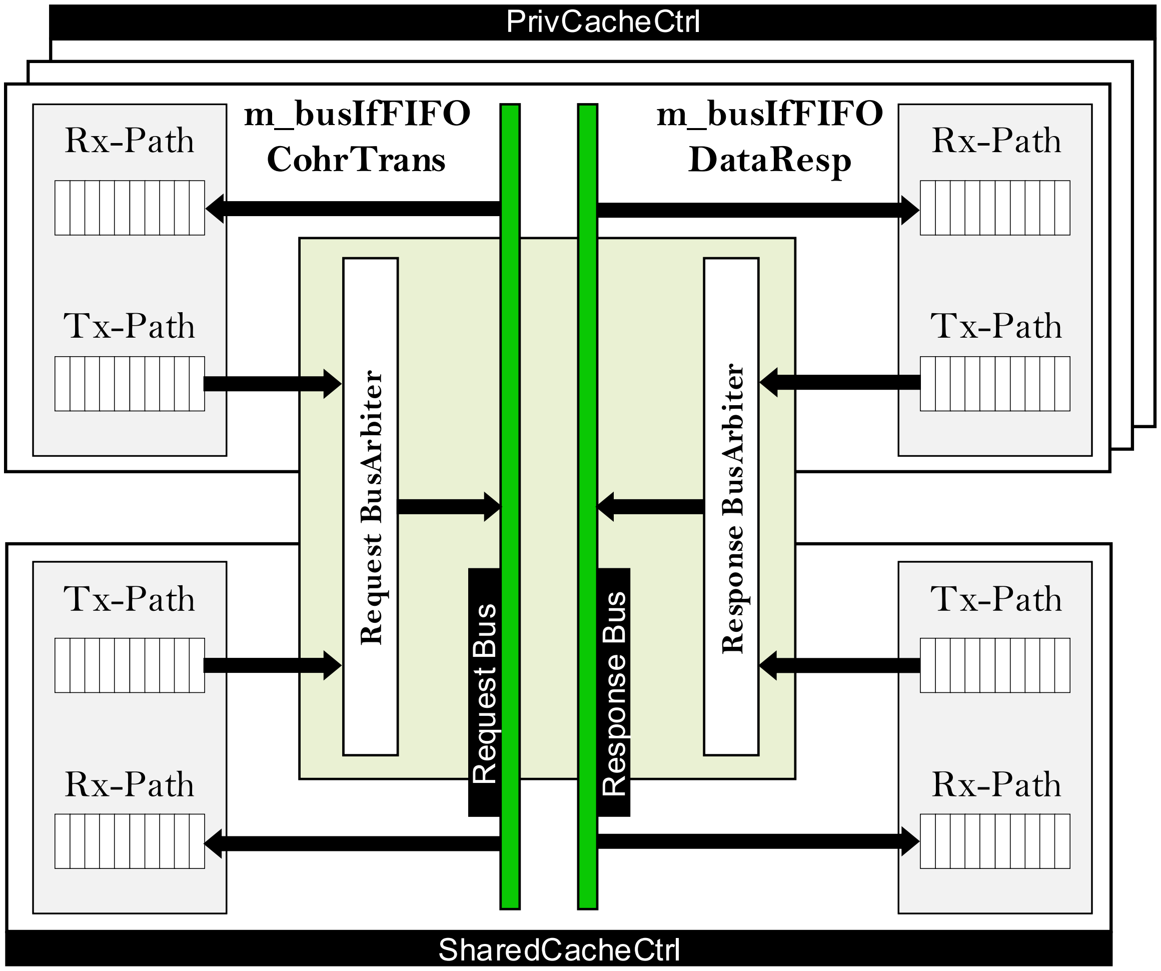


Figure 4.5: Interconnect bus system model

```
class BusArbiter : public ns3::Object {
  private:
    uint16_t m_reqlatency;  // requst bus latency
    uint16_t m_resplatency; // response bus latency
    // a list of PrivCachCtrl bus interface buffers
    std::list<ns3::Ptr<ns3::BusIfFIFO> > m_busIfFIFO;
    // a pointer to SharedCachCtrl bus IF buffers
    ns3::Ptr<ns3::BusIfFIFO> m_sharedCacheBusIfFIFO;
    // a pointer to Interconnect FIFOs
    ns3::Ptr<ns3::InterConnectFIFO>  m_interConnectFIFO;
    // send data response message on the bus
    void SendData (BusIfFIFO::BusRespMsg msg, AGENT agent);
    // send coherence messages on the bus
    void SendMemCohrMsg (BusIfFIFO::BusReqMsg msg, bool BroadCast);
    // other methods
    ...
  public:
    void init(); // initialize function
    // callback function to schedule request arbiter
    static void ReqStep(Ptr<BusArbiter> busArbiter);
    // callback function to schedule response arbiter
    static void RespStep(Ptr<BusArbiter> busArbiter);
    // scheduling bus events every dt
    static void Step(Ptr<BusArbiter> busArbiter);
    // set bus architecture
    void SetBusArchitecture (string bus_arch);
    // set arbitration policy
    void SetBusArbitration (string bus_arb);
    // other methods
    ...
};
```

Code 4.12: `BusArbiter` class interface

Chapter 5

PISCOT

We now show the effectiveness of CacheSim by utilizing it to prototype a new coherent interconnect solution that allows for coherent sharing of data in multi-core real-time systems without requiring any changes to the coherence protocols while notoriously reducing the worst-case latency upon accessing the cache hierarchy. We start our discussion in Section 5.1 by the case study we made in Section 2.4.1 about the existing predictable cache-coherent solution and show the limitation of these solutions. Then motivated by this discussion, we propose PISCOT, a predictable and coherent bus architecture that substantially reduces coherence delays, while improving overall system performance (Section 5.2). In Section 5.3, we conduct a detailed timing analysis for the latency suffered by any memory request under PISCOT. The analysis provides an analytical bound that guarantees the system predictability. The derived bounds are 4× tighter than the state-of-art predictable coherent buses [1, 24, 25]. Finally, we use our simulator to evaluate PISCOT with both the representative SPLASH-3 benchmarks as well as synthetic benchmarks. The results in Subsection 6.2.2.3 show that compared with existing solutions, PISCOT achieves up to 5× better performance

(2.8× on average), while increasing memory bandwidth utilization by 12× on average across the SPLASH-3 benchmarks.

5.1 Motivation

Two key observations we make in Section 2.4.1 about the existing predictable cache-coherent TDM-based solutions. First, their previously highlighted large WCLs are mainly because they inherit the scheduling paradigm of traditional real-time arbiters (such as TDM in this case but the argument applies to other arbiters such as RR). This paradigm when applied to systems with shared data, it couples two different types of communication into the same bus arbitration. Namely, it couples both coherence messages and data transfers and schedules them using the same bus arbitration, which is inherited from traditional non-data-sharing TDM schedules. This in addition to the fact that the TDM slot has to accommodate for at least one memory transfer to be efficient to service ready memory requests, leading to the excessively large memory delays when introducing data sharing. Second, they impose certain modifications to the coherence protocol to enable predictability. As previously discussed, modifications to coherence protocols are highly costly in terms of verification and are thus inconceivable to adopt by industry. Based on these observations, PISCOT targets to enable data sharing in real-time systems, while significantly reducing the associated coherence delays by decoupling the two different communication types. This is achieved by using a split-bus architecture, where requests (through coherence messages) and responses (i.e. data transfers) are issued in different buses and are managed using different arbitration mechanisms. In addition, PISCOT does not impose any changes to existing coherence protocols; therefore, disburden system designers from the need

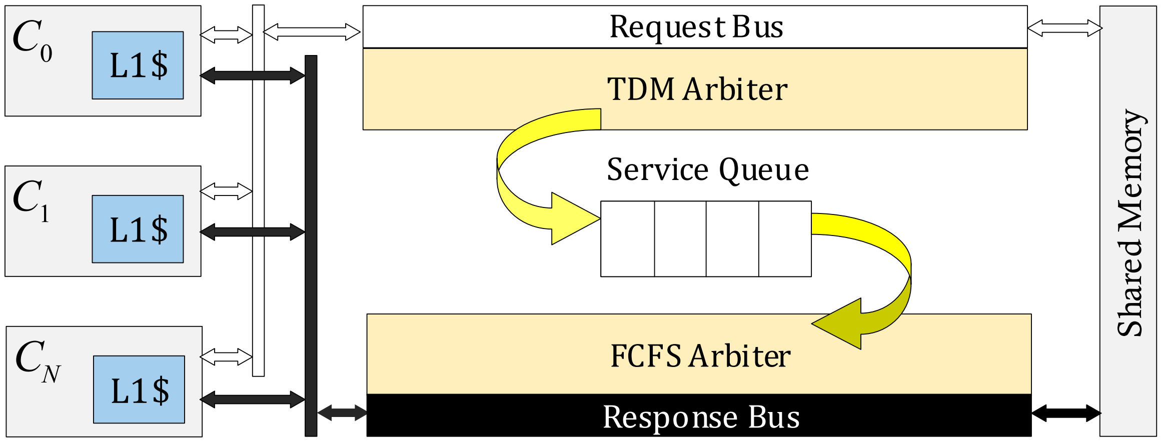


Figure 5.1: PISCOT architecture.

to re-verify the coherence protocol.

5.2 Proposed solution

In this section, we detail the architectural details of PISCOT, which Figure 5.1 delineates its high-level modules. Compared to the solutions discussed in Section 2.4.1 and highlighted in Table 2.4, PISCOT makes multiple architecture decisions to take into account predictability by design, while maintaining a high average-case performance.

- PISCOT's architecture migrates from the traditional arbitration schemes considered by the community (such as TDM and RR) to a split-transaction bus interconnect that connects private caches and the shared memory as Rule 1 explains.

Rule 1 PISCOT *implements a split-transaction bus through deploying two buses: a* `Request Bus` *and a* `Response Bus`*. The* `Request Bus` *is responsible for broadcasting the coherence messages initiating memory requests, while the* `Response Bus` *transfers data as a response to these requests.*

- Aiming at performance, the `Request Bus` and the `Response Bus` operate in parallel. On the other hand, to simplify system analysis and maintain predictability, both buses communicate through only one module: the `Service Queue`. Requests broadcasted on the `Request Bus` are buffered into the `Service Queue` until they are selected by the `Response Bus`'s arbiter.

- Unlike conventional solutions that use high-performance arbiters at the expense of predictability (e.g. FCFS), the `Request Bus` in PISCOT is managed using a TDM arbiter to predictably manage interference among different cores (Rule 2). To increase system performance, a work-conserving TDM is deployed, where at any slot, if the dedicated core does not have a ready request, the arbiter picks the next core with a pending request instead of leaving the slot idle as in traditional non work-conserving TDM.

Rule 2 PISCOT *manages the* ***`Request Bus`*** *using a work-conserving TDM arbiter.*

- The `Response Bus`'s arbiter implements a First-Come First-Serve (FCFS) scheduler, and thus, serves requests based on their arrival time on the `Service Queue` (Rule 3). The oldest request will be at the head of the queue, and therefore, is serviced first by the FCFS response arbiter. Once selected by the FCFS arbiter, the requested data is transferred on the `Response Bus` to the requesting core's private cache, the request message is removed from the `Service Queue`, then the core proceeds with its load/store operation indicating that the request is successfully finished.

Rule 3 PISCOT *manages the* ***`Response Bus`*** *using a FCFS arbiter.*

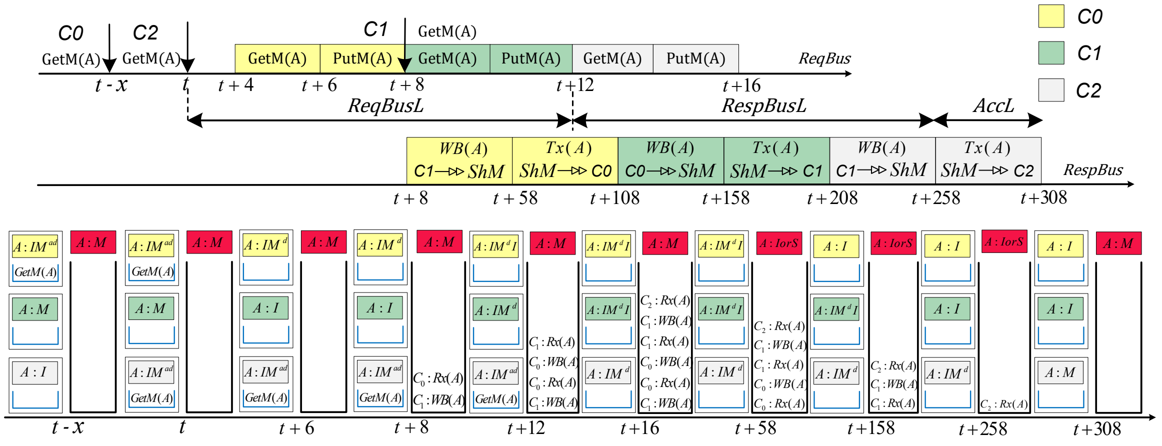


Figure 5.2: An illustrative example for the operation of PISCOT. Latency components are for the getM(A) request from $C2$. At different time instances: the bottom of the figure shows the state of the private core's cache line (left side), shared memory state (marked in red), and the Service Queue contents on the right side.

- PISCOT supports out-of-order execution and allows cores to have multiple outstanding requests. Nonetheless, according to Rule 4, those requests from a certain core would remain in its local buffer and will not be picked by the TDM arbiter if the core already has one request in-service (i.e. queued in the Service Queue). The rationale for this is to limit the coherence interference among cores such that a request from any core can suffer interference due to a maximum of only one request from each other core, which leads to tightening worst-case latencies and minimizing interference from other cores compared to the conventional MSI protocol with FCFS split-transaction bus as we detail in the latency analysis in Section 5.3.

Rule 4 PISCOT *supports OOO architectures by allowing cores to issue multiple outstanding requests. However, to limit coherence interference, it only services at most one request from any given core at a time.*

5.2.1 Illustrative Example

To better explain the operation of PISCOT, we use the same example from Section 2.4.1 for a system with three cores: $C0$ — $C2$ and delineates PISCOT's behavior in Figure 5.2. The example focuses on a single cache line A, which is assumed to be initially owned by $C1$. At timestamp $t - x$, a store request to A from $C0$ misses in its private cache (it was originally in `I` state). As a result, a GetM(A) message is placed in its cache controller's local buffer waiting for $C0$'s slot on the request bus. The line state changes in the private cache from `I` to $\texttt{IM}^{ad}$ waiting for its message to appear on the bus. The same situation occurs for $C2$ at timestamp t. At $t + 4$, $C0$ is granted a slot by the TDM arbiter and its request is issued on the `Request Bus`. The coherence message is assumed to consume two cycles to be broadcasted. Accordingly, $C0$ observes its OwnGetM(A) on the bus and move to $\texttt{IM}^{d}$ while waiting for data. On the other hand, once $C1$ observes $C0$'s GetM(A) (OtherGetM(A)) and since $C1$ is the owner of A, it responds with placing the updated data in its local buffer to be written back to the shared memory (timestamp $t + 6$) and moves to `I` state. In addition, two actions are pushed into the `Service Queue` as a result of $C0$'s request. This is because $C1$ has to write back its updated A first to the shared memory and then the shared memory will send the data to $C0$; these are indicated in Figure 5.2 in the `Service Queue` as $C1$:WB(A) and $C0$:Rx(A), respectively.

