12-2022

Simncore: Multilevel Cache Memory Design Simulator for Manycore System

Provashish Roy
The University of Texas Rio Grande Valley

SIMNCORE: MULTILEVEL CACHE MEMORY DESIGN SIMULATOR FOR MANYCORE SYSTEM

A Thesis

by

PROVASHISH ROY

Submitted in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE IN ENGINEERING

Major Subject: Electrical Engineering

The University of Texas Rio Grande Valley

December 2022

SIMNCORE: MULTILEVEL CACHE MEMORY DESIGN SIMULATOR FOR MANYCORE SYSTEM

A Thesis

by

PROVASHISH ROY

COMMITTEE MEMBERS

Dr. Mark Yul Chu

Chair of Committee

Dr. Weidong Kuang

Committee Member

Dr. Wenjie Dong

Committee Member

December 2022

ABSTRACT

Roy, Provashish, Simncore: Multilevel Cache Memory Design Simulator For Manycore System. Master of Science in Engineering (MSE), December, 2022, 177 pp., 1 table, 96 figures, references, 37 titles.

The current trend in a processor design has moved from multicore to manycore (tens to hundreds, or more cores) to support more computational power based on parallelism. One of challenges is how to handle such large number of cores' data in cache memories through an efficient inter-core communication and cache coherence. To meet the demand, this paper presents a manycore cache memory simulator for research and education purposes. The proposed simulator, called as SIMNCORE, is to design and evaluate various multi-level, such as L1 and L2, cache memories for manycore processing. The SIMNCORE will implement various trace files collected from parallel benchmark programs, such as PARSEC or SPLASH2, by using the Pin Tool. The Pin Tool, developed by Intel, is to generate the traces of benchmark programs by intercepting the execution of the instructions and surveilling memory addresses. Our experimental evaluation shows that the SIMNCORE is highly efficient in designing and evaluating manycore cache memories by comparing it with the other well-established simulators, such as SMPCache or FM-SIM. Therefore, we expect SIMNCORE can be an effective simulation tool to conduct cache memory-related studies for research related purposes.

DEDICATION

The completion of my Masters in Electrical Engineering at The University of Texas would not be possible support of my family. My gratitude goes out to my Father, Gour Roy; my mother, Karuna Roy; my brother, Debashish Roy and my wife, Trina Sutradhar; for being what keeps me striving to be better every day.

ACKNOWLEDGMENTS

I would like to express my eternal gratitude towards Dr. Mark Yul Chu, Chair of my thesis committee, for the mentoring and advisement that he has provided me over the time I have gotten to know him. From research and data required for my work to the patience of making this manuscript the best it can be, he has provided much of what without which my work would remain incomplete. My gratitude also goes out to Dr. Weidong Kuang and Dr. Wenjie Dong, who have helped ensure the quality and integrity of the work presented here

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER I

INTRODUCTION

The CPU generation is moving from the multicore processor era to the manycore processor era. Though often contemplated coequally, multicore generally refers to 2-8 cores of processor, while manycore can range from a dozen to hundreds of cores. Multicore, and manycore processors have their own notable and distinctive characteristics and differences, such as performance per watt, single thread performance, etc. [1]. CPU with manycore processors performs better almost in every aspect as they often offer better throughput [2]. However, until lower memory latency is ensured, increasing the number of cores alone won't be able to guarantee better CPU performance. CPU performance is impeded by memory latency since memory cannot keep up with the processor's speed. Cache memory has been viewed as a crucial functional unit for addressing this problem [3]. Cache memory alleviates the performance bottlenecks between the processor and the system memory. Multiple studies have discovered that suitable memory configurations with the processor and different levels of caches avail more capacity, improve speed, and lead to better efficiency [4]. Thus, cache memory has become a key element in improving the performance of current multicore and manycore CPU systems.

Cache configurations often urge testing and simulation before employment. The goal of simulation is to evaluate detailed characteristics of the cache, such as cache misses, average memory access time, coherency effectiveness, bus traffic, power consumption, cost, etc. Cache simulators frequently come in two varieties: execution-driven and trace-based [5]. Execution-

driven simulators operate by executing a program's instructions directly (e.g., SimpleScalar [6]). Contrarily, trace-driven simulators operate by reading/parsing a recorded trace of instructions and outcomes (e.g., SMPCache [7]). According to [8], trace-driven simulators appear to be more preferred for general purposes than the other types of simulators since they are simple and convenient to work with, especially to test a new cache scheme or validate a new cache coherence protocol.

1.1 Motivation

There are a good number of cache memory simulators available on the market at present. Many of the simulators remain free and open source, but some are being used commercially. While only a small number of non-commercial cache simulators are regularly updated by their developers and the research community, most of them are rarely used nowadays due to the inaccessibility of the source code or the lack of support for accommodating recent innovations in cache memory technologies. Simulators that allow users to experiment with novel concepts and designs in caching investigations are consistently preferred by researchers. There are very few cache simulators that are attempting to support the late manycore architecture with multilevel cache memory. Because of the massive data overhead and complexity of simulation methodology, simulators are becoming exhausted when simulating manycore cache memory models [4]. Some simulators are well-liked for simulating multilevel cache memory with a single-core CPU, but they are not favored for simulating multicore designs (e.g., SimpleScalar) [8]. On the other hand, some simulators are capable of simulating manycore cache memory schemes with one level of cache but lack support

for multilevel cache memory (e.g., FM-SIM [8]). Consequently, there have been significant calls for building a simulator that can simulate different levels of cache memory in manycore CPU.

Hence, the brainwork of this thesis converges around the idea of designing a competent cache memory simulation tool that simulates multilevel cache memory for manycore systems, for general and academic purposes. For our designed trace-based cache simulator, instructions and data traces are required as the workload. Pin Tool [9] is used to collect traces by executing real-world and benchmark programs. Our proposed cache performance simulator lets users try out different cache hierarchies virtually so their performance and cost-effectiveness can be compared in great detail.

CHAPTER II

RELATED WORKS

Simplescalar [6] is a microarchitectural performance simulator suite. It's an execution-driven simulator that uses the portable instruction set architecture. Simplescalar is facilitated with many features like, trace traffic measurement, energy modeling, and so on, which have been employed in numerous research areas [10]. Though it provides a lot of details at the execution level, the simulation technique often makes the process time-consuming [11]. Besides, Simplescalar is practiced for single-core architectural models. Despite supporting different extensions for multithreading architectures (like MSim [12] or SSMT [13]), it hasn't been favored for studying multicore processors [14].

Multi2Sim [15] is an acknowledged simulation tool for CPU and GPU computation. It supports different instruction set architectures and multicore processor simulation with a long range of scalability. The core design can carry out timing simulation with its three simulation models: functional, detailed, and event driven. It is recognized for its low percentage of errors for any simulation [16]. Nonetheless, implementation of functional units, for instance, different level caches, is a strenuous task in this simulator [8].

Gem5 is considered a complete-system simulator with various types of ISAs and CPU model compatibility [17]. Its versatility makes it super popular among researchers. It inspires and supports a wide range of ongoing research projects, including the ARM multiprocessor CPU model, CPU power modeling, distributed computer clustering, and so on [18], [19], and [20]. In

some areas, it holds the best runtime efficiency, with an accuracy percentage of 87% to 95% for Spec CPU2006 [21] benchmark programs. It supports multithreaded and multiprocessor memory mapping with superscalar, pipelining, and configurable hardware features. However, just as professed as a timing simulator, it offers so much but leaves out a minimum scope for changes from the user's end [22]. Gem5 usually processes an enormous workload, even for simple executables, which often exhausts the system with a prolonged execution period [23]. That's why it can be easily perceived that the demanding and incomprehensible nature of this simulator makes it a bit unsuited for educational purposes.

According to Catline et al. [24], the MARS MIPS simulator [25] is a well-organized tool for the academic level. It is helpful for learning actions inside the hardware (register to register, register to memory, etc.) layer. However, it only operates with assembly language. Another renowned software for cache memory simulation is SMPCache [7]. It allows users to design and analyze caches using trace-based simulation. It is an open-source software like FM-Sim (Flexible Multicore Simulator) [8]. According to [4], SMPcache deals with some shortcomings like confined access to parameters, limited OS-based portability, etc.

2.1 Problem Statement

Trace-based simulators offer a few advantages over execution-driven simulators in the research and academic fields. The overhead of running the input program is entirely eliminated by trace-driven simulation. Once a trace file is created, simulation can be performed with it repeatedly while taking various settings into account. The trace file formats are typically unaffected by the simulators or platforms that the programs were run on. To evaluate CPUs and cache memory, benchmark programs (e.g., Parsec 3.0 [26], and Splash2 [27]) have become standards in the

computer system community. Simulators often use trace files generated from benchmark programs to evaluate the performance of cache memory. There are several resources to create tarce files. Some simulators (e.g., SMPCache) demand trace files in particular formats, which must be created using intricate processes, while other simulators use trace files that are very easy to generate. CMPSim [28] and FM-SIM are two such simulators that use trace files, generated by Intel PIN, for multicore cache memory simulation. Pin is a dynamic binary instrumentation framework that offers several tools to collect memory references, instructions, etc. Though creating trace files and performing simulation with such simulators is convenient, there are also some limitations. The source codes of CMPSim are not available for practice anymore. FM-SIM is intelligible for the unsophisticated simulation method through a command line, but it only offers level-1 cache design and simulation.

While inspired moderately by other simulators' aspects, this paper holds its exclusivity in the place of multi-level cache accessibility and simplicity for ground-level academic learning. Keeping in mind all the scopes for improvement, we have been wondering how to design a ready-to-go application with broad exploring features. We built a program, freeing the core dependency for the concurrent coherency with the cache memory. We chose to adopt the trace-based simulation method for its candid, yet explicit nature. Execution-driven applications always deal with massive data overhead due to runtime trace formation, which is tedious, and sometimes ends up in unreliable interpretation [5]. As the trace-driven software performs on prepped-up traces, it can wholly concentrate on simulation objectives. In addition, we prepared a guide to collect traces with Intel PIN, which can be an effective tool for manycore cache memory research.

CHAPTER III

PROPOSED SIMULATOR: SIMNCORE

Most of the cache simulators require much time to comprehend; and at times, generating trace files for them needs to follow complex procedures. Some simulators are easy to learn but often lack the multilevel cache scheme or the manycore processor model [5]. Therefore, our primary focus is to develop a design and simulation tool for multilevel cache memory in n-core (both multicore and manycore) processor.

The workflow of our proposed cache simulator, SIMNCORE, consists of four main steps from developers' point of view. First, we will design the simulator to construct different cache memory hierarchies based on cache size, block size, associativity, etc. Secondly, we will implement several coherence protocols to be associated with multicore and manycore system. After that, the trace data needs to be fed into the simulator. After entering all the required information and input file, the simulator will display the resulting data by evaluating the cache memory in terms of cache hit and cache miss. The flowchart in Figure 3.1 can give a glimpse of the working principle.