Simultaneously at $t + 8$, a GetM(A) request from $C1$ arrives and is issued on the `Request Bus` immediately since it is $C1$'s slot. Similar to what happened during $C0$'s slot, $C1$ moves to the $\texttt{IM}^{d}$ state and two actions are pushed into the `Service Queue`: $C0$:WB(A) and $C1$:Rx(A). The reason for this is that $C0$ should obtain its requested data first, according to the FCFS schedule, conduct its store operation, and write back the updated data to the shared memory before $C1$ can proceed with its GetM(A) request. For the same reason, $C0$ moves to the `IM`^{d}`I` state. This is indicated at timestamp $t + 12$. Now, $C2$ is finally granted access to the `Request Bus` and issues its request. Similar events to those during $C1$'s slot occur with the difference that $C1$ is the owner responsible to write-back A before the shared memory sends it to $C2$ according to the FCFS order. For the `Response Bus`, it services requests in the `Service Queue` in order of their arrival as previously explained. Assuming that one data transfer requires 50 cycles, it finishes the data transfer of $C1$'s WB(A) to shared memory at $t + 58$. $C0$'s Rx(A) from shared memory at $t + 108$, performs its store operation and places the new data in its local buffer and moves to `I` state. $C0$'s WB(A) to shared memory finishes at $t + 158$. $C1$'s Rx(A) from shared memory at $t + 208$, performs its store operation and places the new data in its local buffer and moves to `I` state. $C1$'s WB(A) to shared memory finishes at $t + 258$, and finally $C2$ receives A from shared memory at $t + 308$.

Comparing this with the behavior of PMSI adopting the traditional TDM bus in Figure 2.8, it shows the clear advantage of PISCOT that reduces the total latency of the same sequence of memory requests by 792 cycles (from $t+1100$ to $t+308$). More detailed comparisons on the effect of both WCL as well as average performance are introduced in the evaluation Chapter 6.

5.2.2 Satisfying Coherence Predictability Invariants

Coherence protocols can generally lead to unpredictable behaviors if not carefully managed. In addition, previous works have shown that combining conventional coherence protocols with traditional predictable arbiters also breaks system's predictability [1]. Since we claim that PISCOT indeed achieves predictability by utilizing conventional coherence while deploying the proposed split-transaction predictable arbiter, we believe it is necessary to elaborate more on how PISCOT achieves this predictability. Authors of [1] introduced 6 invariants that they stated that they must be satisfied to ensure predictability in the existence of coherence. We now show how PISCOT, unlike PMSI [1], is satisfying those invariants without the need to modify the coherence protocol. This discussion also illustrates the novel operation of PISCOT compared to traditional predictable arbiters such as TDM when tasks can share data. For inclusiveness, we state each invariant and then prove how PISCOT satisfies it. We prove each case by contradiction starting with a hypothesis that PISCOT breaks such invariant and then show that this contradicts PISCOT's operation explained at the beginning of this section.

Invariant 1 *A predictable bus arbiter must manage coherence messages on the bus such that each core may issue a coherence request on the bus if and only if it is granted an access slot to the bus.*

Lemma 1 PISCOT *satisfies Invariant* 1.

Proof: The proof is trivial since allowing a core to send a request without being granted access by the arbiter contradicts with PISCOT's TDM arbiter at the `Request Bus`.

Invariant 2 *The shared memory services requests to the same line in the order of their arrival to the shared memory.*

Lemma 2 PISCOT *satisfies Invariants* 2.

Proof: Let Req_i and Req_j be two requests to the same cache line such that Req_i arrived to the shared memory first. Assume that the shared memory serviced Req_j before Req_i such that Invariant 2 is broken. (1)
Now considering PISCOT's operation, Req_i will arrive to the shared memory first only if it is broadcasted on the `Request Bus` first. Hence, Req_i arriving at the shared memory first indicates that it has been queued into the `Service Queue` ahead of Req_j. Now, according to the `Response Bus`'s FCFS, Req_i must be serviced before Req_j. (2)
(1) and (2) contradicts, which completes the proof.

Invariant 3 *A core responds to coherence requests in the order of their arrival to that core.*

Lemma 3 PISCOT *satisfies Invariant* 3.

Proof: Let $Req_i(A)$ and $Req_j(B)$ be two requests to cache lines A and B respectively that are owned by Core Ck such that Ck observes $Req_i(A)$ first. To break Invariant 3, PISCOT has to service $Req_j(B)$ before $Req_i(A)$. (1)
Now, according to PISCOT's operation, a core responds to a request for a cache line that it owns by placing the data immediately in its local buffer. Additionally, a WB action is queued into the `Service Queue` along with its initiating coherence message of the request itself during the same `Request Bus`'s TDM slot. For instance, at time

$t + 8$ in Figure 5.2, $C0$'s GetM(A) message resulted in pushing two actions to the `Service Queue`: 1) $C1$ has to write back A (WB(A)) first and only afterwards 2) $C2$ can receive its requested data (RX(A)) from shared memory. Since Ck observes $Req_i(A)$ first, it mandates under PISCOT that $Req_i(A)$ was issued in the `Request Bus` before $Req_j(B)$. Additionally, since requests are queued in the `Service Queue` based on their appearance timestamp on the `Request Bus`, it mandates that $Req_i(A)$ and its corresponding WB(A) are queued in the `Service Queue` ahead of $Req_i(B)$ and its WB(B). Finally, according to the `Response Bus`'s FCFS policy, $Req_i(A)$ will get its data before $Req_i(B)$. (2)
(1) and (2) contradicts, which completes the proof.

Invariant 4 *A write request from a core that is a hit to a non-modified line in its private cache has to wait for the arbiter to grant this core an access to the bus.*

Lemma 4 PISCOT *satisfies Invariant* 4.

Proof: Let $Req_i(A)$ be a write request from core Ck to line A that Ck has in the S state in its private cache. To break Invariant 4, PISCOT shall allow $Req_i(A)$ to hit in the private cache and execute the operation silently without waiting for any permission from the bus arbiter. (1)
According to PISCOT's coherence protocol inherited from conventional MSI (Figure 2.5), A store to a cache line in S state has to issue a getM() coherence message and wait in the SM^{ad} state. Afterwards, this message is only issued on the bus once its core is granted access according to the Request Bus's TDM schedule. (2)
(1) and (2) contradicts, which completes the proof.

Invariant 5 *A write request from a core that is a hit to a non-modified line,* A, *in its private cache has to wait until all waiting cores that previously requested* A *get an access to* A.

Lemma 5 PISCOT *satisfies Invariant* 5.

Proof: Let cache line A to be initially in the S state in core Cj's private cache. Let also $Req_i(A)$ be a read request from core Ci to cache line A that is broadcasted on the bus at time $t1$. Then, assume that Cj at time $t1 + \delta$ (where $\delta > 0$) has a store request $Req_j(A)$ to A. To break Invariant 5, assume that $Req_j(A)$ is serviced before $Req_i(A)$. (1)
However, from Lemma 4, it follows that $Req_j(A)$ has to wait for Cj's TDM slot on the `Request Bus` to broadcast a $GetM(A)$ message on the bus before it can proceed with its store operation. Assume that this happens at time $t2$. Since $Req_j(A)$ arrived at $t + \delta$, it follows that $t2 \geq t1 + \delta$. As a result and from Lemma 2, $Req_i(A)$ request is serviced before $Req_j(A)$ since $t2 > t1$. (2)
(1) and (2) contradicts, which completes the proof.

Invariant 6 *Each core has to deploy a predictable arbitration between its own generated requests and its responses to requests from other cores.*

Lemma 6 PISCOT *satisfies Invariant* 6.

Proof: Assume a system with N cores C0 to CN such that one core Ci, $0 \leq i \leq N$, has a request to service from the memory, say Req_i, while all the other $N-1$ cores keep issuing requests to cache lines that are modified (owned) by Ci. To break Invariant 6, Ci keeps servicing these requests and is not granted a guaranteed time at all where it can finish its Req_i request. (1)

Now, we discuss how PISCOT schedules these requests. First, each core can only issue requests during its dedicated TDM slot (Lemma 1). Second, an owner core responds to requests from another core immediately during this other core slot and not its own slot (Lemma 3). Accordingly, for our dictated scenario, Req_i has a guaranteed time slot to be issued on the `Request Bus`. Finally, since the `Response Bus` services requests in their order on the `Service Queue`, Req_i is guaranteed to finish its data transfer once all requests in front of it in the `Service Queue` finish their transfers. Now, it remains to show that the number of these requests is bounded. According to the operation described at the beginning of this section, PISCOT only allows a maximum of one request from any core at any time in the `Service Queue`. As a result, Req_i cannot have more than $N-1$ requests ahead of it `Service Queue`, which guarantees it a bound on the time it can be serviced (Section 5.3 provides a detailed latency analysis to derive these bounds). (2)

For now, (2) clearly contradicts (1), which completes the proof.

5.3 Analytical Worst-Case Latency

We derive the WCL suffered by any single request to the cache hierarchy that is managed by PISCOT. In doing so, we will use Figure 5.2, where the GetM(A) from $C2$ is the request under analysis or *rua*. As previously explained, the system in Figure 5.2 has three cores. As the figure illustrates, upon the arrival of the *rua* at timestamp t, there is a pending request from $C0$ to the same cache line A, which is initially owned by $C1$ in the M state. Generally, from its arrival to the private cache controller buffer until it completely receives the requested data, a request suffers from three different latency components. Namely, it suffers from latency due to arbitration on the request bus, denoted as $ReqBusL$, latency due to arbitration on the response bus, denoted as $ResBusL$, and finally the latency needed to transfer its data from the memory denoted as $AccL$. The $AccL$ depends on the time required to access the shared memory and transfer one cache line to the requesting core's private cache. Now, we derive the worst-case latency of each of the other two components.

Lemma 7 ***Worst-Case Request-Bus Latency*** **(**$ReqBusL^{WC}$**).** *For a system with N cores, a request has to wait for a maximum of $ReqBusL^{WC}$ cycles as calculated in Equation 5.3.1 before it is granted access to the request bus, where S^{Req} is the TDM slot width of the request bus in cycles.*

$$ReqBusL^{WC} = N \cdot S^{Req} \tag{5.3.1}$$

Proof: Recall that the request bus is managed using a TDM arbiter. In the worst case, the *rua* arrives such that its core has just missed its own slot. Since we have N cores and each core is allocated one TDM slot of width S^{Req}, the *rua* has to wait for $N \cdot S^{Req}$ cycles before its corresponding core gets another slot. In Figure 5.2, $S^{Req} = 4$ cycles and $N = 3$; thus, the GetM(A) from $C2$ waits until $t + 12$ to gain access to the bus.

Lemma 8 ***Worst-Case Response-Bus Latency*** **(**$ResBusL^{WC}$**).** *For a system with N cores, a request has to wait for a maximum of $ResBusL^{WC}$ cycles from its arrival time to the* ***Service Queue*** *before it can start receiving its requested data.*

$ResBusL^{WC}$ is calculated by Equation 5.3.2, where S^{Res} is the time required to conduct one memory transfer on the response bus.

$$ResBusL^{WC} = (2 \cdot N - 1) \cdot S^{Res} \tag{5.3.2}$$

Proof: Recall that the `Response Bus` services requests that arrive to the `Service Queue` from the `Request Bus` in a FCFS fashion. In addition, PISCOT allows each core to have at most one request in the `Service Queue` at any given time. Accordingly, the *rua* waits in worst-case for a request from every other core to get serviced. Moreover, in worst-case, each request can require two memory transfers. This is because each request can be modified by another core and hence requires a write-back before the shared memory can send the updated data to the requesting core. Since we have $N - 1$ other cores, this consumes a total of $(N - 1) \cdot 2 \cdot S^{Res}$. Finally, the *rua* itself in worst-case requires a write-back before it can start transferring its own data, which consumes an additional S^{Res}. This leads to $ResBusL^{WC} = (N - 1) \cdot 2 \cdot S^{Res} + S^{Res}$ or $(2 \cdot N - 1) \cdot S^{Res}$. In Figure 5.2, where $S^{Res} = 50$ cycles, the GetM(A) from $C2$ incurs a $ResBusL$ from $t + 8$ to $t + 258$, which is 250 cycles.

Lemma 9 ***Total Request Worst-Case Latency ($TotL^{WC}$).*** *For a system with N cores, the maximum total latency that a request can encounter from its arrival time to its private cache controller before it can start receiving its requested data can be calculated as:*

$$TotL^{WC} = N \cdot (S^{Req} + 2S^{Res}) \tag{5.3.3}$$

Proof: Since $TotL^{WC} = ReqBusL^{WC} + RespBusL^{WC} + accL$, the proof directly follows from Lemmas 7 and 8, and the fact that $accL = S^{Res}$ per definition.

5.3.1 Direct Cache-to-Cache Communication

In this case, only one response slot is needed for any request as Lemma 10 proves. Therefore, the total request WCL for such architecture reduces to the value in Lemma 11.

Lemma 10 ***Worst-Case Response-Bus Latency with Cache-to-Cache Support*** **(*$ResBusL^{WC}_{C2C}$*).** *For a system with N cores that supports direct communication among cores' private caches, the maximum latency a request can suffer from its arrival time to the global response queue before it can start receiving its requested data can be calculated as in Equation 5.3.4, where S^{Res} is the time required to conduct a memory transfer on the response bus.*

$$ResBusL^{WC}_{C2C} = (N-1) \cdot S^{Res} \tag{5.3.4}$$

Proof: The proof directly follows from the proof of Lemma 8, with the exception that only one response slot is required per core instead of two as follows. For any request, there are three possibilities. 1) A core requests to read from or write to a cache line that is up-to-date at the shared memory. In this case, the shared memory transfers this line to the requesting core. 2) A core requests to write to a line that is modified by another core. Thus, the owner core has to send this line to the requesting core. Since the latter is going to update the line, the shared memory does not need to receive the line at the moment. 3) A core requests to read from a line that is modified by another core. In this case, the owner has to send this line to both the requesting core and the shared memory. However, since the architecture supports cache-to-cache communication, the data can be sent to both at the same slot. This proves that under all these possibilities, only one response slot is needed instead of two compared to Lemma 8. In conclusion, the $ResBusL^{WC}_{C2C} = (N-1) \cdot S^{Res}$.

Lemma 11 ***Total Request Worst-Case Latency with Cache-to-Cache Support ($TotL^{WC}_{C2C}$).*** *For a system with N cores that supports direct communication among cores' private caches, the maximum total latency that a request can encounter from its arrival time to its private cache controller before it can start receiving its requested data can be calculated as:*

$$TotL^{WC} = N \cdot (S^{Req} + S^{Res}) \tag{5.3.5}$$

Proof: The proof directly follows from summing the latency components in Lemmas 7 and 10, and the $AccL$.

5.3.2 Total Task's Worst-Case Memory Latency

The latencies derived so far are concerned with a single memory request. However, to derive the total task's WCET, the total memory latency, $WCML$, has to be obtained and then added to the worst-case computation time, $WCCT$, such that:

$$WCET = WCCT + WCML \tag{5.3.6}$$

Let WCL_{Req} to be the per-request WCL to differentiate it from the total $WCML$, where WCL_{Req} is either the $TotL^{WC}$ in Lemma 9 if no cache-to-cache is supported, or the $TotL^{WC}_{C2C}$ in Lemma 11 otherwise. We now show different approaches to utilize this WCL_{Req} to derive $WCML$.