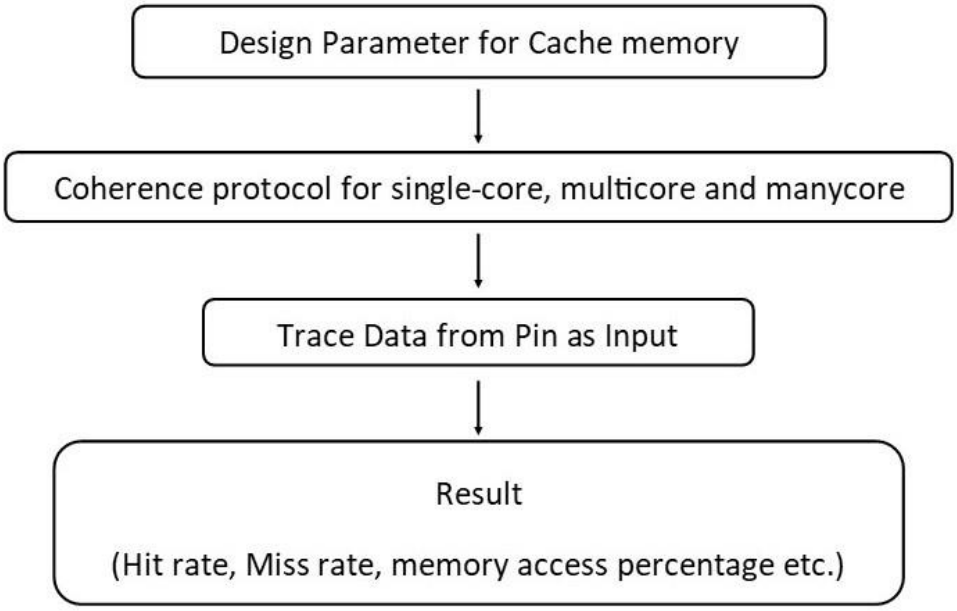

Figure 3.1 Working principle of the proposed simulator

3.1 Design Parameter for Cache Memory

Our simulator will give users the freedom to design their cache memory in their own terms. Through the command line, users can define block size, the number of index bits, write policy, set associativity, etc. After completing one task, the parameters can be changed again for a new observation. Figure 3.2 can explain the steps thoroughly.

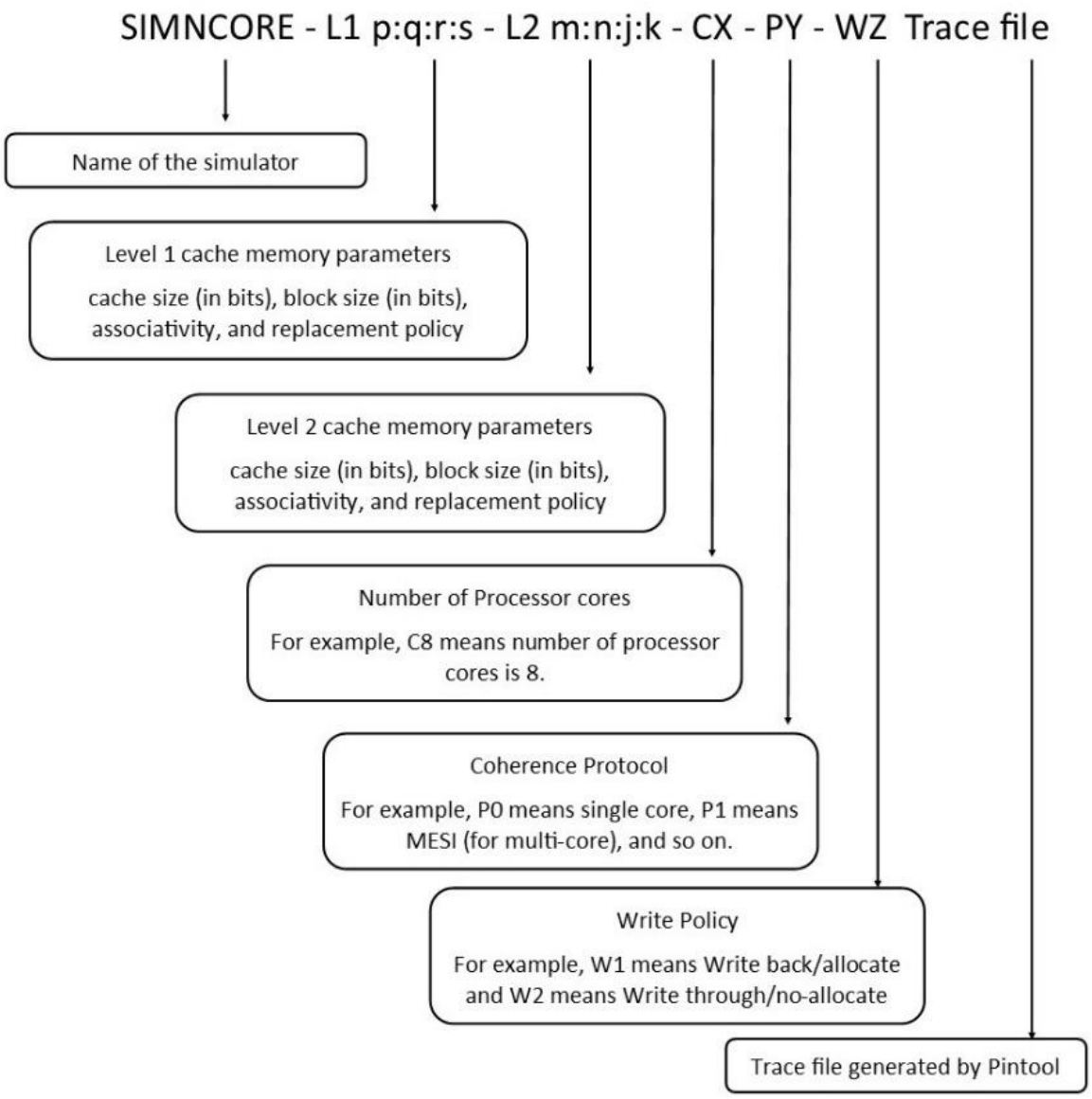

Figure 3.2: Command line for proposed simulator

The four values following L1 and L2 refer to Level 1 and Level 2 cache parameters, respectively. The values are given as log2 based numbers. The three values stand for cache size, block size, associativity, and replacement policy respectively. For example, the first value ‘p’ refers to a cache size of 2^p. The digit 'r' refers to the associativity of the respective cache. Our simulator can form different cache mappings like direct, 2-way, 4-way, etc. by defining r=0, 1, 2, … etc. For example, if value of ‘r’ is set as 2, then then associativity will be 22 = 4. As well, if for

r=4, associativity is 24 = 16. To set Direct-mapped cache, value of ‘r’ should be 0. In SIMNCORE, we use LRU (Least Recently Used) as our replacement policy and is denoted by the last digit ‘r’ = 1. Developers can port new replacement policies in the simulator and assign different values to access them.

For example, let’s consider a user wants to construct, 16 KB L1 32 bytes block size 2-way set associative and 256 KB L2 128 bytes block size 8-way set associative cache with LRU replacement policy, then the terminal instruction should be, {-l1 14:5:1:1 -l2 18:7:3:1}. To build evaluation only for L1 cache, users can define the case setting L2 parameters as 0. ‘C’ stands for 'Cores'. So, by expressing 'CX', an 'X' number of cores is declared for the system. A coherent protocol for the multicore system should also be declared in the command. We have designed four protocols so far. Single-Core, MESI, MSI/SNOOPY, and FIREFLY are labeled as P0, P1, P2 and P3 respectively. New coherence protocols can be added in the simulator and introduced with new value in ‘P’ like P4, P5, and so on. The letter 'Z' following the letter 'W' indicates the write policy, which is write-back/write-allocate for Z=1 and write-through/write no-allocate for Z=2. In the end, trace files need to be added to the command. More of the details for the simulator are included in the manual provided with the simulator source code (refer to Appendix).

Before each cache entry in the cache memory, index, tag, and block data are needed to be collected from each memory address. If we know the cache size and the block size, we can deduce the index and tag information using equation 3.1 and 3.2 [29].

$$index\ bit = log_2\left(\frac{cache\ size}{associativity\ X\ block\ size}\right) \quad \text{.....3.1)}$$

$$tag\ bit = memory\ address\ size\ (32\ bit\ or\ 64\ bit) - index\ bit - block\ bit \quad \text{.....3.2)}$$

For example, for a 16 KB Cache with 32 bytes block size, 4-way set-associativity and 32bit long memory address, tag, index and block are 20, 7, and 5 bits long, respectively.

This step is very crucial, as simulator look for a particular cache block by searching the tag. If the tag from the memory address tag bits is found in any cache block, that means a cache 'Hit'. Otherwise, it will be counted as a cache 'Miss' and follow the algorithm instruction to decide next step. More of the hit and miss events instruction will be discussed in next section.

3.2 Replacement Policy

As cache blocks are often crowded with data, sometimes a new data entry requires replacing any existing cache block. That is why, in particular, caching requires a data replacement policy. For cache algorithms, some popular replacement policies are RR (random replacement), FIFO (first in, first out), LIFO (last in, first out), and LRU (least recently used). SIMNCORE is equipped with the LRU replacement policy. For every cache block, the simulator sets up a separate value, named 'LRU', through a 'Update LRU' function. When a cache block gets a hit, its LRU value is set to 0. For every fetched instruction, the 'Update LRU' function increments the value of each cache block if it's not a hit. LRU value helps to determine which cache block should be replaced first. For any new entry of data from memory or processor, the algorithm first checks for unoccupied space to place the new data. If no empty space is found, the 'Update LRU' function checks the highest LRU value within the cache bank (which is least recently used), to replace it with new data in its place.

3.3 Cache Coherence Policy

For the consistency of data, we need to choose the coherence policy for the simulator. A single core doesn't require the coherence policy as the data is accessed by only one processor. But for multicore usage, the simulator requires a coherence policy (like SNOOPY or MESI [29, 30, 31]) for validation and uniformity of data while being used among different processors. The datapath between the processing unit/s and main memory is given in Figure 03, further illustration.

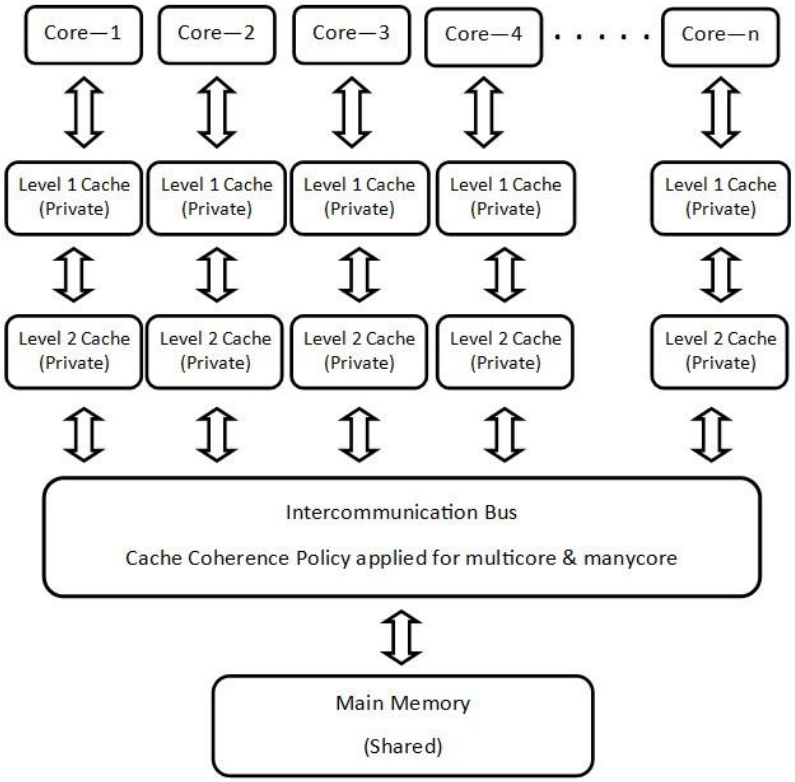

Figure 3.3: Basic manycore cache memory configurations

3.3.1 Caching in Single-core Processor

Our simulator constructs both level-1 (L1) and level-2 (L2) cache, fetched from command line parameters. And of course, alongside manycore simulation, SIMNCORE is capable of

uniprocessor simulation as well. A single processor architecture with two levels of cache is shown in Figure 4. Caching results in two decisions, Hit or Miss. Based on the type of instruction, we can catalog the events in four cases: 1) cache hit for Read and other instructions, 2) cache hit for Write instructions, 3) cache miss for Read and other instructions, and 4) cache miss for Write instructions.

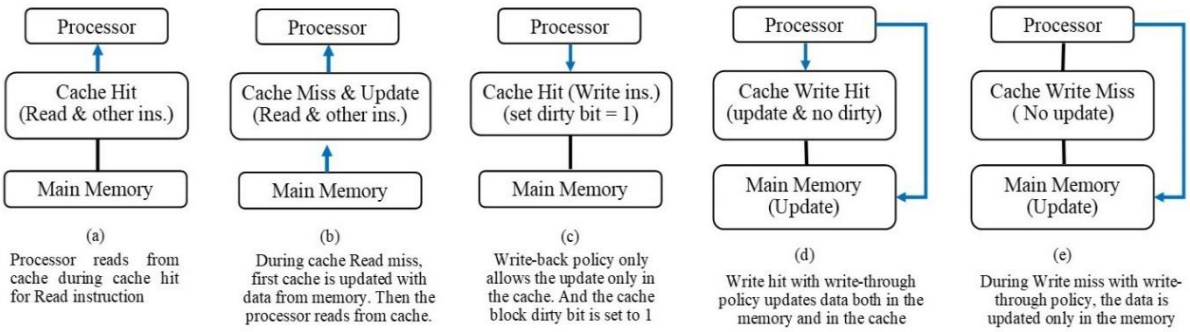

Figure 3.4: Cache Hit and Cache Miss events on Single-core processor

Cache hit for read and other instructions refers the processor requested data is found in the cache, and the processor can read the data from the cache block (Figure 3.4(a)). Cache miss for Read and other instruction indicates the requested data is not in the cache, and the processor needs to access the data from the memory. In this case, at first the data is updated in the cache memory, and then processor reads the data from the cache (Figure 3.4(b)).

Write operation in the cache demands a writing policy. One of the writing policies is ‘Write-back’. If the write policy is write-back, the data is only updated in the cache and not in memory at the same time during a write hit (Figure 3.4(c)). Rather, it triggers an extra control bit in the updated cache block. That control bit is called ‘Dirty bit’. After a write operation is executed in a particular cache block, the dirty bit associated with it becomes 1, which means the data it is holding has been modified and is different from the one located in the main memory. The update

of main memory takes place when the dirty cache blocks need to be replaced. After the main memory is updated, the dirty bit from the cache block becomes 0. The other writing policy, 'Write-through', keeps the main memory and the cache in sync during a write hit. That means when a write hit is found, the data is updated both in the cache and in the main memory (Figure 3.4(d)). From a performance perspective, a write-back policy can save a lot of bandwidth, as it doesn't have to update memory every time it encounters a write operation, unlike the write-through policy [32]. In the event of a write miss, the write-back policy allows the update only in cache. The write-through/write no-allocate policy prevents the cache from being updated during a write miss. Updates occur directly in memory (Figure 3.4(e)).

Both Write-back policy, and Write-thourgh policy, designed for single-core processor with two level of cache in SIMNCORE, is discussed here.

3.3.1.1 Write-Back Policy for Two Levels of Cache in Single-Core Processor. As an inclusive cache model, all of L1 cache data and their respective states are always in sync with L2 cache. That means, if any update happens in any particular cache block in L1, it corresponding L2 copy will go through the same update. When use the write-back policy for any simulation, both L1 and L2 cache will follow the policy. In this circumstance, both levels of cache are allowed to have dirty bits. As for a cache block, it is said to be dirty if it carries any recently written value which is not updated in the main memory.

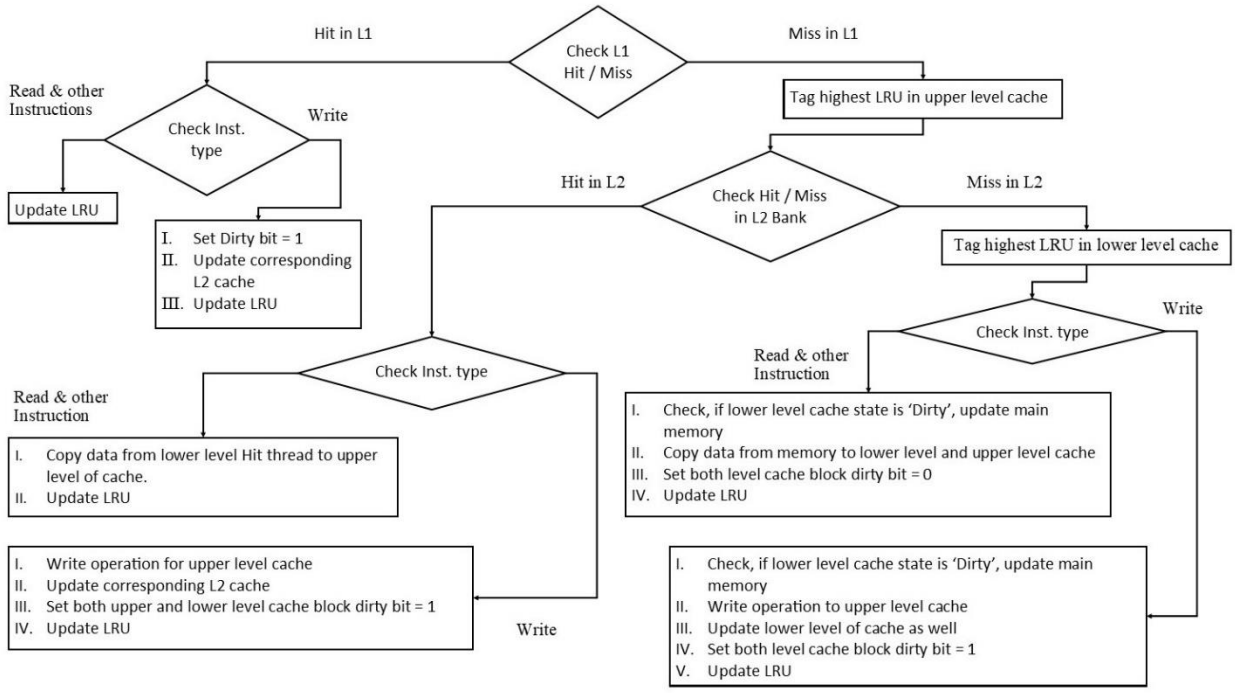

Figure 3.5: Write-back policy for Two Levels of Cache in Single-Core Processor

Now for case studies for hit and miss events in uniprocessor and two levels of caches with write-back policy, we formulate the algorithm which is shown in Figure 3.5. The events are described as follows:

a) *Cache Hit in L1 (read and other instructions)*: When a hit is found, the simulator just updates the LRU of the cache block. The 'Update LRU' function assists the simulator in determining which cache blocks have recently been hit. The lowest LRU value refers to the most recently used cache block. LRU functionality is discussed in section 3.3.

b) *Cache Hit L1 (write instructions):* For the write-back policy, the L1's hit cache block is updated with a new value and the dirty bit is set to 1. The corresponding L2 cache is updated accordingly. As the cache block is written, its LRU value will be updated too.

c) *Cache Miss in L1 but hit in L2 (read and other instructions)*: The processor has the fastest communication with the nearest cache, which is L1. When the processor finds a miss in L1

and later finds a hit in L2, at first, L1 is updated with L2 hit cache block. It will copy data to L1's empty cache block and then the processor will read the data. But if no empty space is found on L1, copying new data to it often leads to evicting or flushing its existing data. Approach on new data allocation or replacement of an existing cache block needs to follow some steps:

i) If no empty place is found on L1, the simulator checks the highest LRU value for the L1 cache bank. The highest LRU value means that the cache block is the least recently used (Figure 3.6(a)). For identification, we name this cache block 'Target L1'. 'Target L1' should be selected depending on the replacement policy.

ii) The hit data (tag, and block data) from L2 will be copied to Target L1 (Figure 3.6(b)).

iii) The processor will move to its read operation (Figure 3.6(c)) and update the LRU.

d) *Cache Miss in L1 but Hit in L2 (write instructions):* Firstly, this scenario will follow the last event's step (i) as described for the read operation. Then the processor writes new data to the L1 & L2 cache block, sets their dirty value to 1, and updates the LRU.

e) *Cache Miss in both L1 and L2 (read and other instructions):* When both levels encounter a cache miss, the data need to be copied to the cache from main memory. The steps are as follows:

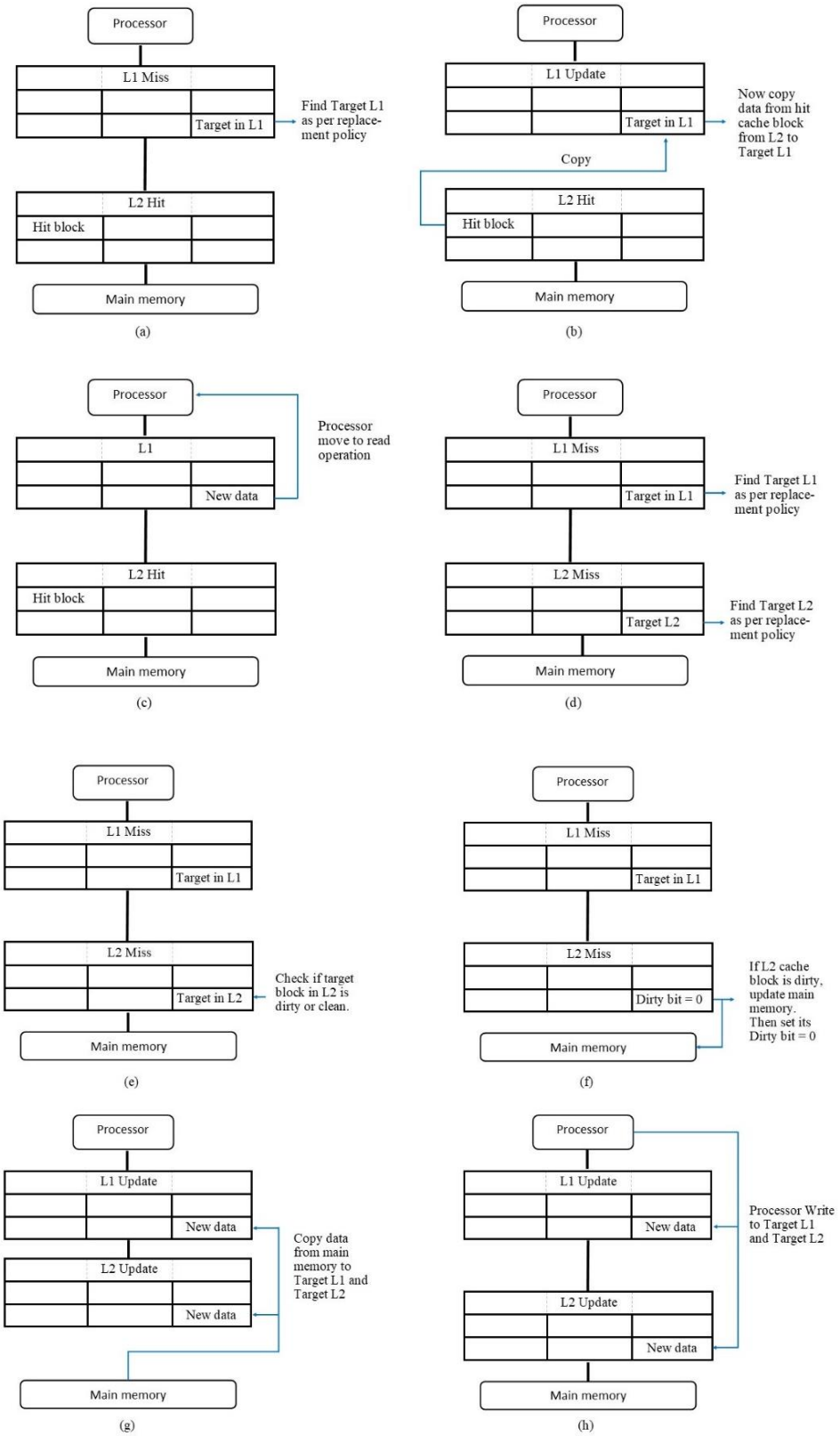

Figure 3.6: Data allocation/replacement approach in L1 and L2 cache

i) For copying new data to L1 and L2 from memory, each cache level will choose the highest LRU valued cache block and set them as ‘Target L1’ and ‘Target L2’ respectively (Figure 3.6(d)).

ii) Before old data is flushed from ‘Target L2’, the dirty bit of ‘Target L2’ is checked (Figure 3.6(e)).

iii) If ‘Target L2’ is dirty, updated cache data from ‘Target L2’ is written back to memory, and the ‘Target L2’ dirty bit is set to 0 (Figure 3.6(f)).

iv) Then desired data from the main memory is copied to ‘Target L1’ and ‘Target L2’ (Figure 3.6(g)).

v) After that, the processor executes the read operation, and updates the LRU.

f) *Cache Miss in both L1 and L2 (write instructions):* Goes as described in the last event from step (i) to step (iii). Then write operation updates L1 and L2 cache block with new data, sets dirty bit = 1, and updates the LRU (Figure 3.6(h)).