5.3.2.1 Using total number of requests

The first approach directly obtains $WCML$ through Equation 5.3.7, where $NReq$ is the worst-case total number of issued memory requests by the task. $NReq$ can be obtained by statically analyzing the task in isolation [39].

$$WCML = NReq \times WCL_{Req} \tag{5.3.7}$$

5.3.2.2 Distinction between private and shared data

Although the bound provided in Equation 5.3.7 is safe, it is rather pessimistic. This is because it assumes that all requests are misses, while in reality some of the requests will hit in the private caches and thus suffer a much less latency than WCL_{Req}. One challenge in data-sharing systems is that whether a task access to shared data hits or misses in the private cache depends on the access pattern of competing tasks, entailing that no reasoning can be made about whether this access hits or misses in the private cache by statically analyzing the task in isolation. Even worse, since shared cache lines can conflict with private lines in the core's private cache and hence evict each other, no analysis can be applied to access to private data as well. In this case, Equation 5.3.7 applies. In contrast, if private and shared data are isolated from each other; for instance, by mapping them to different cache sets, tighter memory latency bounds can be obtained for requests to the private data. Assuming this isolation, a task's hit ratio to the private data obtained from analyzing the task in isolation still holds upon interference from co-running other tasks. As a result, in such system, we can obtain the $WCML$ as in Equation 5.3.8, where $NReq^{priv}$ is the number of requests to private data, among them $NReq^{priv}_{hit}$ are hits in the private cache, and

$NReq^{priv}_{miss}$ are misses. L_{hit} is the hit latency of the private cache and $NReq^{shrd}$ is the number of requests to shared data. Since $L_{hit} << WCL_{req}$ (L_{hit} is one or two cycles in modern architectures), the $WCML$ bound in Equation 5.3.8 is generally tighter than that of Equation 5.3.7. The actual values depend on the ratio of requests to private and shared data, and hence, is application dependent.

$$WCML = NReq^{priv}_{hit} \times L_{hit} + (NReq^{priv}_{miss} + NReq^{shrd}) \times WCL_{Req} \tag{5.3.8}$$

5.3.3 Replacement of Dirty Cache Lines

The analysis in Lemmas 7 — 11 assumes that when a request misses in the private cache, it is sent directly to the bus arbiter to fetch the requested data. However, it is possible that the requested cache line is mapped to an entry that already has a valid data of another cache line. This is called a cache conflict. In this case, the previous cache line is to be evicted from the private cache and the requested cache line is to be fetched to the same entry. If the evicted cache line has modified data, it has to be written first to the shared memory; otherwise, this data is going to be lost. This adds an extra latency of one memory transfer (or S^{Res}) for each miss request in the worst case. In other words, this adds $N \times S^{Res}$ to the latencies in Lemmas 9 and 11. However, assuming that every request is going to an eviction to a modified line is overly pessimistic and a tighter bound can be obtained as follows.

5.3.3.1 Total number of writes

Since the additional latency component is caused only upon evicting a dirty cache line, the total number of these replacements is bounded by the total number of write requests of the task, $WReq$. Accordingly, the effect of the replacement is better to be considered at the task level by updating Equation 5.3.7 to:

$$WCML = NReq \times WCL_{Req} + WReq \times S^{Res} \tag{5.3.9}$$

5.3.3.2 Distinction between private and shared data

Moreover, if the isolation between private and shared data discussed in Section 5.3.2.2 is adopted, the delay effects of replacement can be further reduced. This is because the number of replacements happening withing private data can also be obtained from analyzing the task using existing static analysis tools. Therefore, integrating the effect of replacements in Equation 5.3.8 leads to the $WCML$ in Equation 5.3.10, where $NRepl^{priv}$ is the worst-case number of dirty cache line replacements within private data, $WReq^{shrd}$ is the worst-case number of write requests to shared data.

$$\begin{aligned} WCML = {} & NReq^{priv}_{hit} \times L_{hit} + NRepl^{priv} \times S^{Res} \\ & + (NReq^{priv}_{miss} + NReq^{shrd}) \times WCL_{Req} + WReq^{shrd} \times S^{Res} \end{aligned} \tag{5.3.10}$$

Chapter 6

Evaluation and Validation

This Chapter provides details of the Verification and Validation (VnV) process used to verify CacheSim. The verification plan is closely tied to the simulator specification and contains a description of what features need to be exercised and the techniques to be used to verify our implementation.

6.1 General Information

The following sections provide more detail about the Verification and Validation plan of the CacheSim software that allows the user to test the correctness of the implemented code. The implementation verification process involves identifying test cases targeting a specific function of the design (i.e. directed test case) and describing a specific set of stimulus to apply to the design. Sometimes, the test case can be self-checking and looks for specific symptoms of failures (i.e. assertion-based tests) in the output stream observed from the design.

6.1.1 Objectives

The objective is to verify the CacheSim thoroughly against the design specification defined in Chapter 4 and make sure there are no functional bugs in addition to conducting detailed experiments to evaluate and explore the performance of different implemented algorithms. While doing this, there should be a way of measuring the completeness of verification. First, code coverage tools provide a first level measure on the verification completeness. The data collected during code coverage has no knowledge of the functionality of the design but provides information on the execution of the code line by line. By guaranteeing that every line of the CacheSim is executed at least once during simulations, a certain level of confidence can be achieved and code coverage tools can help achieve that. Second, directed and regression tests are used for functional coverage to ensure that stimulus vectors exercise all features in the CacheSim. Finally, we will use validation metrics defined in Section 6.1.2 to verify the simulator results.

6.1.2 Properties of a Correct Solution

We will use the following metrics to validate the simulator results.

1. For PISCOT proposed solution, we compare the performance metrics generated by the simulator such as 1) request and response bus latencies. 2) latency penalty due to L1 cache replacement. 3) observed worst-case latency, average-case performance, and total execution time against the theoretical static analysis that we introduced in Section 5.3 and made sure that all latency components observed by the simulator are within their analytical bound.

2. For existing solutions, journal articles that explore the latency analysis is used as a reference. For instance, [1] and [8]. By conducting the same experiments in the evaluation section, CacheSim should produce the same results as reported in those experiments.

3. Cache Coherence Protocols need to pass the implemented finite state machine (FSM) assertions to make sure that all possible state transitions are occurred correctly under the correct input stimulus without violating any of the requirements specified in the state table of coherence protocols in Section 4.2.8.

6.1.3 Automated Testing and Verification Tools

CacheSim core modules are written in C++ using object-oriented programming concepts. The simulator integrates some of ns-3 [55] network simulator libraries such as ns-3 event scheduler layer that keeps track of all events that are scheduled to execute at a specified simulation time. However, it is important to note that while ns3 is event-based (which is more suitable for networking), our simulator is cycle-accurate. CacheSim relies on TinyXml [56] library to acquire and resolve simulation configuration parameters from the input XML document. Gcov [57] tool is used to perform code coverage. A detailed summary of the coverage analysis results can be found in Appendix A.3. Bash scripting framework is used for running unit tests in continuous integration and reporting results of the tests (e.g. number of tests failed, runtime duration, etc.). The whole process can be done through GitLab by defining the set of tests to run whenever a new revision is submitted into the project repository. Finally, Waf [58] tools is used for configuration, compiling, and build our simulator.

6.1.4 Benchmarks

There are three sets of benchmarks used in this thesis: 1) We craft 16 synthetic workloads to stress the behavior of the evaluated features. Some of the synthetic workload resemble the maximum data sharing among cores (all lines are shared) with different read/write ratio, while others explore different patterns of temporal and spatial locality of the access data. For more details about these benchmark, one can refer to Appendix A.1. 2) Besides, we use EEMBC benchmarks suite. it is an industry-standard benchmarks for the software used in autonomous driving, mobile imaging, the Internet of Things (IoT), and mobile devices application developed by [59] to verify CacheSim. 3) We also consider SPLASH-3 [60] benchmark suite of multi-threaded applications to run multi-core simulations with data sharing. Due to GitHub space limitation, we only uploaded the synthetics and EEMBC benchmark suites to CacheSim project repository [10]. However, the entire benchmarks suites can be downloaded from our research group's GitLab repository [61].

6.1.5 CacheSim Evaluation Setup

The system context used for evaluation is shown in Figure 6.1. The simulation environment consists of a multi-core system with configurable number of cores and cache organization. The default simulator parameters are chosen to emulate the behavior of quad-core system running at 2.5 GHz with out-of-order pipelines, 8 kB direct-mapped L1 per-core private cache, and a 4 MB 8-ways set-associative L2 shared cache across all cores. Both L1 and LLC have a cache line size of 64 bytes.

It is important to note that this configuration is only used as a baseline. We study the effect of each of these configuration parameters in details in the experiments discussed in Section 6.2. Therefore, to increase the readability of this section, the following configuration values in Table 6.1 should be used as the default configuration in all the subsequent tests unless they are explicitly overwritten.

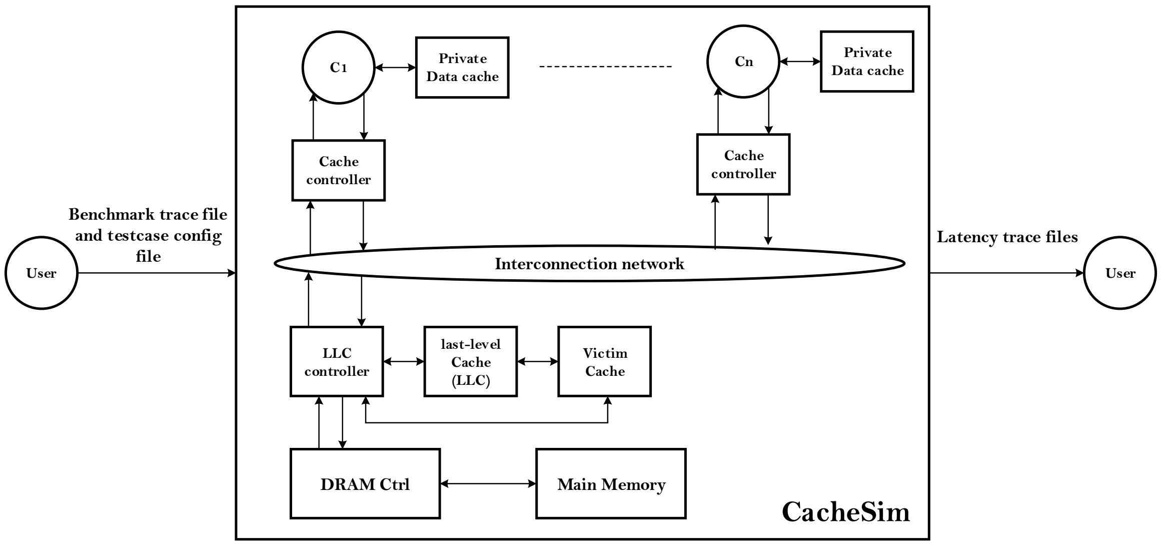


Figure 6.1: Baseline Topology

Table 6.1: Test case default configuration parameters

Parameter	**Value**
Simulation Step Size (Δ)	1 ns
T_{clk}	100 ns
N_{cores}	4
$Cache\ Block\ Size$	64 B
$L1\ CacheSize$	8 kB
$L1\ associativity$	1
$L1\ NSets$	128
$L2\ CacheSize$	4 MB
$L2\ associativity$	8

$L2\ Nsets$	8192
$L2\ interconnect\ Request\ Latency$	4 cycles
$L2\ interconnect\ Response\ Latency$	50 cycles
$Core\ Pending\ Reqs$	1
$Coherence\ Protocol$	MESI
$ReqBus_{Arb}$	FCFS
$RespBus_{Arb}$	FCFS
$Replc_{policy}$	LRU
$DRAM\ Latency$	200 cycles
$DRAM\ Outstanding\ Reqs$	16
$Perfect\ LLC\ Enable$	True
$LLC\ Cold\ Start\ Enable$	False
$C2C\ Transfer\ Enable$	True

6.2 Tests for Functional Requirements

This section describes the test cases that will be used to ensure the correctness of the simulator features. This section is divided into different testing scopes. These are:

- Cache Coherence Protocols testing.
- Interconnection Network testing.
- Cache Replacement Policies testing.
- Victim Cache and Fixed DRAM Latency testing.

6.2.1 Cache Coherence Protocols

This testing suite verifies the Cache Coherence Protocols supported by the simulator for both C2C and No-C2C architectures. Figure 6.2 depicts the execution time for the synthetic benchmarks. From this result, one can distinguish different groups of efficiency among the coherence protocols: MESI and MOESI are the best candidates for implementation when the running applications explore spatial and temporal locality of the access data as shown in the first four benchmarks. For instance, in read-modify-write benchmarks (RWStrideNoIntrf and RWStride32Rand), MESI and MOESI are 48% and 42% faster than MSI protocol. On the other hand, MSI shows higher performance at 50% and 63% for R75andL1Miss and R85RandL1Miss benchmarks compared to MESI and MOESI.

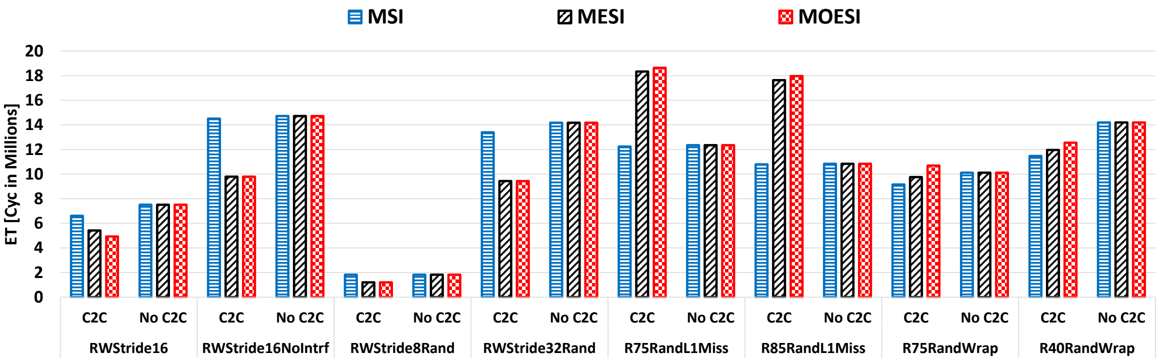


Figure 6.2: Execution Time for the synthetic workloads (FCFS, IO)

At first glance, it could be surprising that MSI outperforms MESI and MOESI. It is simply due to the absence of access locality in these benchmarks and the read percentage is higher than the write one. Therefore, in case of MESI and MOESI, a core cannot invalidate a cache line in "E" or "O" silently without write-back the data to the L2 cache when another core request to modify it. To further investigate the behaviour of MESI and MOESI protocols compared to the conventional MSI

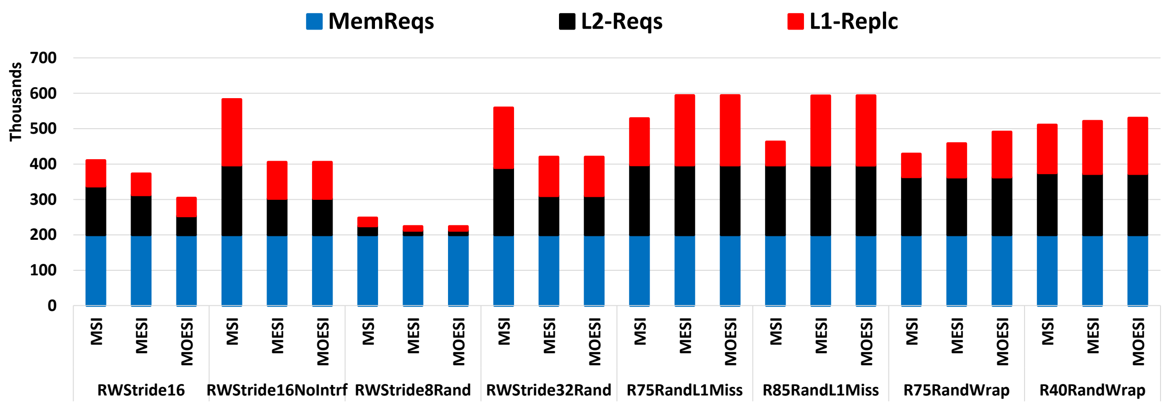


Figure 6.3: Breakdown of Synthetic workloads memory requests (FCFS, IO)

protocol when the application has no data locality, Figure 6.3 plots the observed L1 replacement count and the total number of memory requests to L2 for the synthetic benchmarks. As the figure illustrates, MESI and MOESI show huge replacement numbers for R75RandL1Miss and R85RandL1Miss compared to conventional MSI protocol. This confirms the same observation that we made in Figure 6.2. To further study the performance behaviour of the coherence protocols, we show the average-case memory latency for the synthetic benchmarks in Figure 6.4. Figure 6.4 confirms the same behaviour observed in the execution time in Figure 6.2.