3.3.1.2 Write-Through Policy for Two Levels of Cache in Single-Core Processor. Write-through policy discards the concepts of ‘Dirty Bit’. Whenever the processor writes to the cache, it also writes to memory simultaneously. Anytime if the cache block data is evicted, and later the CPU needs it again, the valid and updated data can be found in the main memory. This approach is simple and relatively inexpensive to implement. However, using a write-through cache cause a lot of memory traffic as writing to memory is a time-consuming task [32].

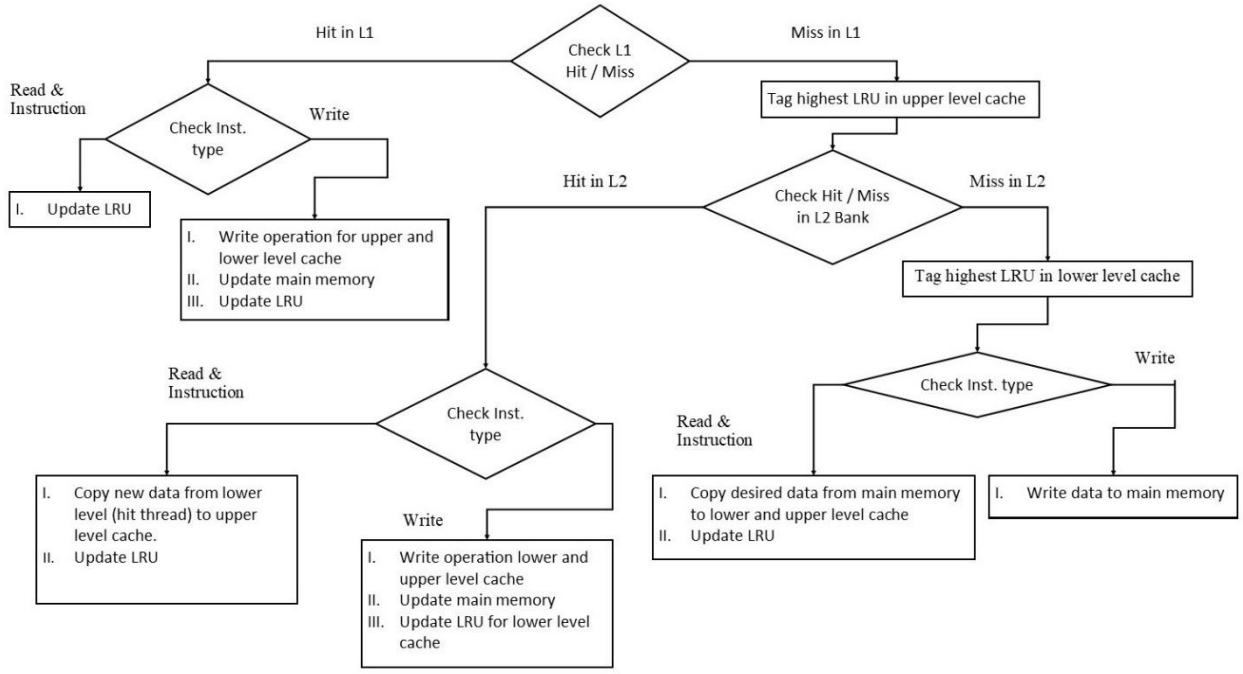

Figure 3.7: Write-through policy for Two Levels of Cache in Single-Core Processor

Hit and miss events in uniprocessor and two levels of caches are shown in Figure 3.7. The events are described as follows:

a) *Cache Hit in L1 (read and other instructions)*: When a hit is found, only LRU is updated.

b) *Cache Hit L1 (write instructions):* For the write-through policy, upper and lower level cache gets new data due to write hit. Then main mameory is updated accordingly and the LRU is updated.

c) *Cache Miss in L1 but Hit in L2 (read and other instructions)*: When the processor finds a miss in L1 and later finds a hit in L2, at first, L1 is updated with L2 hit cache block. To put new data in L1 cache LRU replacement policy is considered, if no empty cache block is found. Lastly, LRU is updated for both level of caches.

d) *Cache Miss in L1 but Hit in L2 (write instructions):* Firstly, this scenario will follow the step (b) as described for the write operation. But since no cache hit is found in L1 cache,

simulator marks highest LRU cache block to place new data in L1 cache. Then the L2 cache block updated accordingly, as a hit is found in L2 cache. Then the main mmeroy is updated too and the LRU is updated..

e) *Cache Miss in both L1 and L2 (read and other instructions):* When both levels encounter a cache miss, the data is copied to both levels of caches from the main memory. Then the LRU is updated.

f) *Cache Miss in both L1 and L2 (write instructions):* For write operation in write-through policy, if no cache hit is found in either of the cache level, the processor doesn't place the new data to any of the caches. Instead, the update takes place only in the main memory. The process ends with the update of LRU.

3.3.2 Caching in Manycore Processor

In a multicore processor system, one particular data block is accessed by multiple cores. This action raises a great concern about the validity and consistency of the data, as more than one processor tries to modify the data. All the private level-1 caches always need to preserve the most up-to-date data. Creating synchronization with the update of the data between cores requires a protocol among cores. Such a protocol keeps on checking the status if any data is modified within any local cache, or shared among cores, or gets invalidated. In our simulator, we designed three different coherence protocol for manycore system.

3.3.2.1 MESI Cache Coherence Protocol with L1 and L2 Cache Memory. In a manycore processor system, one particular data block is accessed by multiple cores. This action raises a great concern about the validity and consistency of the data, as more than one

processor tries to modify the data. All the private level-1 caches always preserve the most up-to-date data. Creating synchronization with the update of the data among cores requires a protocol that needs to be followed. Such a protocol keeps on checking the status if any data is modified within a local cache, or shared among cores, or gets invalidated. Figure 3.8 shows one of the most popular coherence protocols, MESI [29].

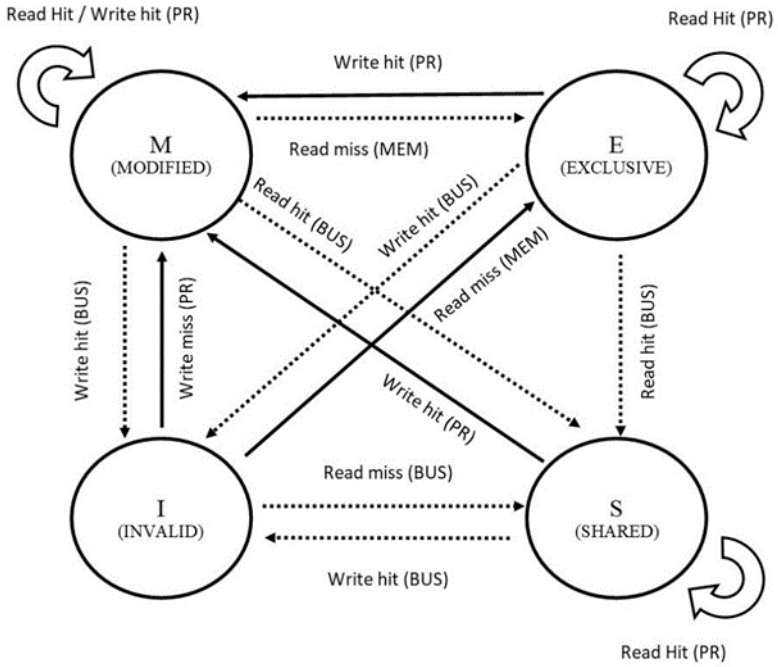

Figure 3.8: Four states of MESI cache coherence protocol

MESI defines four different states for a cache line: 1) Modified (M): when block has been modified recently and hold the data different from the main memory, 2)Exclusive (E): when any cache block has the only valid copy of a data and up-to-date with main memory, 3) Shared (S): the data cache is shared among multiple cores' cache memory; and 4) Invalid (I): the cache block is no longer valid due to a recent update to one of its copy/copies.

The transition between the states follows strict rules for keeping data valid for future use. In Figure 3.8, ‘PR’ refers to a private or local cache call, while ‘BUS’ means some other core's cache has been called. ‘MEM’ indicates reading from memory during a read miss. Here, we are trying to explain the protocol with two levels of cache that is designed in the simulator. One more thing to remember: both of the cache levels (L1 and L2 cache) are ‘Private’ as shown in Figure 3.2, and MESI coherence protocol is applied in the interconnection bus. All the data replacement, such as: 1) between L1 and L1 caches, and 2) between L1 and L2 caches, 3) memory to L1/L2 caches, follows the steps described in Figure 3.6. We also use one more binary control bit, named, ‘valid bit’. If the valid bit is 1, that means the simulator is allowed to access its data. But if the valid bit is ‘0’, that means the cache block is invalid and no longer permitted to be used. An invalid cache block can be used after replacing it with new data and setting the valid bit as ‘1’. In Figure 3.9, we will focus on explaining the logical operation and the transition of states in different situations for the MESI coherence protocol. Figure 3.9 goes as follows:

a) *Hit in L1 private cache*:

- If the instruction type is load/read, the cache block status remains unchanged.
- If the instruction type is store/write, new data will be written to the cache block and the new state will be changed to ‘Modified’. If the previous state was shared, the simulator will send out an invalidation signal to the bus. The invalidation signal causes the matched cache block valid bit to be set to 0. The corresponding copy of L2 cache needs to be updated accordingly.

b) Miss in L1 private cache but Hit in L2 private cache:

- For load operation, if a cache miss in L1 is followed by a hit in L2, new data must be placed in L1 from L2. In this case, the L1 cache block's state is set to same as the state of L2 cache block.
- For store operation, the desired tag will be copied to L1 from L2. Then the processor will perform write operation in L1 and set its status to 'Modified'. The corresponding copy of L2 is updated too. After write operation an invalidation signal is sent to Bus to invalidate copies.

c) *Miss In L1 & L2 private caches and Hit in Bus*:

- For load operation, if the cache is a miss in the private L1 & L2 cache but hit in the Bus (other core's L2 caches), the simulator will bring the data to the private core's (requester) L1 & L2 cache. If the hit cache block's state is 'Modified', it needs to update lower memory first. After the update, states of the hit cache block and requester cache block in the L1 & L2 becomes 'Shared'.