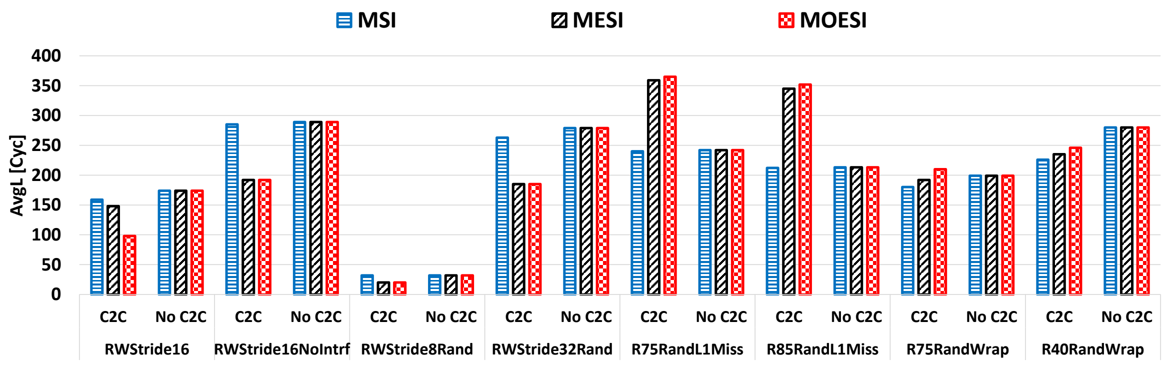


Figure 6.4: Average Latency of the Synthetic workloads (FCFS, IO)

We also tested the cache coherence protocols for EEMBC benchmarks. Figures

6.5 and 6.6 depicts the execution time and average-case memory latency for EEMBC benchmarks, respectively. Across all benchmarks, C2C architecture achieves up to 25% on average better performance than No-C2C architecture. The Intuition behind such behaviour is that for No-C2C architecture, the data always needs to come from the shared memory to the requested core which increases the congestion at the interconnect network.

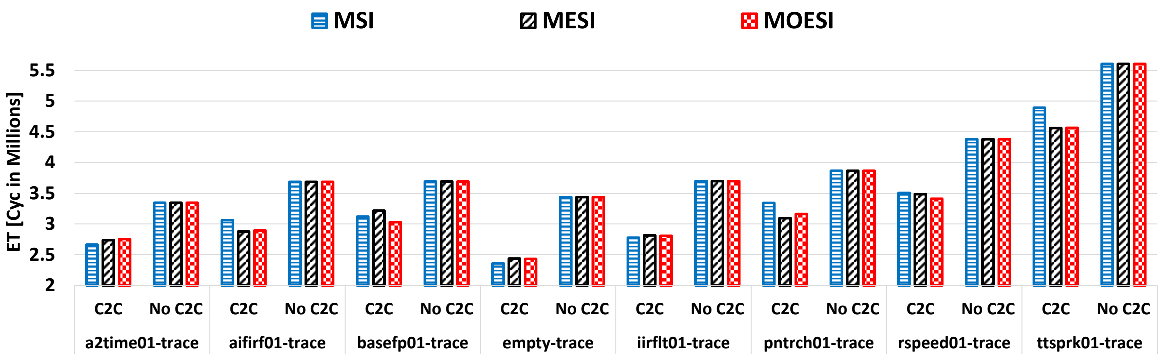


Figure 6.5: Execution Time for the EEMBC workloads (FCFS, IO)

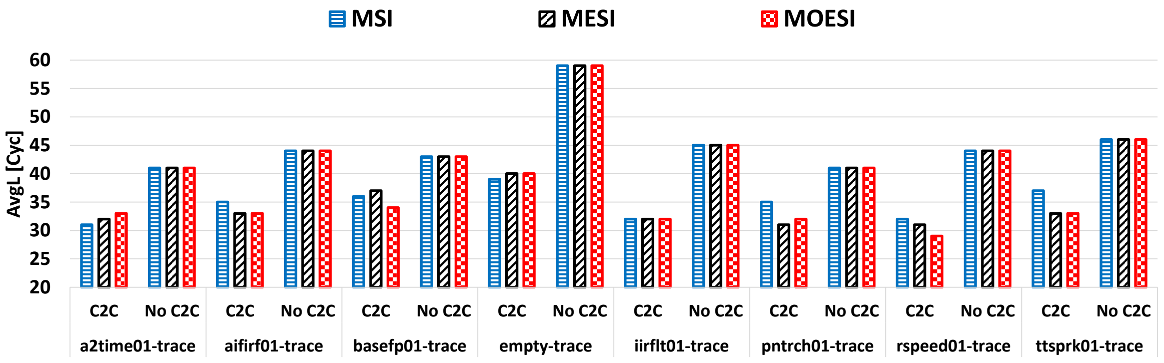


Figure 6.6: Average Latency of the EEMBC workloads (FCFS, IO)

6.2.2 Interconnection Network

In this set of experiments, we verify the three arbitration polices implemented at the interconnect network. PISCOT bus arbitration introduced in Chapter 5, conventional

MSI bus arbitration (Subsection 2.4.2), and PMSI bus arbitration (Subsection 2.4.4).

6.2.2.1 PISCOT Bus Arbitration

As explained in Chapter 5, the request and the response buses are split and operate independently in PISCOT. The former uses work-conserving TDM arbitration amongst cores while the latter services the responses in FCFS fashion. In order to verify PISCOT, we run multiple experiments with different numbers of pending requests (Npend) a core can issue to cover both in-order and out-of-order execution modes. In addition, we test PISCOT for C2C and No-C2C architectures. Figures 6.7 depicts the WCL for any request to the cache hierarchy for both EEMBC benchmarks. The figure shows both the analytical WCL bounds (T bars) and the observed (experimental) WCL (colored solid bars). From this figure, we make the following observations. 1) All observed WC latencies under PISCOT operation are lower than its corresponding analytical bounds derived in Section 5.3, which confirms the predictability of PISCOT. 2) WCL of PISCOT is insensitive to the variation of the number pending request a core can issue. The intuition behind such behaviour is that PISCOT allows cores to issue multiple outstanding requests. However, it only services at most one request from any given core at a time. The rationale for this is to limit the coherence interference among cores such that a request from any core can suffer interference due to a maximum of only one request from each other core.

Figure 6.8 depicts the overall execution time for EEMBC benchmarks. From this experiment, we make the following observation. Although PISCOT limits the number of requests issued on the bus from a given core to one request at a time, increasing the number of pending requests still improves performance (reduce execution time).

Recall that PISCOT doesn't limit the number of outstanding requests a core can issue to its private cache controller. Therefore, it still allows for private cache hits while waiting for data response for the missed request to come from L2 cache.

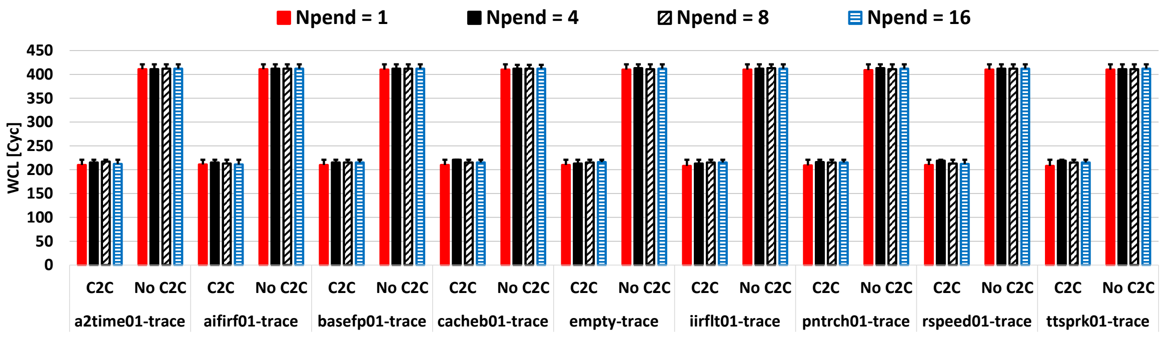


Figure 6.7: PISCOT Per-request worst-case latency for EEMBC suite.

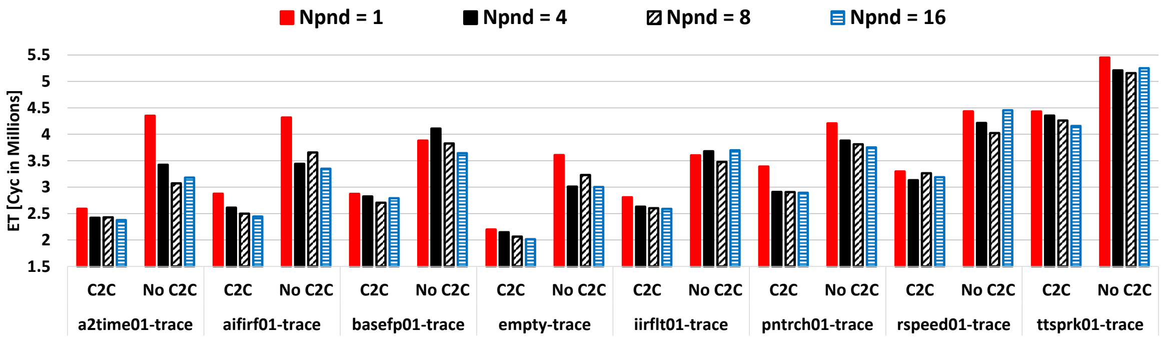


Figure 6.8: PISCOT execution time for EEMBC suite.

Figure 6.9 shows both the analytical bound for the total WCL derived by Equation 5.3.7 (T bars) and the observed total latencies (colored solid bars). In this case study we are interested in calculating the total memory WCL suffered by the total number of memory requests generated by a core during a period of time t. Furthermore, the observed one is decomposed to its sub-components: a) the request bus arbitration latency, b) the response bus memory transfer latency, c) the hit latency in the core's private cache, and d) the write-backs latency due to replacement. From

Figure 6.9, we conclude the following observations. 1) The response bus latency component dominates the total WCL for all applications. For instance, the total observed response latency reach up to 8× (barnes and volrend) and 4.3× on average larger than the replacement latency. This emphasises the conclusion we made in Section 5.3.3 that the effect of the eviction delays should be considered at the task-level and not the request-level. 2) Since SPLASH-3 applications exhibit a reduced ratio of writes compared to reads, they do not stress the difference between No-C2C and C2C architecture in the observed response bus latency. Therefore, to further show this effect, we execute synthetic experiments using the synthetic benchmarks that are used to generate WCL in Figure 6.7 except that we change the percentage of the CPU memory write request to 50% of total memory requests. The results show that with C2C communication, PISCOT achieves up to 1.74× (1.56× on average) higher bandwidth compared to No-C2C scheme.

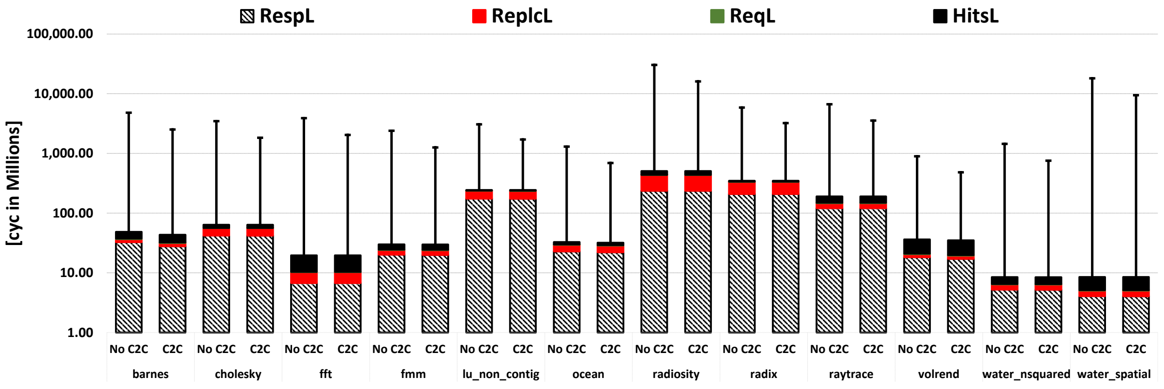


Figure 6.9: PISCOT Total observed and analytical memory latency for Splash-3 benchmarks. Values in y-axis are in log scale.

6.2.2.2 Conventional MSI Bus Arbitration

Conventional MSI is a commodity performance-oriented arbitration policy that is used in COTS platforms that favour system performance over other metrics such as fairness and predictability. Such arbiter prioritizes requests based on their arrival time. In Figures 6.10 and 6.11, we compare Conventional MSI performance against PISCOT for EEMBC workloads. Figures 6.10 clearly illustrate the benefits of conventional MSI. Conventional MSI outperforms PISCOT for all benchmarks: it achieves up to 13% on average better performance than PISCOT for No C2C architecture. Figure 6.11 confirms the same behaviour observed in the execution time in Figures 6.10. For some benchmarks; namely a2time01-trace and aifirf01-trace, MSI achieves up to 37% and 24% less latency than PISCOT.

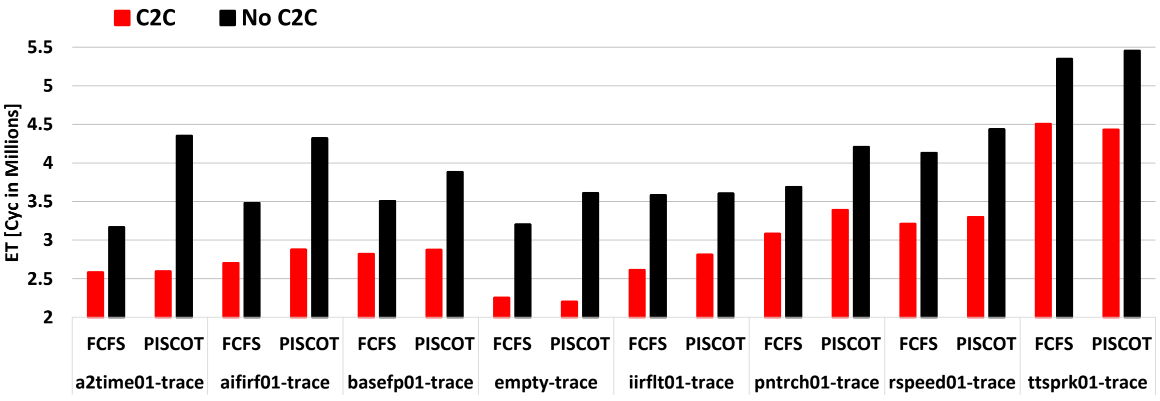


Figure 6.10: PISCOT Execution time comparison to conventional MSI protocol with FCFS split-transaction bus.

On the other hand, conventional MSI is not predictable since it provides no latency guarantees upon accessing the shared memory. This is because one core can have a request that is pending (theoretically) forever, while other cores are saturating the queues. Figures 6.12 and 6.13 highlight this behaviour. As shown from this result, MSI's WCL changes considerably when the number of pending requests a core can

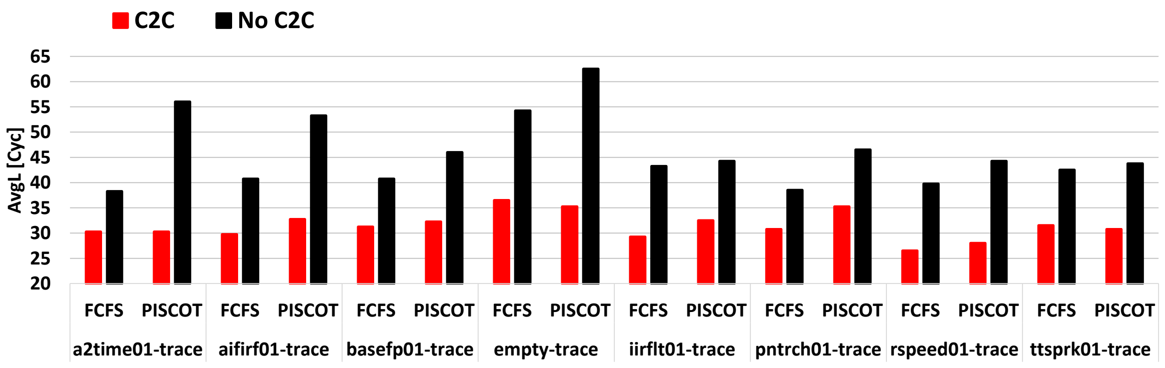


Figure 6.11: PISCOT Average Latency comparison to conventional MSI protocol with FCFS split-transaction bus.

issue increase. For instance, for a2time01-trace benchmark, MSI's WCL increased by 2× (for No C2C setup) and 3× (for C2C setups) when the number of core's pending requests changes from 1 to 16. Contrarily, PISCOT achieves predictable latency bound across all benchmarks. It achieves up to 5× on average tighter bound compared to conventional MSI.

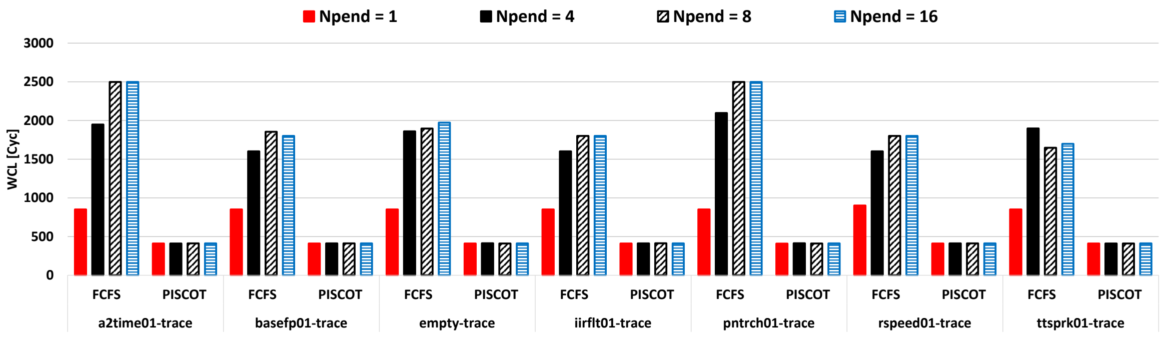


Figure 6.12: PISCOT versus FCFS split-transaction bus WCL with different number of pending core requests (MSI, No-C2C, L2-Lat = 50 Cycles).

In Figure 6.14, we study the impact on WCL upon increasing the L2 cache data response latency for both conventional MSI and PISCOT. In this case study, we fix the number of pending requests a core can issue to 1 and sweep the L2 cache response latency from 25 to 100 cycles. Each running core also executes the EEMBC workload

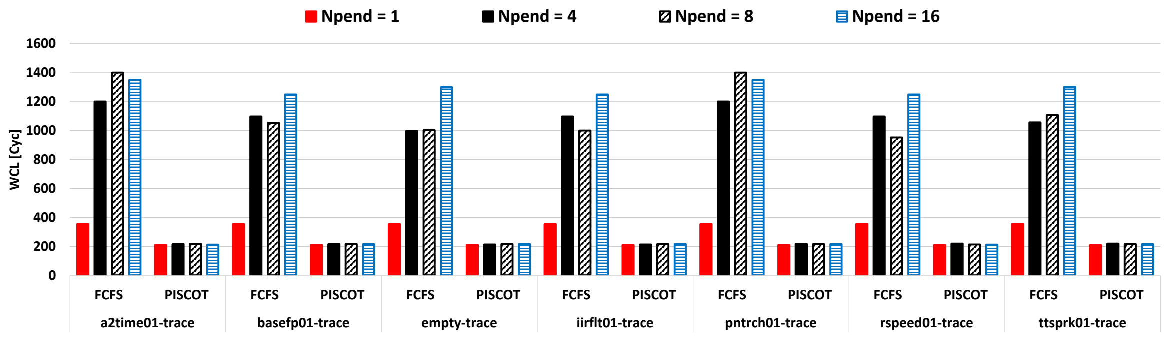


Figure 6.13: PISCOT versus FCFS split-transaction bus WCL with different number of pending core requests (MSI, C2C, L2-Lat = 50 Cycles).

with No C2C architecture setup. **Observations.** 1) Clearly, increasing the L2 cache latency increase the WCL for both MSI and PISCOT with the same ratio. For instance, increasing L2 cache latency from 25 to 100 cycles results in 3$\times$ increase in WCL on average in both conventional MSI and PISCOT. 2) The latency bound in PISCOT is much tighter than conventional MSI. PISCOT achieves up to 5$\times$ on average tighter bound compared to conventional MSI in all experiments. Figure 6.15 shows the impact of increasing the L2 cache response latency on the overall execution time of the running application. Again, the result is consistent with the observation that we made in Figure 6.14.

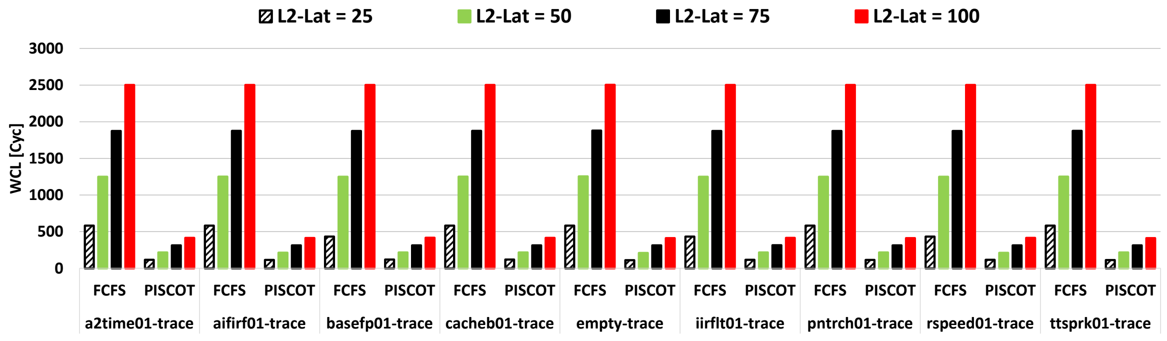


Figure 6.14: WCL of PISCOT versus FCFS split-transaction bus with different number of L2 cache response latency. (MSI, No-C2C, Npend = 1).

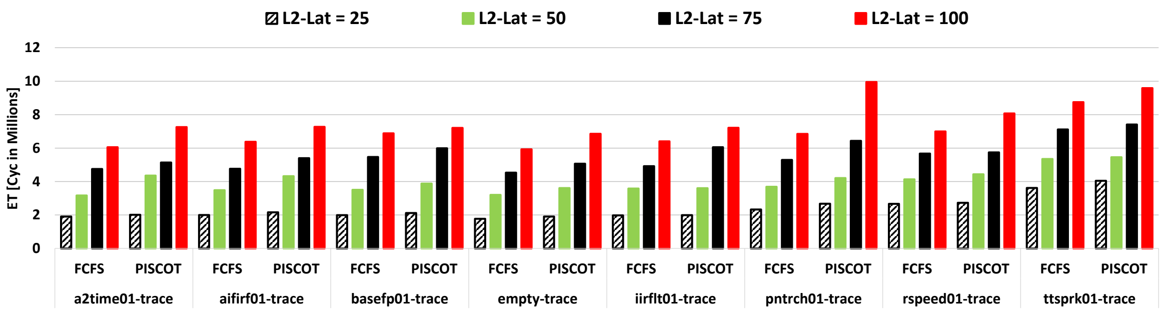


Figure 6.15: Execution time for PISCOT and FCFS split-transaction bus with different number of L2 cache response latency. (MSI, No-C2C, Npend = 1).

6.2.2.3 PMSI Bus Arbitration

Figures 6.16 and 6.17 depict the WCL for any request to the cache hierarchy for both SPLASH-3 benchmarks and the EEMBC workloads, respectively. The figures show both the analytical WCL bounds (T bars) and the observed (experimental) WCL (colored solid bars). We compare the WCL of the two PMSI schemes (where PMSI-WrkConsv is the one using work conserving TDM, while PMSI-NonWrkConsv is the one using non work conserving TDM with PISCOT and conventional MSI (FCFS) approaches. From this experiment, we make the following observations. 1) For PISCOT and PMSI, all the observed WC latencies are always within the analytical worst-case latency bounds. 2) PISCOT shows up to 4.9× improvement in the analytical WCL compared to PMSI. The analytical WCL of PMSI is 2050 cycles compared to 416 cycles in PISCOT. 3) Compared to PMSI, the observed WCLs in PISCOT achieve up to 2.74× tighter bounds on average across benchmarks. 4) PMSI incurs a large gap between experimental and analytical WCLs. In the SPLASH-3 benchmarks (Figure 6.16), this gap ranges from 70% (barnes and ocean) and reaches up to 3.4× (cholesky and radix). This is because PMSI's analytical WCL assumes a pathological worst-case scenario that is hard to construct in real applications. Even with EEMBC experiments

(Figure 6.17), the gap is more than 45% for most benchmarks. On the other hand, PISCOT achieves a tighter bound for the derived WCL. PISCOT achieves this tightness by enforcing FCFS arbitration policy on the response bus.

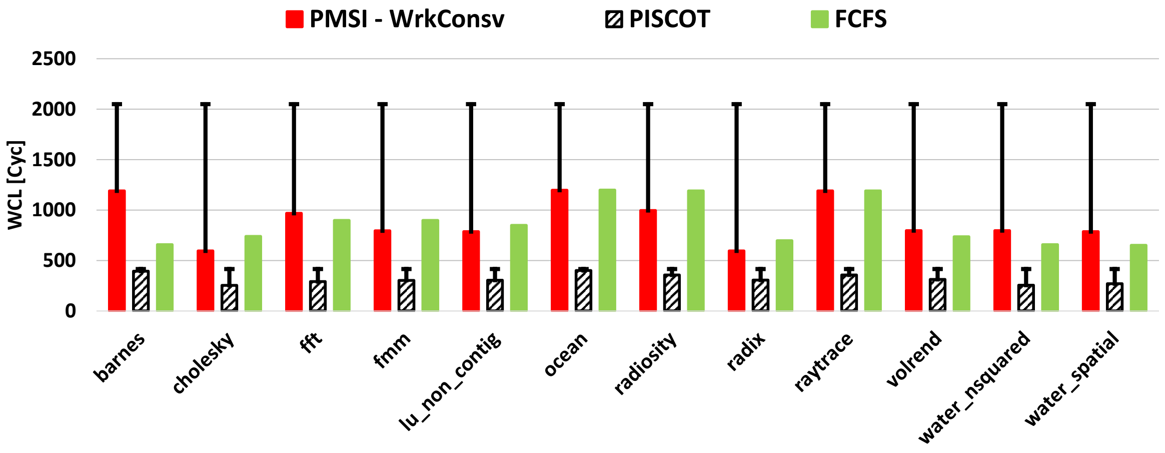


Figure 6.16: Per-request worst-case latency for SPLASH-3 suite.

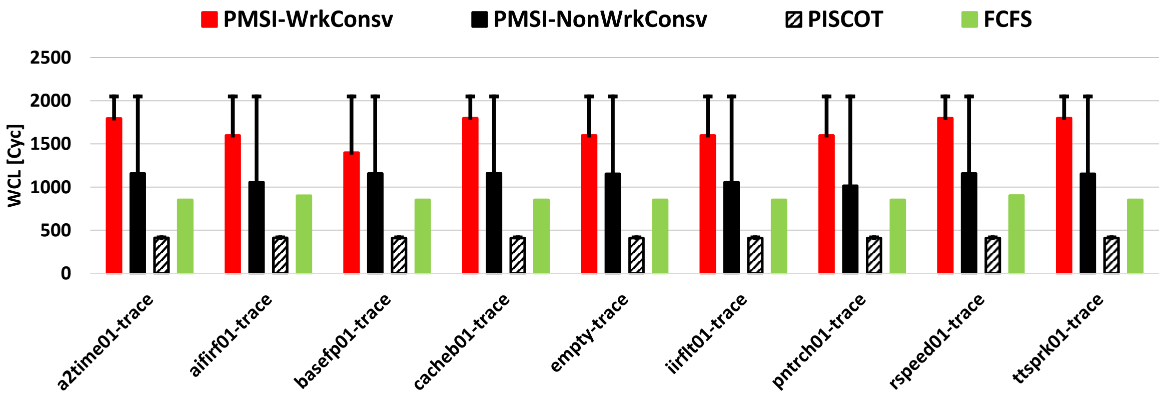


Figure 6.17: Per-request worst-case latency for the EEMBC suite.

Figure 6.18 shows the slowdown of PISCOT and PMSI-WrkConsv compared to the conventional MSI with split-transaction FCFS bus.

PMSI's slowdown is 2× on average (and up to 4.3×) across all benchmarks. This is due to the coupling of coherence and data transfer on the same TDM bus as explained in Section 5.2 in addition to the enforced protocol changes. Authors of [1] compared

PMSI with an MSI+conventional TDM arbiter, for which they reported that PMSI showed only a 45% slowdown. Recall here we consider MSI+split-transaction bus. These results combined emphasise our observation that the split-bus architecture can significantly increase performance compared to the traditionally considered bus architectures by the real-time community. On the other hand, Figure 6.18 shows that PISCOT achieves comparable results with slowdown in the range of 1%–4%. This is clearly a negligible cost for achieving timing predictability with tight latency bounds.