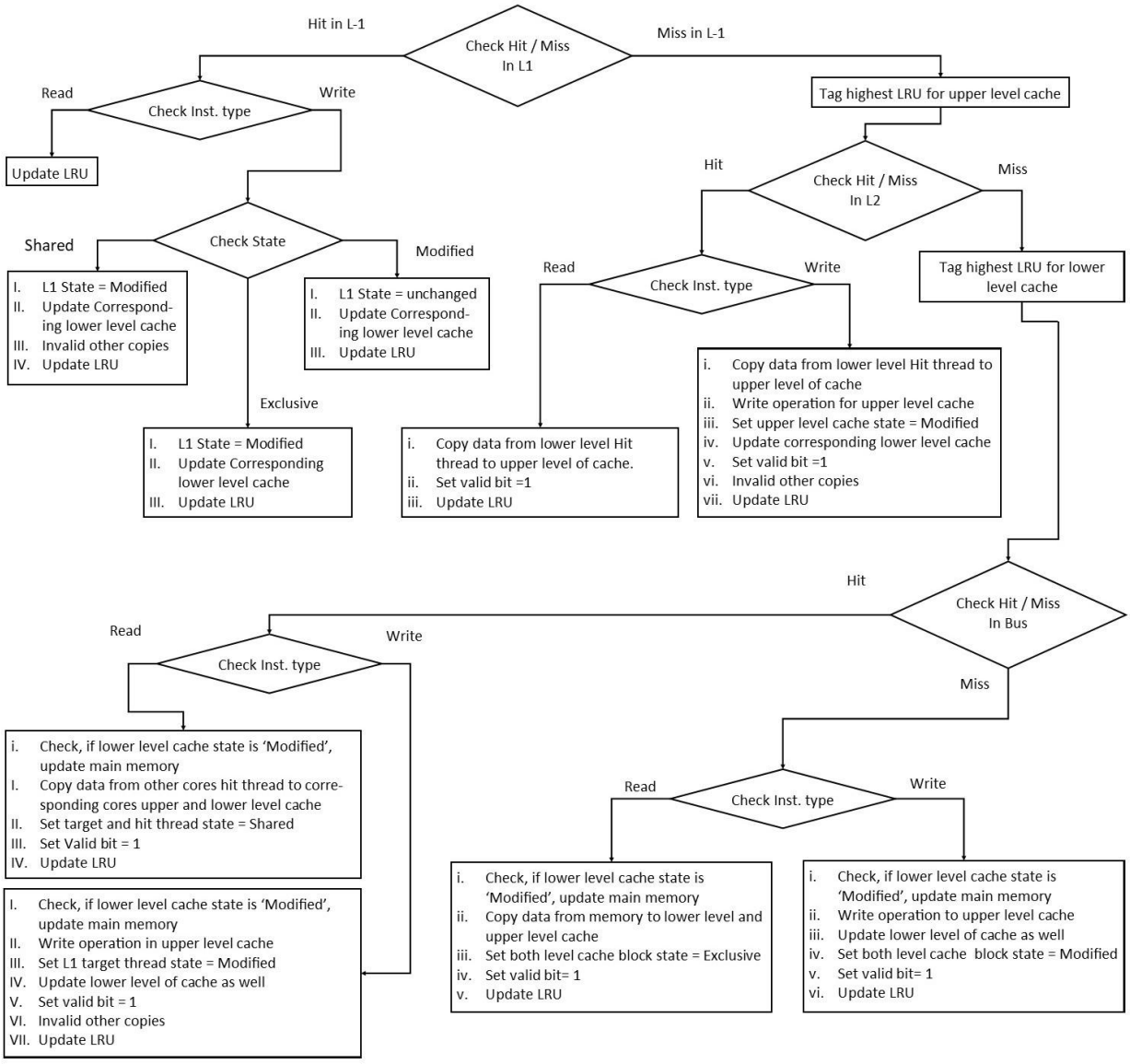

Figure 3.9: MESI protocol for two levels of cache

- For store operation, it places the new data in the L1 private cache and set state as 'Modified'. After that, the L2 cache is updated accordingly. As there are other copies found in the Bus, an invalid signal is sent to void those cache blocks.

d) *Miss in L1 and L2 private caches and in Bus*:

- For load operation, memory provides the desired data to both the L1 and L2 caches. If the L2 cache block 'Modified', it needs to update main memory first. L1 and L2 cache block state becomes 'Exclusive' after receiving data from memory.
- For store operation, the processor writes new data to L1 and sets its state as 'Modified'; then update L2 accordingly.

Each of the steps mentioned above are completed by the 'UPDATE LRU' function. When any L1 cache block is updated to 'Modified' state, its correspondent L2 cache block becomes 'Modified' as well, since both cache levels need to be in sync. For the same reason, when an invalidation signal is sent to the Bus, it applies for every associated L1 & L2 caches.

3.3.2.2 SNOOPY Cache Coherence Protocol with L1 and L2 Cache Memory. SNOOPY or MSI [29] cache coherence protocol is much like MESI protocol. It has three transition states: Modified, Shared and Invalid. The only difference with the MESI protocol is that it doesn't have the 'Exclusive' state. The transition of states in the Snoopy protocol is shown in Figure 3.10.

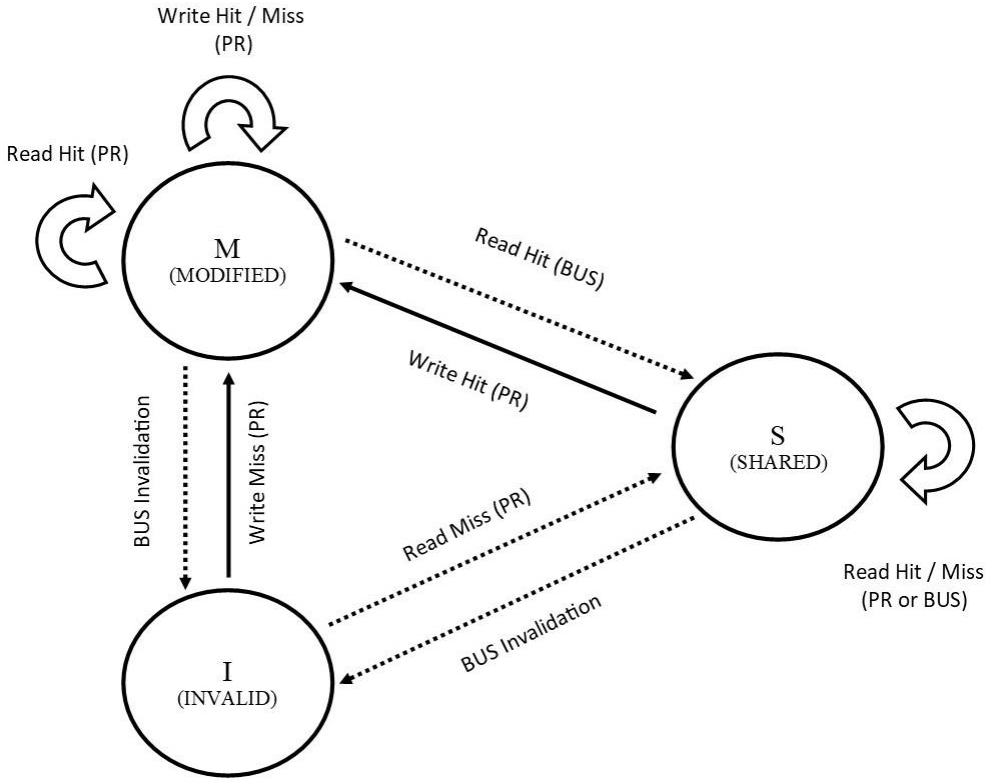

Figure 3.10: Three states of SNOOPY cache coherence protocol

The transition among the states is explained as follows:

a) Modified state:

- For Read hit (PR) and Write hit/miss (PR), current state remains same.
- For Read hit (BUS), state is changed to ‘Shared’.
- For a Write hit in Bus, current state will be ‘Invalid’

b) Shared state:

- For Read hit/miss (PR/BUS), state will remain same.
- In occurrence of a Bus invalidation signal is , new state will be Invalid.
- Write hit (PR) will change the state to Modified.

c) Invalid State:

- For Read Miss (PR), state will change to Shared.
- For Write Miss (PR), state will change to Modified.

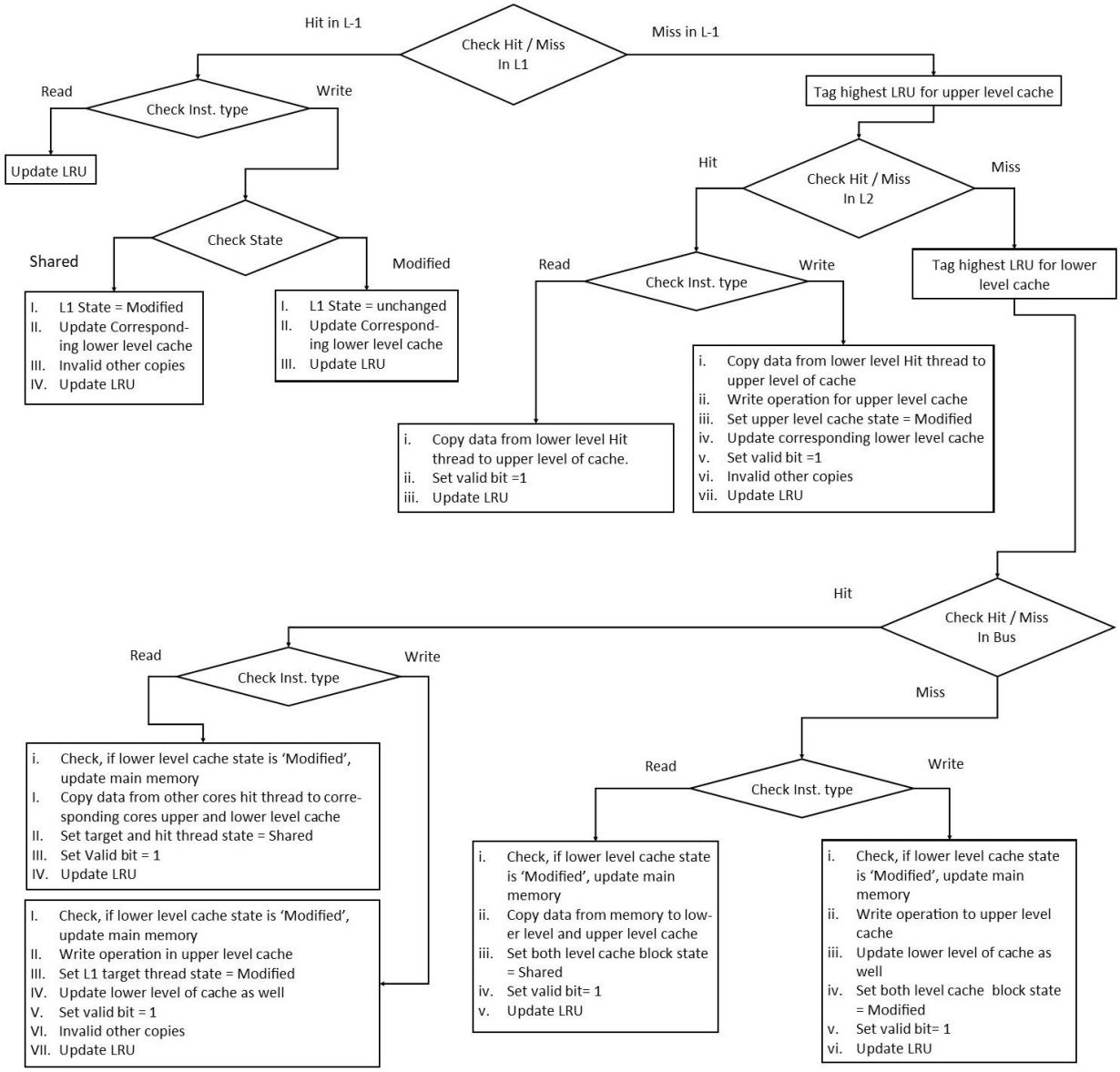

Figure 3.11: SNOOPY protocol for two levels of cache

The flowchart in Figure 3.11 summarizes the Snoopy protocol for L1 and L2 cache memory for manycore systems. As we can see, most parts of the Snoopy protocol are similar to the MESI

protocol (Figure 3.9). To avoid repetition and simplify understanding of the Snoopy protocol, we only discussed its differences with the MESI protocol.

- MESI protocol's 'Shared' and 'Exclusive' states are combined in the Snoopy protocol as 'Shared' state. All the caching events with 'Exclusive' state and 'Shared' state in MESI protocol are being merged into the 'Shared' state in Snoopy protocol.
- In event of Cache Read Miss in both private cache and in BUS, after getting data from main memory, current state will be changed to 'Shared'. In MESI protocol, this event changes the state to 'Exclusive' state (Figure 3.9).

Like, MESI, every data replacement among L1 cache, L2 cache, and main memory, will follow step shown in Figure 3.6.

3.2.2.3 FIREFLY Cache Coherence Protocol with L1 and L2 Cache Memory. FIREFLY [33] cache coherence protocol consists of three different states: 1) Valid-Exclusive (V): when data is valid & only copy in a private cache, in-sync with lower memory. 2) Shared (S): when data is valid and has multiple copies. 3) Dirty (D): data have been modified recently and lower memory not updated accordingly.

'CopiesExist' (C) is a control bit that is used to characterize the states in the Firefly protocol. During caching, C=1 means there is already other copy/copies of a particular data block in the bus. When C = 0, the data block doesn't have any copies [31].

During cache hit or miss in the Firefly protocol, the state of a particular cache block changes as follows (Figure 3.12):

- Read Hit (PR): State remains unchanged regardless of the value of ‘C’.
- Read Miss (PR): If C=0, new data is copied from lower memory and set state as ‘Valid’. If copy/copies found in other caches (C=1), state becomes ‘Shared’
- Write Hit (PR): If the current state is ‘Dirty’ or ‘Valid’ & (C=0), new state will be ‘Dirty’ after write operation. If current state shared and C=1, new state will be ‘shared’. But when state is shared and C=0, new state will be ‘valid’
- Write Miss: When C=1, cache state becomes ‘Shared’. If C=0, after new data is written, state is set as ‘Dirty’.
- Read & Write request (BUS): If the cache block is in Valid state, Bus Read or Write request changes the state to ‘Shared’. If the cache block is in Dirty state, Bus Read or Write request sets the state as ‘Shared’. If the block’s state is ‘Shared’, after Bus Read or Write request, state remains the same.

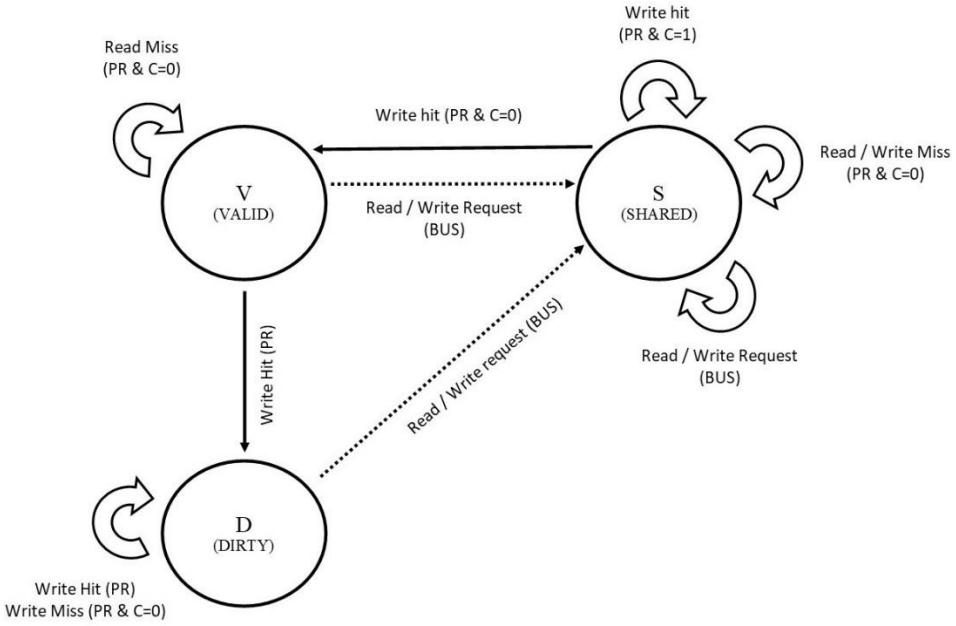

Figure 3.12: Transition diagram of FIREFLY protocol

The Figure 3.13 shows the flowchart for FIREFLY protocol for two levels of cache memory.

a) Hit in L1 private cache:

- If the instruction type is load/read, the cache block status remains unchanged.
- If the instruction type is store/write and the cache block state is 'Dirty' or 'Valid', new data will be written to the cache block and the new state will be 'Dirty'. If the state was 'Shared', the simulator will check the presence of other copies (C) in the bus. When C=1, new state will be 'Shared'; otherwise new state will be 'Dirty'. The corresponding copy of the L2 cache needs to be updated accordingly.

b) Miss in L1 private cache but Hit in L2 private cache:

- For a load operation, if a cache miss in L1 is followed by a hit in L2, new data must be placed in L1 from L2. In this case, the L1 cache block's state is set to the same as the state of the L2 cache block.

- For store operation, the desired tag will be copied to the L1 cache from the L2 cache. The rest of the process will follow the step (b). Write operation will put new data in L1 and L2 cache. If other copies are found in the bus, new state will be ‘Shared’. In case, no copy is found, new state will be ‘Dirty’.

c) Miss In L1 & L2 private caches and Hit in Bus:

- For load operation, if the cache is a miss in the private L1 and L2 cache but hit in the bus (other core’s L1/L2 caches), the simulator will bring that data to the private core’s (requester's) L1 and L2 cache. If other cores’ hit thread is ‘Dirty’, update main memory first. After that, the states of the hit cache block and requester cache block in the L1 and L2 become ‘Shared’.

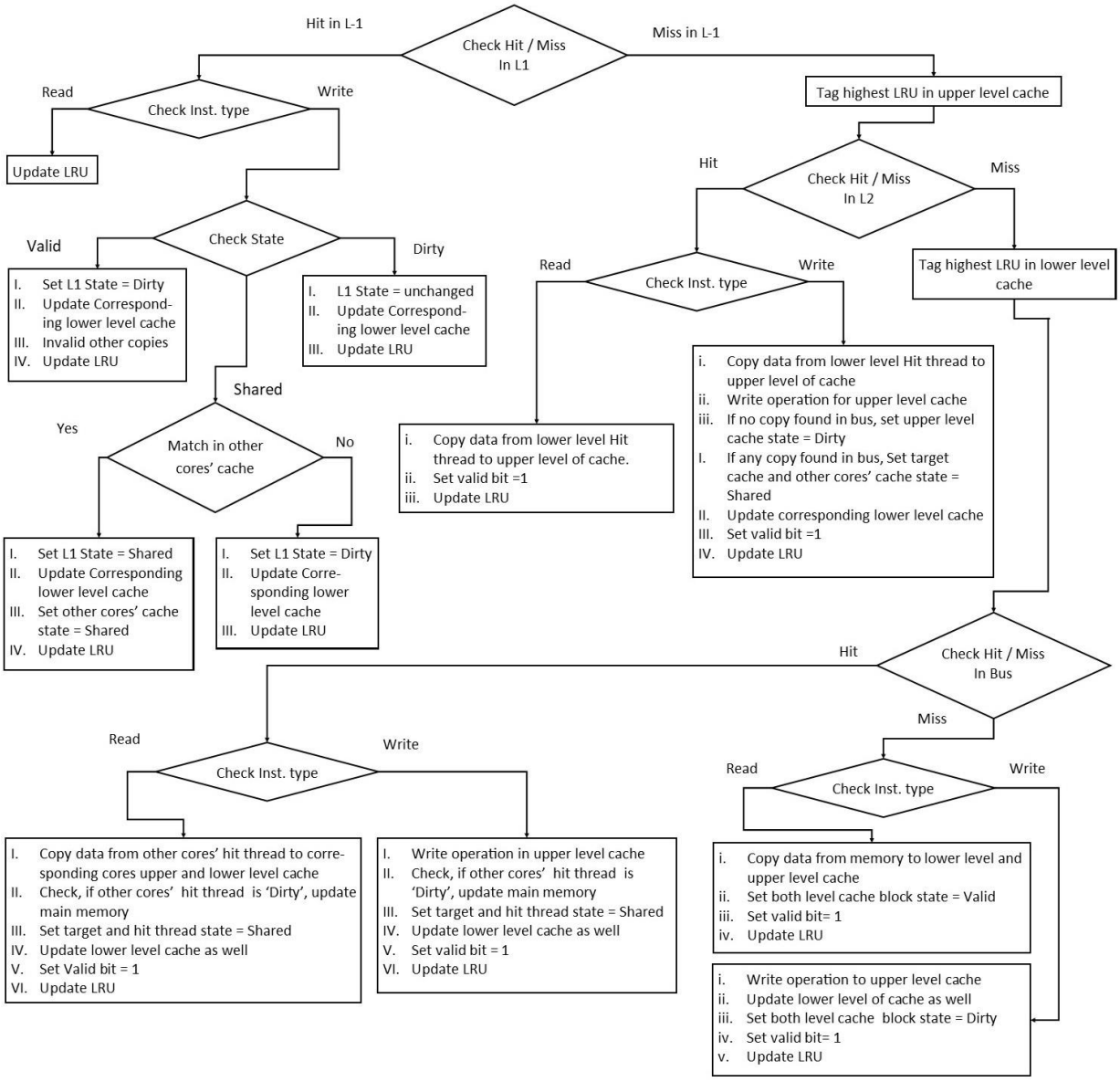

Figure 3.13: FIREFLY protocol for two levels of cache

- For store operation, it places the new data in the L1 and L2 private cache. If other cores' hit thread is 'Dirty', update main memory first. After that, the states of the hit cache block and requester cache block in the L1 and L2 become 'Shared'.

d) Miss in L1 and L2 private caches and in Bus:

- For load operation, memory provides the desired data to both the L1 and L2 caches. L1 and L2 cache block states become 'Valid' after receiving data from memory.
- For store operation, the processor writes new data to L1 and sets its state to 'Dirty'; then update L2 accordingly.

All the steps end with updating corresponding LRU.

3.4 Generating Input Trace Data using Pintool

One of the crucial steps for the simulator is to use the trace files of a benchmark program as input, generated from Pin [9]. Pin is popular in the computer architecture research area for its versatile features. It offers a large number of binary tools to attach and instrument any program for tracing and analyzing purposes. Tools like 'Pinatrace' can be utilized to generate the desired trace file, by attaching it with any benchmark (like Parsec [26] or Splash2 [27]) or real-world programs. When we have the trace file ready, we can feed it into the simulator through the command line for the simulation procedure. A detailed trace generation procedure is discussed in section 4.

3.5 Simulation Outputs

We can expect a number of attributes for the result section after the simulation process. Some of the common attributes are going to be hit rate, miss rate, percentages of hit rate and miss rate in each level of cache, bus traffic, etc. (Figure 3.14). The source code of the simulator will allow users to add more attributes based on their interests.