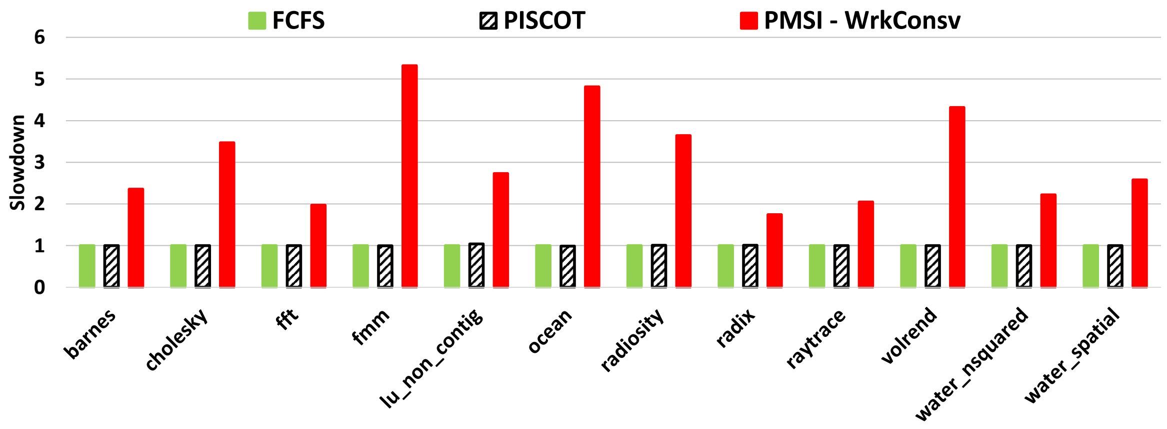


Figure 6.18: Execution time slowdown compared to conventional MSI protocol with split-transaction bus.

6.2.3 Cache Replacement Policies

This set of experiments studies and validates the different replacement algorithms that are implemented at L1 and L2 caches. CacheSim supports different replacement policies: FIFO, LIFO, LRU, MRU, LFU, MFU, and RAND. These policies are explained in details in Section 2.2.3. We observe that SPLASH-3 and EEMBC benchmarks lead to hit ratio over 99% at L2 cache. For this reason, we craft our own synthetic benchmarks in order to obtain larger cache miss ratio at L2 cache and therefore, show the relative performance of the different replacement algorithms. Please refer to Appendix A.1 for more details about these benchmarks. The results of these synthetic benchmarks' simulations are given in Figures 6.19 and 6.21, both experiments have the same configuration except that Figure 6.19 deploys PISCOT arbitration at the interconnect network while Figure 6.20 deploys conventional MSI (FCFS) arbitration. From these results, one can distinguish different groups of efficiency among the replacement policies: LRU and LFU, MRU and MFU, FIFO and LIFO, and RAND. LRU and LFU group appear as the best candidates for implementation. Across all benchmarks, LRU and LFU achieve the lowest miss rate up 9% on average compared to (27% MFU, 32% MRU). Also FIFO and LIFO achieve comparable performance, around 11% miss rate on average across all benchmarks. RAND replacement suffers only 3% performance loss compared to LRU on average.

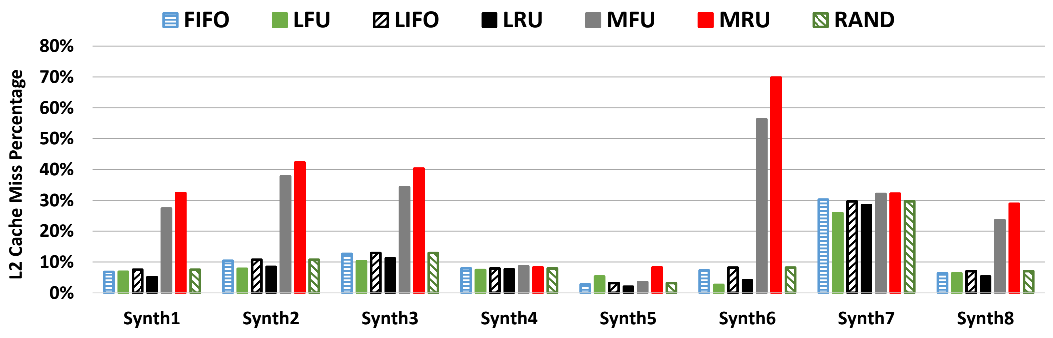


Figure 6.19: L2 Cache Miss Percentage for the synthetic benchmarks (MESI, PISCOT).

Figures 6.21 and 6.22 depict the effect of different replacement policies on the total execution time for PISCOT and conventional MSI arbiters, respectively. From these results we make the following observations. 1) As expected, conventional MSI outperform PISCOT in all benchmarks, this confirm the results we got in Figures 6.10 and 6.15. 2) Again, LRU and LFU group outperform all other replacement policies.

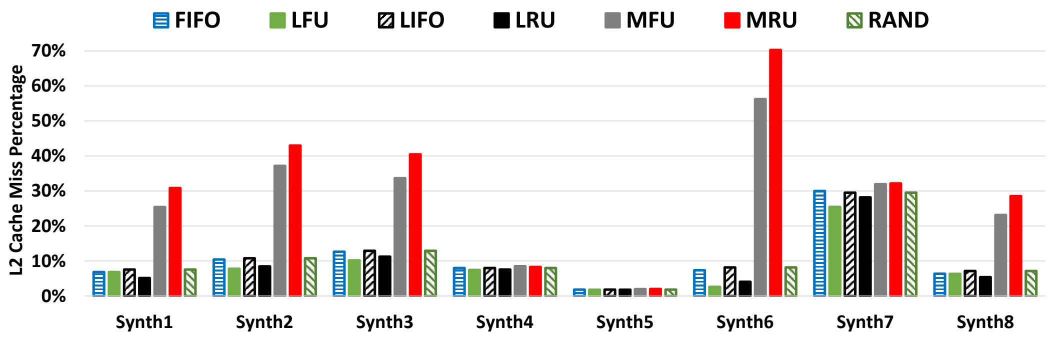


Figure 6.20: L2 Cache Miss Percentage for the synthetic benchmarks (MESI, FCFS).

This confirms the same observation we made in Figure 6.19 and 6.20.

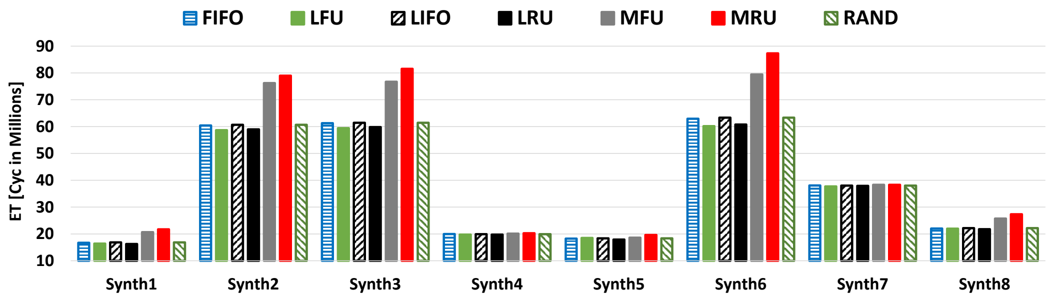


Figure 6.21: Execution Time for the synthetic benchmarks (MESI, PISCOT).

6.2.4 Victim Cache and Fixed DRAM Latency

This set of experiments studies verifies the Victim Cache and Fixed DRAM Latency module of CacheSim. The goal is to add a third hierarchy level to the multicore simulation environment (i.e. DRAM memory) and study the performance in the present of L2 cache replacements. In these experiments, we explore the performance impact of different access latency cycles for L2 cache and DRAM module, and different configurations for the number of pending core requests. Section 6.2.4.1 evaluates configuration with perfect L2 cache (with no replacements occurs at L2 cache), while Section 6.2.4.2 evaluates the performance of the synthetic benchmarks in the presence

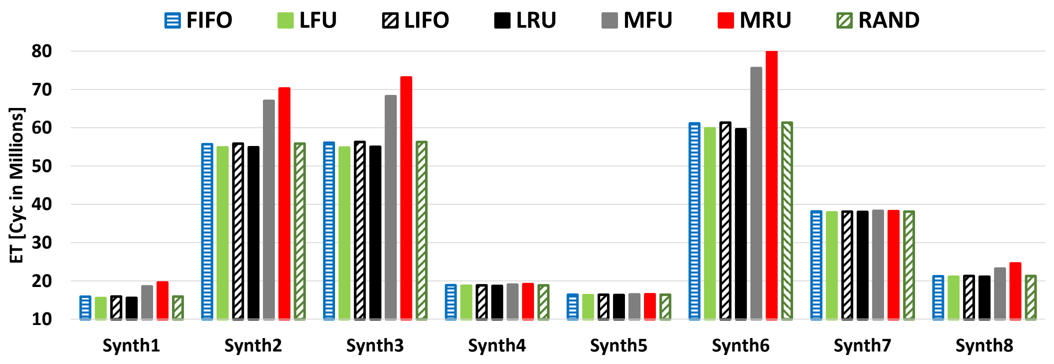


Figure 6.22: Execution Time for the synthetic benchmarks (MESI, FCFS).

of DRAM and L2 cache replacements.

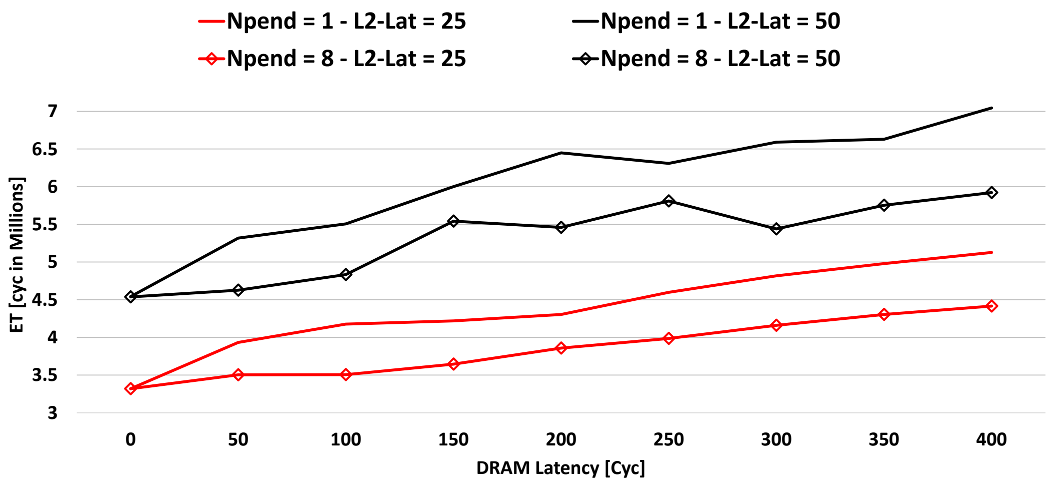


Figure 6.23: Standalone DRAM Latency impact on the Execution Time.

6.2.4.1 Standalone DRAM Latency impact

Methodology. We study the impact of increasing the DRAM latency on the performance of the running application. In this experiment, we want to show the standalone effect of increasing the DRAM latency. Therefore, we consider the EEMBC workloads as their L2's hit rate equal 100% in order to isolate the contribution of cache replacement. Then, we initialize L2 cache in the cold start mode (i.e. initially, all cache lines are invalids in L2 cache) and sweep the DRAM latency from 0 to 400 cycles. Figure 6.23 shows the overall execution time for different L2 cache response latency (L2-lat) for both in-order (Npend = 1) and out-of-order (Npend = 8) cores.

Observations. Clearly, increasing the DRAM latency, the increasing rate in the execution time is higher in in-order cores (Npend = 1) than that of the out-of-order ones (Npend = 8). Moreover, increasing the L2 cache latency degrades the performance linearly, this finding is consistent with the result we got in Figure 6.15. Figures 6.24 and 6.25 shows the DRAM latency effect on the execution time across all EEMBC benchmarks for both 25 cycles and 50 cycles L2 cache response latency configuration. The results resemble the same observations that we made in Figure 6.23 above.

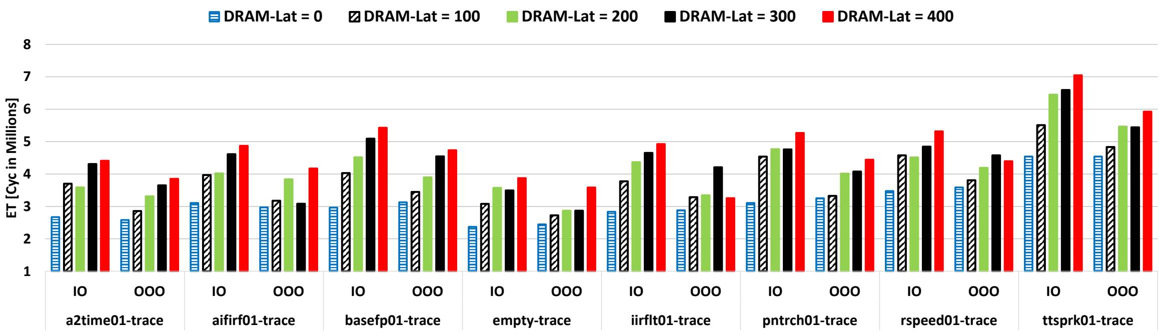


Figure 6.24: Standalone DRAM Latency impact on the Execution Time for the EEMBC benchmark with L2-Lat = 50 cycles. (MESI, FCFS).

6.2.4.2 Impact of DRAM latency in the present of cache replacement

In this subsection, we uses the synthetic workloads to evaluate the multi-threaded workloads performance in the presents of DRAM and L2 cache replacement. In this experiment, we used LRU replacement policy and FCFS split bus arbiter. As per the results we obtained in Figure 6.20, LRU L2 miss rate varying between 9% to 29% across these benchmarks.

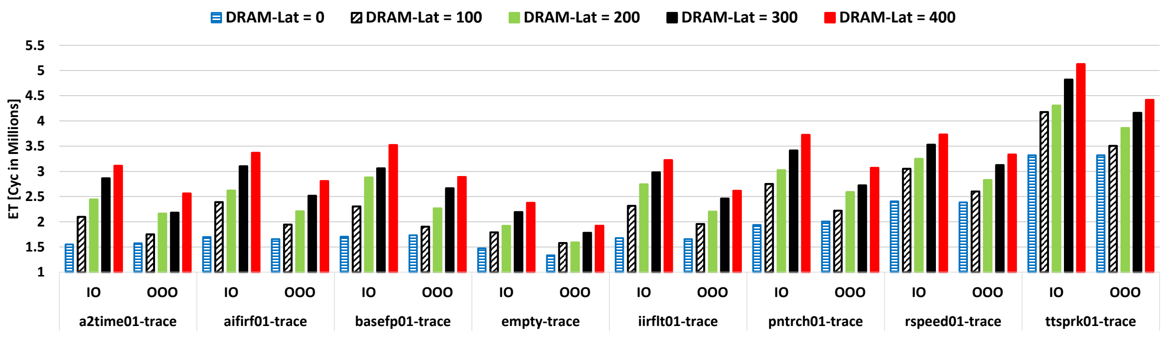


Figure 6.25: Standalone DRAM Latency impact on the Execution Time for the EEMBC benchmark with L2-Lat = 25 cycles. (MESI, FCFS).

Figure 6.26 shows the overall execution time of the synthetic benchmarks for in-order and out-of-order cores at different DRAM latencies and L2 cache response latencies. From this result, we make the following observations. 1) Clearly, out-of-order execution (Npend = 8) improves performance significantly compared to in-order cores. For instance, in DRAM-Lat = 400 cycles and L2-Lat = 25 cycles configurations, out-of-order cores achieves up to 2.86 × (Stride512Bytes0Cycle) improvement compared to in-order cores. 2) Notably, increasing DRAM latency degrades performance significantly in case of in-order cores for the benchmarks with higher L2 miss rates (i.e. Random0Cycle, Random0CycleW, and Stride512Bytes0Cycle), while interestingly for out-of-order cores, the execution time remains almost constant across the same benchmarks. 3) Increasing L2 cache response latency has the same effect on performance (reducing the execution time) of the running applications for both in-order and out-of-order cores. In Figure 6.27, we run one instance of the Stride512Bytes0Cycle workload that has 25% L2 miss rate, and sweep the DRAM latency from 0 to 400 cycles. Then, we repeat the same experiment for in-order and out-of-order cores at different L2 cache response latency numbers. As the figure illustrates, 1) when L2 latency is small enough (i.e 25 cycles case), the impact of increasing the DRAM latency on the performance happens immediately, Once the DRAM latency exceeds 50 cycles, the performance starts degrading linearly. On the other hand, for larger L2 response latency (i.e. 50 cycles), the performance is dominated by the L2 response itself until the DRAM latency reach a certain limit (above 150 cycles in this case), then the effect of the DRAM latency starts contributing to the performance degradation. 2) The out-of-order configuration is useful when L2 miss ratio is high, in this case increasing the CPU instruction pipeline hides the effect of the DRAM latency on the overall application execution time.

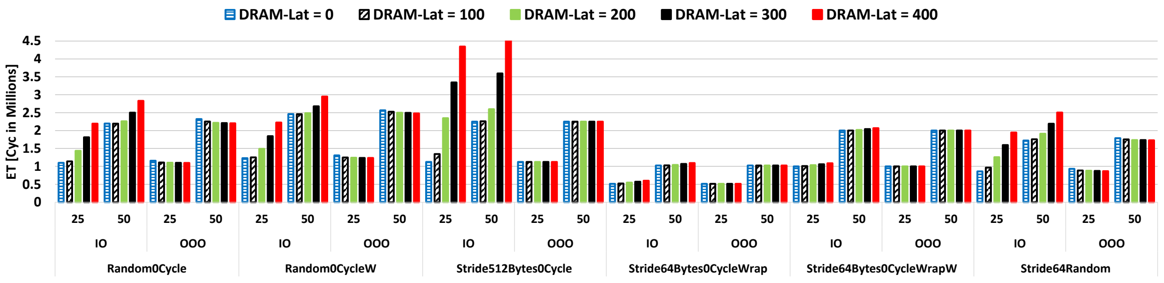


Figure 6.26: Execution Time with different DRAM latency, Npend = 8, L2-Lat = 25 and 50 cycles.

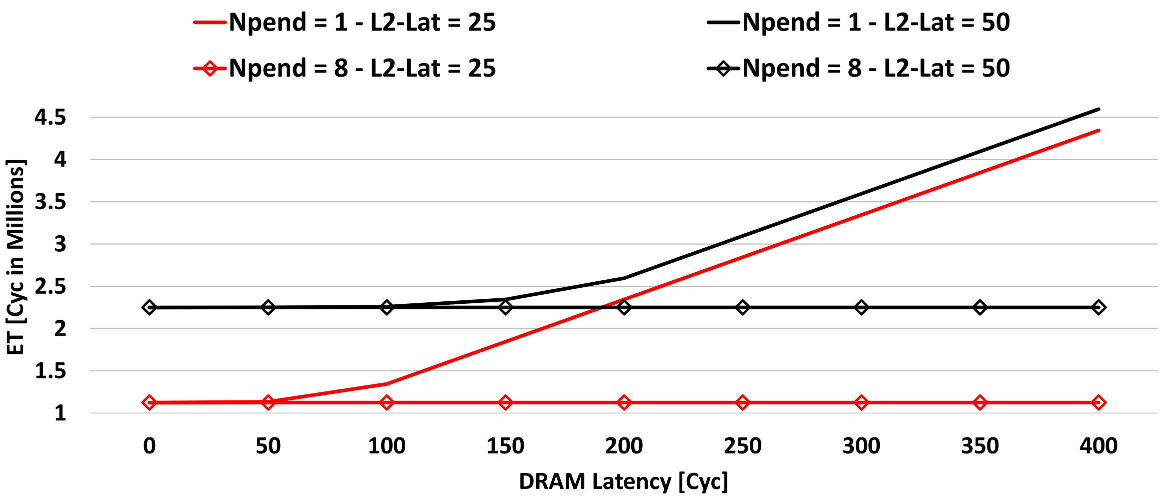


Figure 6.27: Execution Time with different DRAM latency

6.3 Tests for Nonfunctional Requirements

6.3.1 Configurability

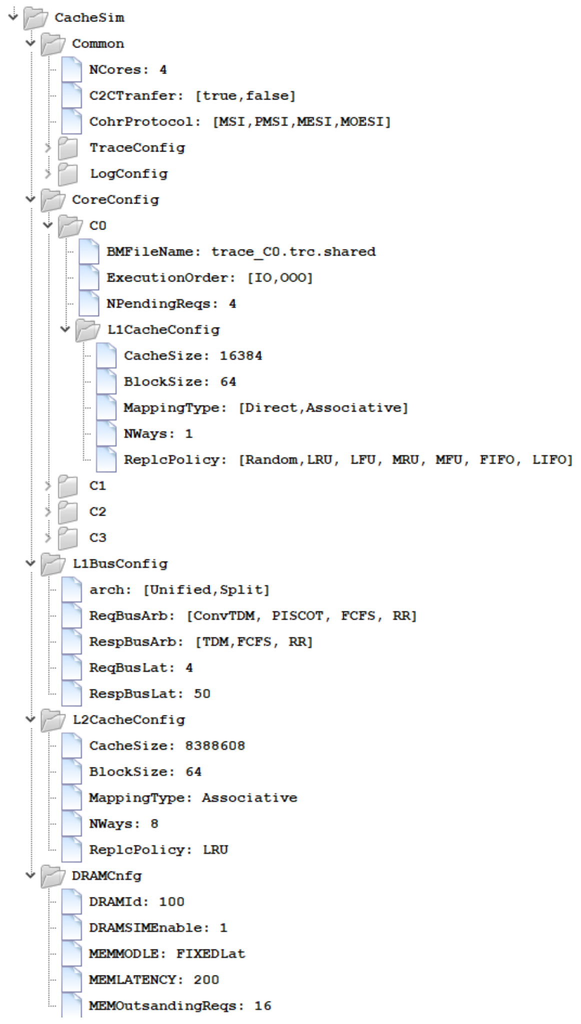
CacheSim allows a high degree of configurability of cache memory configuration in terms of the number of levels, cache organization, and replacement policy. Besides, it supports flexible cache coherence models and bus arbitration policies. These system specification parameters are grouped into a single XML document (Figure 6.28) that allows the simulator to be fully configurable in advance by the user.

6.3.2 Producability

Reproducibility of experiments is very important in order to study the impacts of different parameters' changes. Therefore, CacheSim is designed for producibility. All experiments we run in this evaluation can be regenerated by running a simple bash script that takes the test case category and benchmark name as an input, and then it lunches CacheSim to run the simulation. For example, the following command can be used to run DRAM latency test case number 2 using EEMbcTrace benchmark number 5.

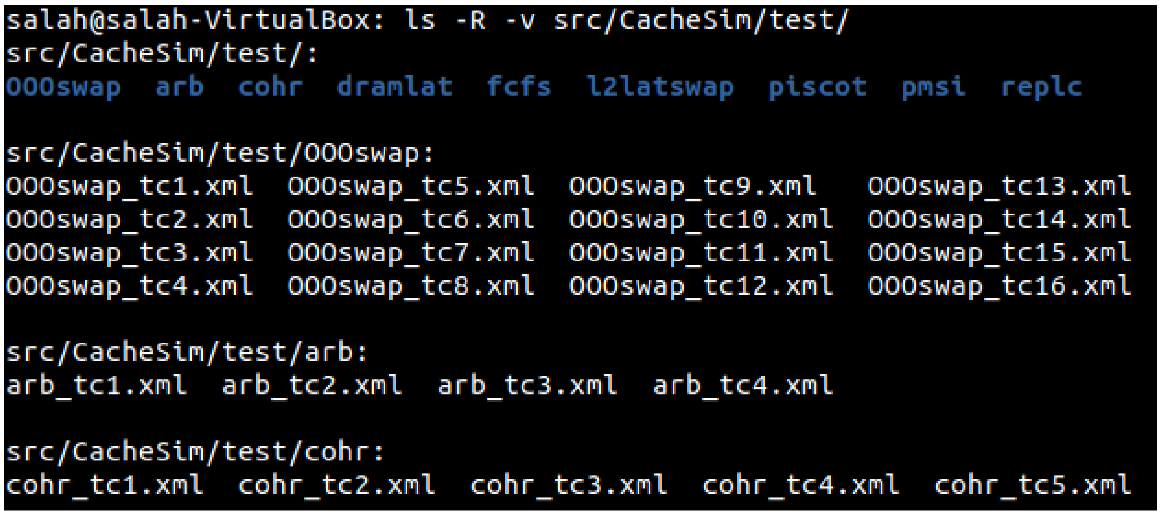
./do_run_tcs.sh dramlat 2 EEMbcTrace Directed 5

All testcase configuration files are stored in the "test" directory as shown by Figure 6.29



```
CacheSim
  Common
    NCores: 4
    C2CTranfer: [true,false]
    CohrProtocol: [MSI,PMSI,MESI,MOESI]
    TraceConfig
    LogConfig
  CoreConfig
    C0
      BMFileName: trace_C0.trc.shared
      ExecutionOrder: [IO,OOO]
      NPendingReqs: 4
      L1CacheConfig
        CacheSize: 16384
        BlockSize: 64
        MappingType: [Direct,Associative]
        NWays: 1
        ReplcPolicy: [Random,LRU, LFU, MRU, MFU, FIFO, LIFO]
    C1
    C2
    C3
  L1BusConfig
    arch: [Unified,Split]
    ReqBusArb: [ConvTDM, PISCOT, FCFS, RR]
    RespBusArb: [TDM,FCFS, RR]
    ReqBusLat: 4
    RespBusLat: 50
  L2CacheConfig
    CacheSize: 8388608
    BlockSize: 64
    MappingType: Associative
    NWays: 8
    ReplcPolicy: LRU
  DRAMCnfg
    DRAMId: 100
    DRAMSIMEnable: 1
    MEMMODLE: FIXEDLat
    MEMLATENCY: 200
    MEMOutsandingReqs: 16
```

Figure 6.28: CacheSim XML configuration file format.



```
salah@salah-VirtualBox: ls -R -v src/CacheSim/test/
src/CacheSim/test/:
OOOswap  arb  cohr  dramlat  fcfs  l2latswap  piscot  pmsi  replc

src/CacheSim/test/OOOswap:
OOOswap_tc1.xml  OOOswap_tc5.xml  OOOswap_tc9.xml   OOOswap_tc13.xml
OOOswap_tc2.xml  OOOswap_tc6.xml  OOOswap_tc10.xml  OOOswap_tc14.xml
OOOswap_tc3.xml  OOOswap_tc7.xml  OOOswap_tc11.xml  OOOswap_tc15.xml
OOOswap_tc4.xml  OOOswap_tc8.xml  OOOswap_tc12.xml  OOOswap_tc16.xml

src/CacheSim/test/arb:
arb_tc1.xml  arb_tc2.xml  arb_tc3.xml  arb_tc4.xml

src/CacheSim/test/cohr:
cohr_tc1.xml  cohr_tc2.xml  cohr_tc3.xml  cohr_tc4.xml  cohr_tc5.xml
```

Figure 6.29: CacheSim test case configuration files.

Chapter 7

Conclusion

This thesis focuses on building a comprehensive, cycle accurate simulation tool for cache-coherent multi-core architectures to help researchers and computer architects explore the design space challenges of multi-core processing chips. We introduced CacheSim, an open-source cycle-accurate simulation tool for cache-coherent interconnect architecture. We demonstrated the CacheSim's advantage in efficiency and extensibility, as well as its holistic support for cache coherency models and modern bus-based interconnects for multi-core architectures. We also used CacheSim to explore the design space of improving shared memory access predictability and developed PISCOT, a predictable and coherent bus architecture that provides a considerably tighter latency bound compared to the existing solutions. We hope that CacheSim would facilitate innovation in multi-level cache memory design in an era when technological performance is undergoing rapid saturation.

7.1 Future Work

In terms of future work, we will continue extending the simulator's capabilities to support configurable multi-level cache hierarchy. Currently, we support the common architecture deployed in real-time systems with two-level cache hierarchy where the L1 cache level is private per each processing core, and the L2 cache level is shared among all core. However, the high-performance platforms support up to three cache hierarchy levels, so we will improve this feature by making the number of cache levels a configurable parameter. We will also add cache exclusivity, directory-based coherence protocols, and network-on-chip (NoC) interconnects features into the simulator. Finally, full-system simulation experiments will be added to test CacheSim with external CPU simulator tools such as gem5 [12] and external DRAM device simulators such as Ramulator [43].

Appendix A

Your Appendix

A.1 Benchmarks

As seen before, we used 16 synthetic benchmarks to stress the verification of the coherence protocols and replacement algorithms. The descriptions of these benchmarks are listed in Table A.1

Table A.1: Synthetic Benchmarks Description

Benchmark	Description
RWStride16	Sequential read-modify-write memory accesses with stride offset equals 16 Bytes.
RWStride16NoIntrf	Sequential read-modify-write memory accesses with stride offset equals 16 Bytes, and no shared data between cores
RWStride8Rand	Random read-modify-write memory accesses with stride offset equals 8 Bytes.
RWStride32Rand	Random read-modify-write memory accesses with stride offset equals 32 Bytes.
R75RandL1Miss	Random memory accesses with read percentage equals 75% and all requests are miss in L1 Cache.

R85RandL1Miss	Random memory accesses with read percentage equals 85% and all requests are miss in L1 Cache.
R75RandWrap	Random memory accesses with read percentage equals 75% and address wrapping every 1024 KBytes.
R40RandWrap	Random memory accesses with read percentage equals 40% and address wrapping every 1024 KBytes.
Synth1 - Synth8	Random memory accesses with different L2 cache miss rates and patterns.

A.2 CacheSim Module Hierarchy

Decomposing a system into modules is a commonly accepted approach to developing software. We advocate CacheSim decomposition based on the principle of information hiding [54]. This principle supports design for change, because the "secrets" that each module hides represent likely future changes. Design for change is valuable in SC, where modifications are frequent, especially during initial development as the solution space is explored. Figure A.1 illustrates the use relation between the modules. It can be seen that the graph is a directed acyclic graph (DAG). Each level of the hierarchy offers a testable and usable subset of the system, and modules in the higher level of the hierarchy are essentially simpler because they use modules from the lower levels.