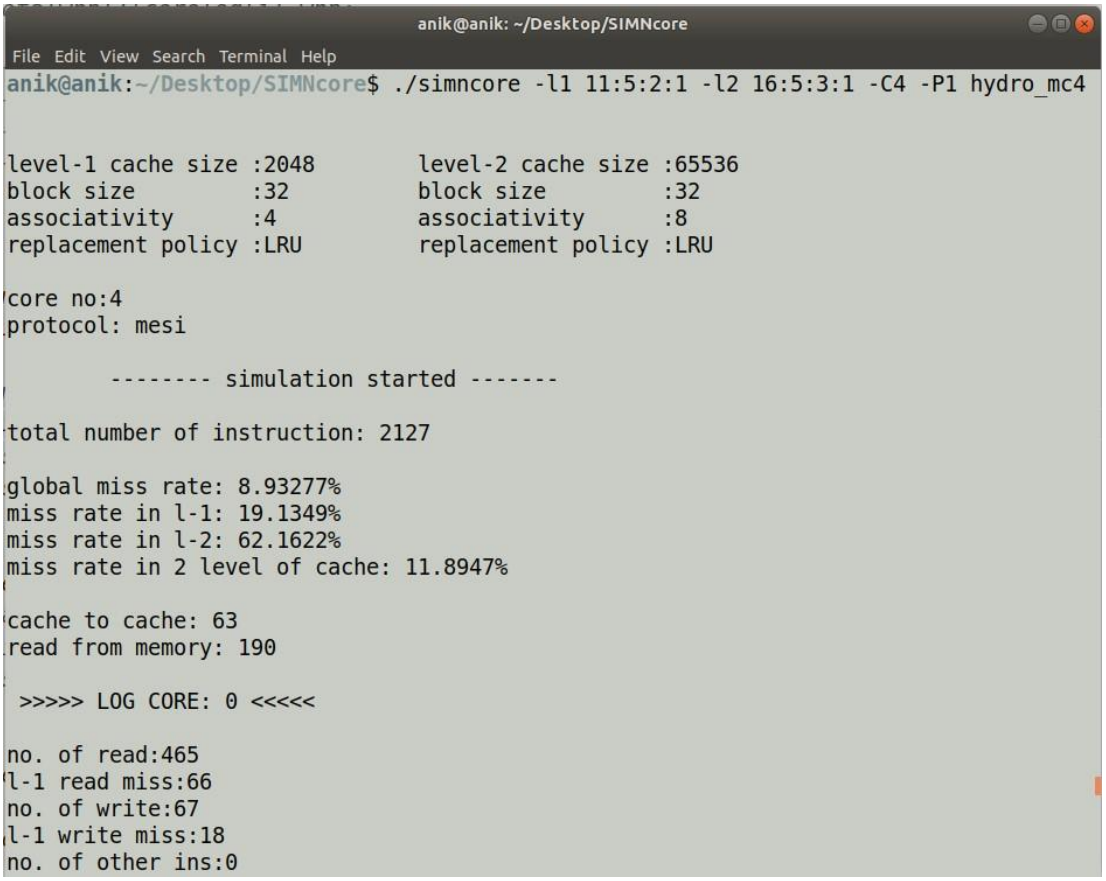

Figure 3.14: Result generated by SIMNCORE

We have used equation 3.3 to calculate miss rate in each level of cache memory. The cache miss rates of several benchmark programs have been evaluated from two viewpoints. Miss rates in Private Cache (Private Miss Rate) are observed from each core's L1 and L2 caches' perspective (Figure 3.15). Global miss rate refers to miss rate considering all the caches in a manycore system. We can also calculate Averagre Memory Access Time for L1 cache with formula 3.4 (only for L1 cache) and Averagre Memory Access Time for L1 and L2 cache with formula 3.5 [29]. Here, 1) Hit Time: time to take data from the cache to the processor, 2) Miss Rate: represented as a percentage over a time period, total cache misses divided by total memory requests, and 3) Miss

penalty: the delay brought on by a cache miss is referred to as a miss penalty. Miss penalty represents the additional time a cache takes to retrieve data from its memory. Each level of cache has different Hit Time, Miss rates, and Miss penalty. Under various cache configurations and workloads, cache miss rates typically vary; while Hit Time and Miss Penalty are intrinsic and static attributes associated with the chip fabrication technologies.

$$Cache\ miss\ rates = \frac{Total\ Cache\ Misses}{Total\ Cache\ Accesses} \quad \text{.....3.3)}$$

$$AMAT_{L1} = Hit\ Time_{L1} + Miss\ Rate_{L1}\ x\ Miss\ Penalty_{L1} \quad \text{.....3.4)}$$

$$AMAT_{(L1\&L2)} = Hit\ Time_{L1} + Miss\ Rate_{L1} \times (Hit\ Time_{L2} + Miss\ Rate_{L2} \times Miss\ Penalty_{L2}) \quad \text{..3.5)}$$

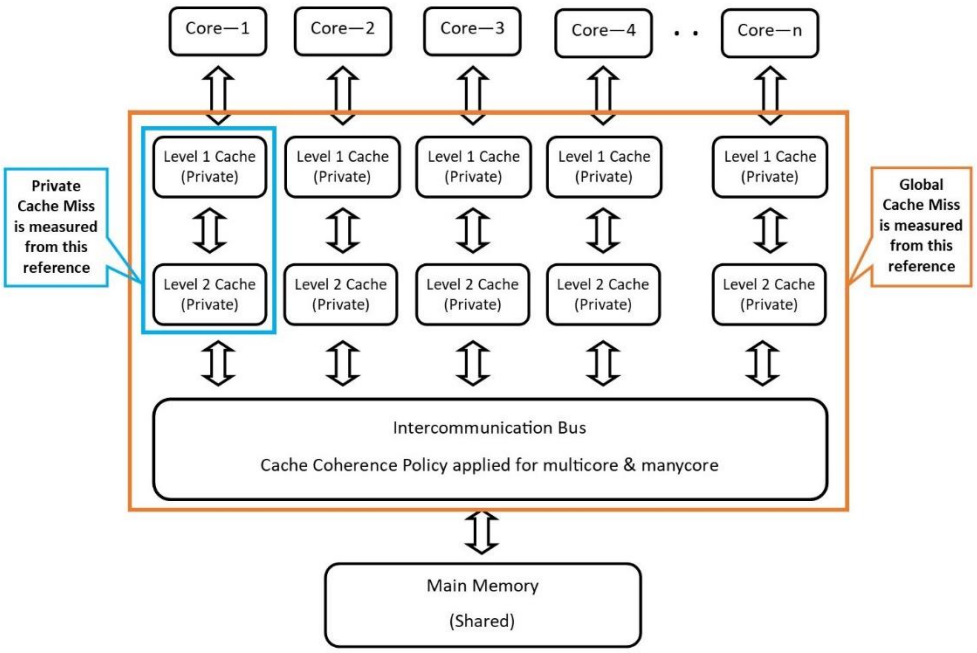

Figure 3.15: Two types of Cache Miss Rate

CHAPTER IV

SIMULATION METHODOLOGY

This section demonstrates the mechanism of the SIMNCORE simulator to measure multi-level cache performance**.** To start, we will be breaking down the process into some segments: 1) preparing trace file preparing benchmark/other programs with Intel Pin; 2) Setting up L1 and L2 cache size, block size and associativity, (for example, L1- 8 KB, block size 32 bytes with associativity 4 and L2- 256 KB, block size 64 bytes with associativity 8) and Number of cores (for instance, 8 or 16)**.** 4) Implementation cache coherence policy (like Single-core, MESI, and MSI); 4) Results and analysis**.** Figure 4.1 can portray the complete methodology in a nutshell.

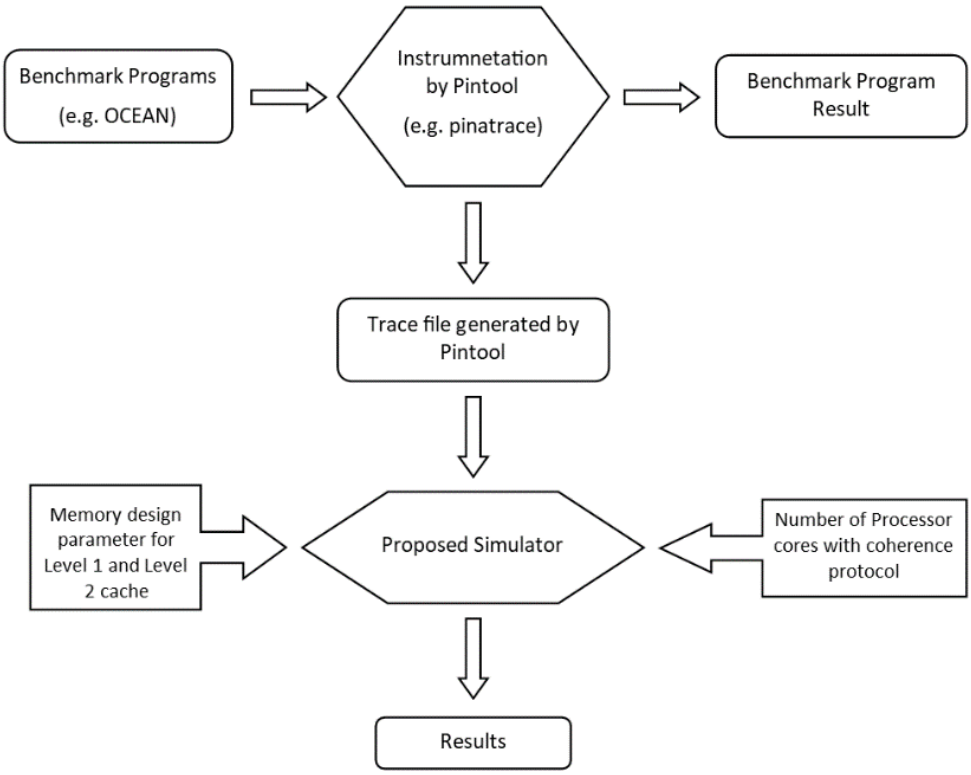

Figure 4.1: Simulation Methodology

4.1 Trace files of Benchmark programs with Pin instrumentation

Pin is a binary instrumentation tool by Intel. Pin instrumentation extracts runtime information or traces of a running program by inserting additional code. Almost every operating system supports Pin. The main advantage of Pin is that it can collect traces without affecting the original program at all. Pin provides tools like instruction count, memory read/write instruction, memory allocation, etc., which make it very nifty for memory-based experiments.

Benchmark programs are widely accepted as the workload for evaluating the exactness of any analytic framework. Like every other simulator, we need to test the behavior and outcomes of

our simulator using benchmark suites like SPLASH2, or Parsec. In Table 1, we provided some programs of different suites with memory references and instruction counts.

TABLE 4.1: List of Trace files for Splash2 and Parsec Benchmark programs

Benchmarks		Input parameters & Environment	Number of Memory References	Number of Instructions
Splash2	Barnes	16384 n-bodies	51.27 M	212.51 M
	FFT	64K complex doubles	23.17 M	59.30 M
	Fmm	256 particles, 2 cluster	3.67 M	17.13 M
	Ocean_cp	258 x 258 grid	168.0 M	407.6 M
	Ocean_ncp	258 x 258 grid	154.4 M	395.26
	Radiosity	17108 elemnts, 364 patches	568 M	2267.95 M
	Radix	radix 1024, 256K keys	24.26 M	52.99 M
	Raytrace	teapot.env	149.6 M	322.5 M
	Volrend	4x4 image block	28.68 M	60.78 M
	Water-nsquared	512 molecules	227.9 M	658.8 M
	Water-spatial	512 molecules	211.8 M	591.29 M
Parsec	Blackscholes	simsmall	101.1 M	234.6 M
	Bodytrack	simsmall	613.95 M	1429.6 M
	Canneal	simsmall	944.2 M	1832 M
	Facesim	simsmall	1032 M	3451 M
	Ferret	test	11.31 M	285.09
	Fluidanimate	test	26.84 M	60.03 M
	Freqmine	test	27.98 M	66.24 M
	Streamcluster	test	1.14 M	2.74 M
	Vips	test	89.22 M	154.57 M
	X264	test	0.73 M	1.61 M

Pin is equipped with a number of functions and modules that allow us to arrange traces according to our expectations. More information about pin for collecting traces for manycore processors will be added in the simulator manual. Here, a simple example of pin instrumentation is shown. 'Pinatrace' is a tool to collect the memory references like read and write instructions when any program is running through processors. The benchmark program used here is 'OCEAN (continuous particle)' with the necessary parameters. The dissection of this terminal command line is given in Figure 4.2.