A.3 CacheSim Code Coverage Analysis

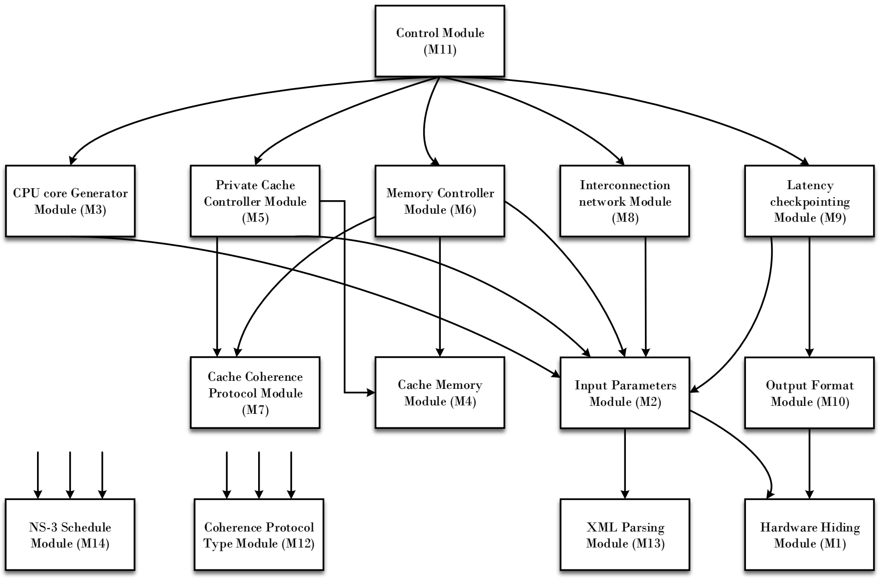
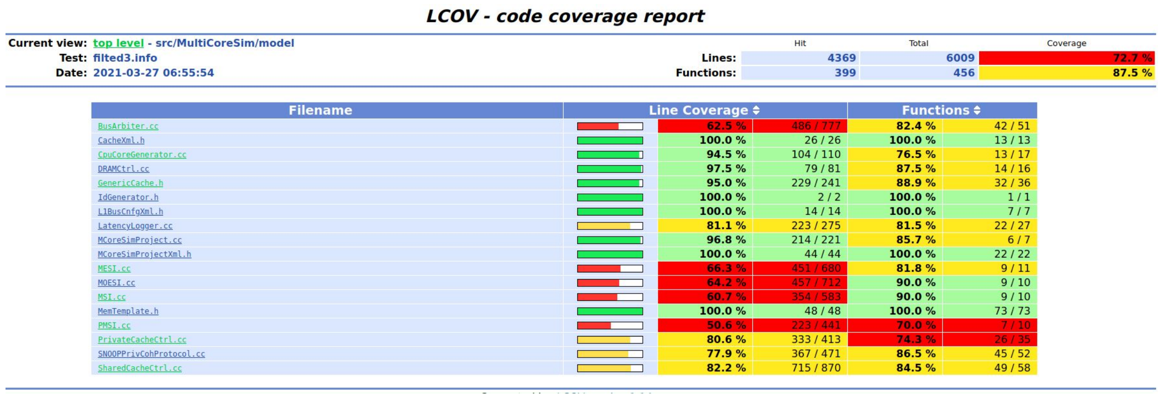

Control Module (M11)
CPU core Generator Module (M3)
Private Cache Controller Module (M5)
Memory Controller Module (M6)
Interconnection network Module (M8)
Latency checkpointing Module (M9)
Cache Coherence Protocol Module (M7)
Cache Memory Module (M4)
Input Parameters Module (M2)
Output Format Module (M10)
NS-3 Schedule Module (M14)
Coherence Protocol Type Module (M12)
XML Parsing Module (M13)
Hardware Hiding Module (M1)

Figure A.1: CacheSimModule Hierarchy



LCOV - code coverage report

Current view:	top level - src/MultiCoreSim/model			
		Hit	Total	Coverage
Test:	filted3.info			
	Lines:	4369	6009	72.7 %
Date:	2021-03-27 06:55:54			
	Functions:	399	456	87.5 %

Filename	Line Coverage			Functions	
BusArbiter.cc		62.5 %	486 / 777	82.4 %	42 / 51
CacheXml.h		100.0 %	26 / 26	100.0 %	13 / 13
CpuCoreGenerator.cc		94.5 %	104 / 110	76.5 %	13 / 17
DRAMCtrl.cc		97.5 %	79 / 81	87.5 %	14 / 16
GenericCache.h		95.0 %	229 / 241	88.9 %	32 / 36
IdGenerator.h		100.0 %	2 / 2	100.0 %	1 / 1
L1BusCnfgXml.h		100.0 %	14 / 14	100.0 %	7 / 7
LatencyLogger.cc		81.1 %	223 / 275	81.5 %	22 / 27
MCoreSimProject.cc		96.8 %	214 / 221	85.7 %	6 / 7
MCoreSimProjectXml.h		100.0 %	44 / 44	100.0 %	22 / 22
MESI.cc		66.3 %	451 / 680	81.8 %	9 / 11
MOESI.cc		64.2 %	457 / 712	90.0 %	9 / 10
MSI.cc		60.7 %	354 / 583	90.0 %	9 / 10
MemTemplate.h		100.0 %	48 / 48	100.0 %	73 / 73
PMSI.cc		50.6 %	223 / 441	70.0 %	7 / 10
PrivateCacheCtrl.cc		80.6 %	333 / 413	74.3 %	26 / 35
SNOOPPrivCohProtocol.cc		77.9 %	367 / 471	86.5 %	45 / 52
SharedCacheCtrl.cc		82.2 %	715 / 870	84.5 %	49 / 58

Figure A.2: CacheSim code coverage analysis

Bibliography

[1] M. Hassan, A. M. Kaushik, and H. Patel. Predictable cache coherence for multi-core real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

[2] S. Schliecker, J. Rox, M. Negrean, K. Richter, M. Jersak, and R. Ernst. System level performance analysis for real-time automotive multicore and network architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2009.

[3] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *Ninth European Dependable Computing Conference*, 2012.

[4] Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. A survey on cache management mechanisms for real-time embedded systems. *ACM Computing Surveys (CSUR)*, 48(2):32, 2015.

[5] Intel Corporation. Intel acknowledges it was too aggressive with its 10nm plans, 2019. URL `https://www.extremetech.com/computing/295159-intel-acknowledges-its-long-10nm-delay-caused-by-being-too-aggressive`.

[6] M. Cornea. New technologies for improved computing. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 96–96, 2019. doi: 10.1109/ARITH.2019.00024.

[7] A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 27–29, 2018. doi: 10.1109/ISCA.2018.00011.

[8] S. Hessien and M. Hassan. The best of all worlds: Improving predictability at the performance of conventional coherence with no protocol modifications. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 218–230, 2020. doi: 10.1109/RTSS49844.2020.00029.

[9] Mohamed Hassan. Discriminative coherence: Balancing performance and latency bounds in data-sharing multi-core real-time systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 1–22, 2020.

[10] Salah Hessien. Cachesim: A cycle-accurate simulation infrastructure for cache-coherent interconnect architectures. `https://gitlab.com/FanosLab/cachesim`, 2021.

[11] R. Mirosanlou, D. Guo, M. Hassan, and R. Pellizzoni. Mcsim: An extensible dram memory controller simulator. *IEEE Computer Architecture Letters*, 19(2): 105–109, 2020. doi: 10.1109/LCA.2020.3008288.

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt,

Ali Saidi, Arkaprava Basu, Joel Hestness, et al. The gem5 simulator. *ACM SIGARCH Comput. Archit. News*, 2011.

[13] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. PLDI '07, page 89–100, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936332. doi: 10.1145/1250734.1250746. URL `https://doi.org/10.1145/1250734.1250746`.

[14] Jan Edler. Dinero iv trace-driven uniprocessor cache simulator. retrieved from. URL `http://www.cs.wisc.edu/&sim;markhill/DineroIV/`.

[15] R. Iyer. On modeling and analyzing cache hierarchies using casper. In *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003.*, pages 182–187, 2003. doi: 10.1109/MASCOT.2003.1240655.

[16] Hyesoon Kim, Jaekyu Lee, Nagesh B Lakshminarayana, Jaewoong Sim, Jieun Lim, and Tri Pho. Macsim: A cpu-gpu heterogeneous simulation framework user guide., 2012.

[17] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore smp systems. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[18] Michael E Thomadakis. The architecture of the Nehalem processor and Nehalem-EP SMP platforms. *Resource*, 3, 2011.

[19] Manpreet S Khaira. Fast first-come first served arbitration method, November 12 1996. US Patent 5,574,867.

[20] WL Bain Jr and SR Ahuja. Performance analysis of high-speed digital buses for multiprocessing systems. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 107–133, 1981.

[21] Michael A Fischer. Fair arbitration technique for a split transaction bus in a multiprocessor computer system, November 15 1988. US Patent 4,785,394.

[22] Ashok Singhal, Bjorn Liencres, Jeff Price, Frederick M Cerauskis, David Broniarczyk, Gerald Cheung, Erik Hagersten, and Nalini Agarwal. Implementing snooping on a split-transaction computer system bus, November 2 1999. US Patent 5,978,874.

[23] Farouk Hebbache, Mathieu Jan, Florian Brandner, and Laurent Pautet. Shedding the shackles of time-division multiplexing. In *IEEE Real-Time Systems Symposium (RTSS)*, 2018.

[24] Nivedita Sritharan, Anirudh Mohan Kaushik, Mohamed Hassan, and Hiren Patel. Enabling predictable, simultaneous and coherent data sharing in mixed criticality systems. 2019.

[25] Anirudh M. Kaushik, Paulos Tegegn, Zhuanhao Wu, and Hiren Patel. Carp: A data communication mechanism for multi-core mixed-criticality systems. In *IEEE Real-Time Systems Symposium (RTSS)*, 2019.

[26] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995. ISSN 0163-5964. doi: 10.1145/216585.216588. URL `https://doi.org/10.1145/216585.216588`.

[27] F. Sampaio, M. Shafique, B. Zatt, S. Bampi, and J. Henkel. Approximation-aware multi-level cells stt-ram cache architecture. In *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 79–88, 2015. doi: 10.1109/CASES.2015.7324548.

[28] James E. Smith and James R. Goodman. A study of instruction cache organizations and replacement policies. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, ISCA '83, page 132–137, New York, NY, USA, 1983. Association for Computing Machinery. ISBN 0897911016. doi: 10.1145/800046.801648. URL `https://doi.org/10.1145/800046.801648`.

[29] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 364–373, 1990. doi: 10.1109/ISCA.1990.134547.

[30] Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 2011.

[31] Paul Sweazey and Alan Jay Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus, 2011.

[32] ARM. ARM CoreLink CCI-550 Cache Coherent Interconnect, Technical Reference Manual, 2015. URL `https://static.docs.arm.com/100282/0001/corelink_cci550_cache_coherent_interconnect_technical_reference_manual_100282_0001_01_en.pdf`.

[33] Dimitrios Ziakas, Allen Baum, Robert A Maddox, and Robert J Safranek. Intel® quickpath interconnect architectural features supporting scalable system architectures. In *2010 18th IEEE Symposium on High Performance Interconnects*, pages 1–6. IEEE, 2010.

[34] Francesco Poletti, Davide Bertozzi, Luca Benini, and Alessandro Bogliolo. Performance analysis of arbitration policies for soc communication architectures. *Design Automation for Embedded Systems*, 8(2-3):189–210, 2003.

[35] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Bus-aware multicore wcet analysis through tdma offset bounds. In *2011 23rd Euromicro Conference on Real-Time Systems*, pages 3–12. IEEE, 2011.

[36] B. Cilku, B. Frömel, and P. Puschner. A dual-layer bus arbiter for mixed-criticality systems with hypervisors. In *IEEE International Conference on Industrial Informatics (INDIN)*, 2014.

[37] Marco Paolieri, Eduardo Quiñones, Francisco J Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for wcet analysis of hard real-time multicore systems. *ACM SIGARCH Computer Architecture News*, 37(3):57–68, 2009.

[38] Man-Ki Yoon, Jung-Eun Kim, and Lui Sha. Optimizing tunable wcet with shared resource allocation and arbitration in hard real-time multicore systems. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 227–238. IEEE, 2011.

[39] M. Hassan and H. Patel. Criticality- and requirement-aware bus arbitration for

multi-core mixed criticality systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[40] N. Sritharan, A. M. Kaushik, M. Hassan, and H. Patel. Hourglass: Predictable time-based cache coherence protocol for dual-critical multi-core systems. 2017.

[41] Sadagopan Srinivasan, Li Zhao, Brinda Ganesh, Bruce Jacob, Mike Espig, and Ravi Iyer. Cmp memory modeling: How much does accuracy matter?

[42] A. Hansson, N. Agarwal, A. Kolli, T. Wenisch, and A. N. Udipi. Simulating dram controllers for future system architecture exploration. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 201–210, 2014. doi: 10.1109/ISPASS.2014.6844484.

[43] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. 15(1):45–49, January 2016. ISSN 1556-6056. doi: 10.1109/LCA.2015.2414456. URL `https://doi.org/10.1109/LCA.2015.2414456`.

[44] Hadi Brais, Rajshekar Kalayappan, and Preeti Ranjan Panda. A survey of cache simulators. *ACM Comput. Surv.*, 53(1), February 2020. ISSN 0360-0300. doi: 10.1145/3372393. URL `https://doi.org/10.1145/3372393`.

[45] Aamer Jaleel, Robert S. Cohn, Chi keung Luk, and Bruce Jacob. Cmp$im: A pin-based on-the-fly multi-core cache simulator.

[46] Derek Bruening and Saman Amarasinghe. Efficient, transparent, and comprehensive runtime code manipulation. ph.d. dissertation. massachusetts institute of technology, department of electrical engineering and computer science., 2004. URL `http://www.cs.wisc.edu/&sim;markhill/DineroIV/`.

[47] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. Marss: A full system simulator for multicore x86 cpus. In *Proceedings of the 48th Design Automation Conference*, DAC '11, page 1050–1055, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306362. doi: 10.1145/2024724.2024954. URL `https://doi.org/10.1145/2024724.2024954`.

[48] J. H. Ahn, S. Li, S. O, and N. P. Jouppi. Mcsima+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 74–85, 2013. doi: 10.1109/ISPASS.2013.6557148.

[49] Hui Kang and Jennifer L Wong. vcsimx86: a cache simulation framework for x86 virtualization hosts. stony brook university., 2013.

[50] Raul Martin Miguel A Vega-Rodriguez and Francisco A Zarallo Gallardo. Simulator for cache memory systems on symmetric multiprocessors., 2006. URL `http://arco.unex.es/smpcache/`.

[51] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2sim: A simulation framework for cpu-gpu computing. PACT '12, page 335–344, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450311823. doi: 10.1145/2370816.2370865. URL `https://doi.org/10.1145/2370816.2370865`.

[52] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 475–486, New York, NY, USA, 2013. Association for Computing Machinery.

ISBN 9781450320795. doi: 10.1145/2485922.2485963. URL https://doi.org/10.1145/2485922.2485963.

[53] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, et al. The gem5 simulator: Version 20.0+, 2020. URL https://arxiv.org/abs/2007.03152.

[54] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(2):1053–1058, December 1972.

[55] The ns-3 simulator. URL https://www.nsnam.org.

[56] Tinyxml parsing library. URL https://sourceforge.net/projects/tinyxml.

[57] Gcovr code coverage analysis tool. URL https://gcovr.com/en/stable/guide.html.

[58] Waf build automation tool. URL https://waf.io/apidocs/tools.html.

[59] Jason Poovey et al. Characterization of the EEMBC benchmark suite. *North Carolina State University*, 2007.

[60] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016.

[61] Benchmark suites used for cachesim verification. https://gitlab.com/FanosLab/piscot/-/tree/master/BMs.