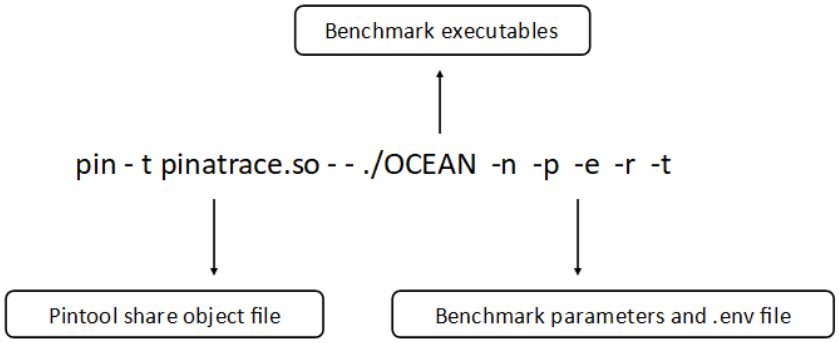

Figure 4.2: Pin instrumentation with associated parameters and environment

Figure 4.3 gives an example of a trace file in the required format for our simulator. Each trace file contains three separate columns. The first column represents a processor id. For example, an 8-core configuration has a processor id ranging from 0 to 7. A single-core system has only one processor id, which is '0'. The type of instruction, whether it is 'write (w)', 'read (r)', or other operation (z), is stored in the second column. The memory addresses for each instruction are given in the last column. The Simulator reads each row as an entry to decide which core is associated with the instruction, what type of instruction it is, and what the memory location for that instruction is.

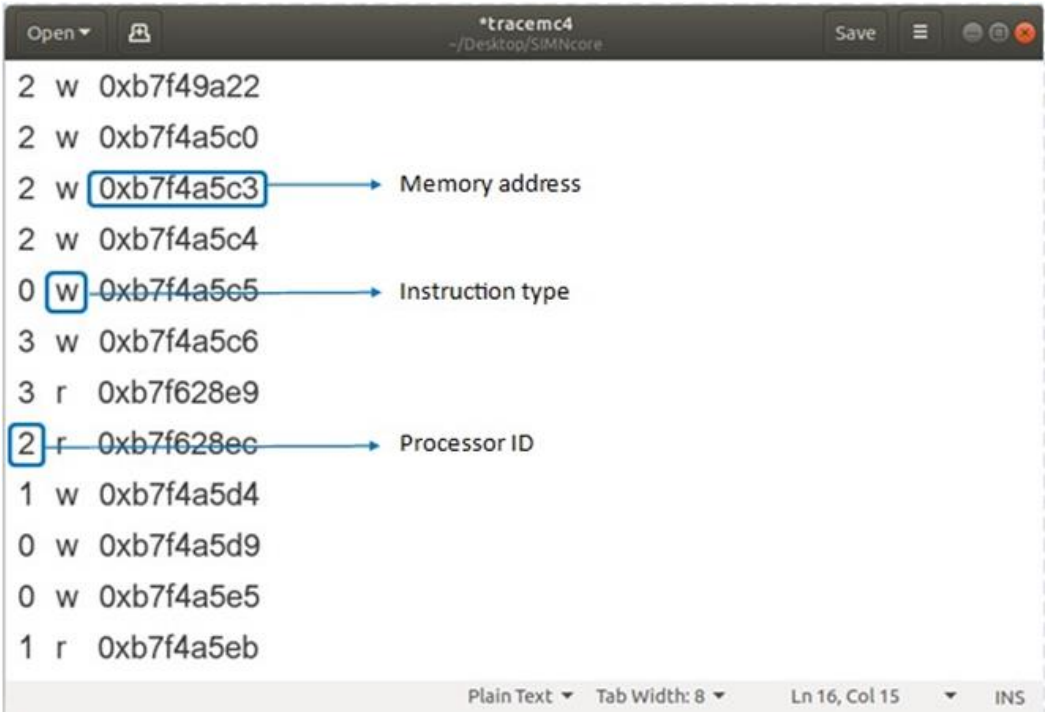

Figure 4.3: Trace file sample for Pintool ‘Pinatrace’

CHAPTER V

EXPERIMENTAL RESULTS

Our proposed SIMNCORE cache design simulator aims to enable the construction of various types of cache memory for general-purpose processing units. We have designed several cache hierarchies with L1 and L2 cache, and evaluated the performance of the designed cache memories using trace files collected from the benchmark programs. This section highlights the performance of our designed cache simulator. For the performance evaluation, trace data generated from Parsec, and Splash2 benchmark programs have been used (refer to Table 4.1).

Figure 5.1 shows the findings regarding the cache miss rates of the five programs (Ocean_cp, Radix, Raytrace, Water_nsq, & Blackscholes) with varying block sizes and cache sizes in L1. The L1 cache is varying from 2 KB to 64 KB in cache size, with two different block size, 32 bytes and 64 bytes, and 4-way associative. MESI coherence protocol is applied to this 8-core processor and L1 cache configuration. The results indicate that with the increase in block size (from 32 bytes to 64 bytes), the cache miss rate decreases proportionately. At the same time, lower cache size indents have a higher miss rate due to capacity misses [8, 35].

The effect of set associativity on cache performance is shown in Figure 5.2. For the five benchmark programs mentioned above, we compared the miss rates for L1 cache. The setup for this test: 1) 4 KB L1 cache with 32 bytes block size, 2) 8 cores and MESI coherence protocol. As it is observed, the miss rates are inversely proportional to associativity. For all the programs, the

direct-mapped cache has the highest miss rate, and the 8-way cache has the lowest, as similar outcomes was found in other accepted simulators too [8, 34].

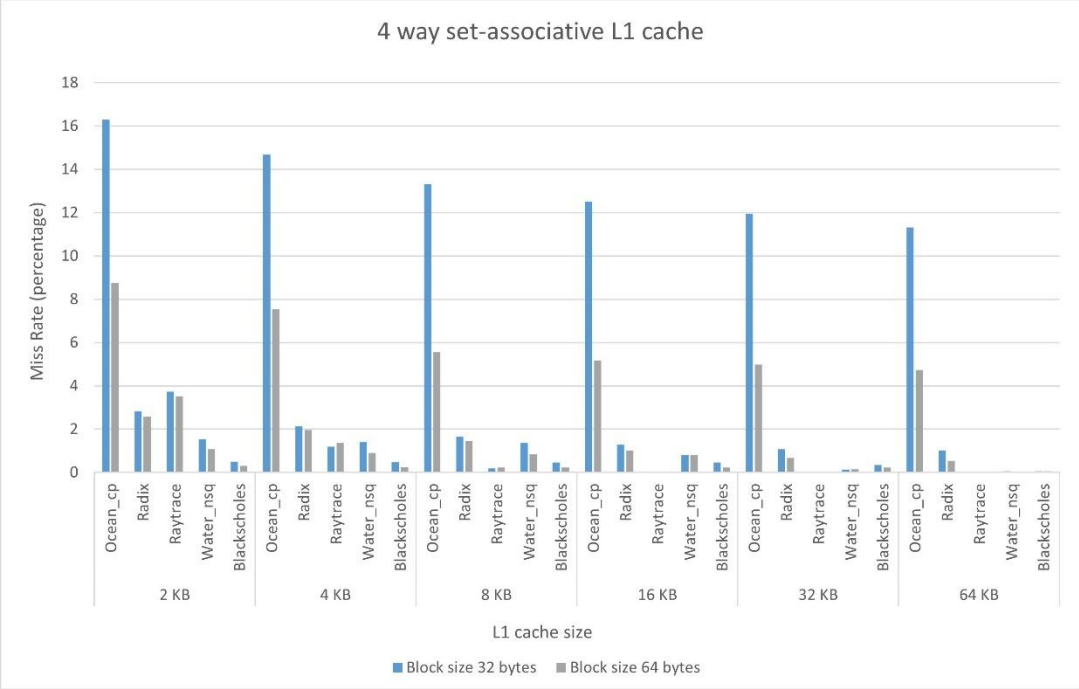

Figure 5.1: Miss rates of L1 cache (8-core, protocol: MESI)

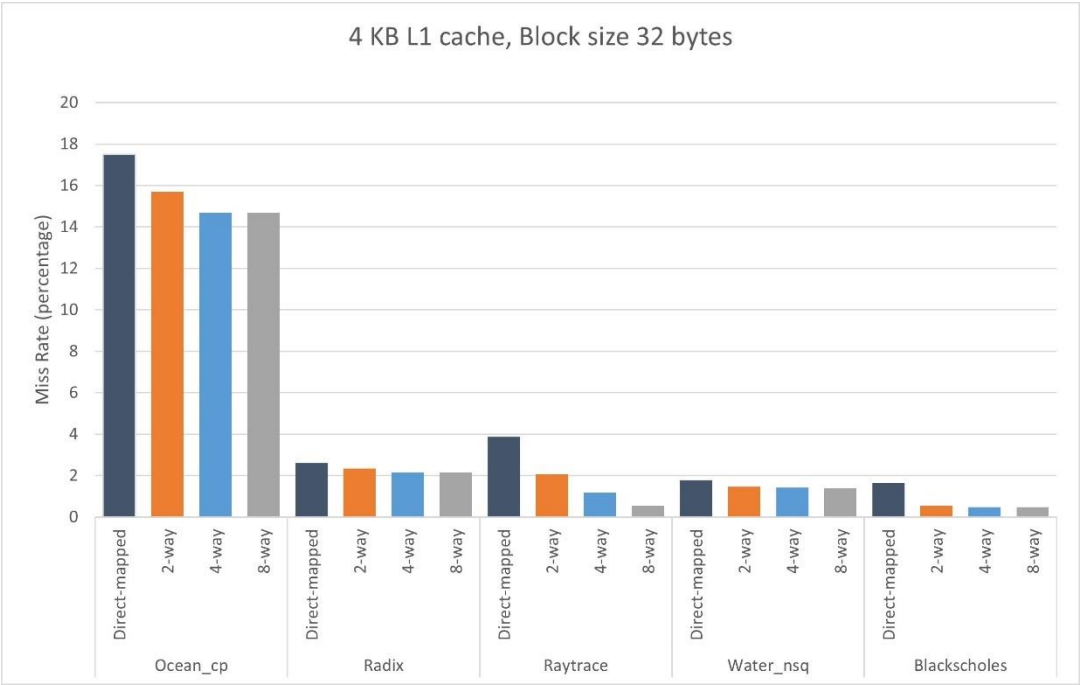

Figure 5.2: Miss rates of L1 cache with varying associativity (8-core, protocol: MESI)

Similar to SIMNCORE, FM-SIM is a multicore cache design simulator that runs simulations of cache memory via the command line. From the user's perspective, the operating principles of SIMNCORE and FM-SIM appear to be relatively similar, even though FM-SIM can only design and simulate level-1 cache. We showed two different sets of outputs obtained from SIMNCORE along with the outputs from FM-SIM (Figure 5.3). For each benchmark program, the first SIMNCORE evaluation follows the same cache parameters of the L1 cache, which has been considered for FM-SIM simulation. The second SIMNCORE evaluation gives the output of the L2 and L1 cache (for specified specifications) along with bus access information. For each case, SIMNCORE shows an almost equal hit rate found in FM-SIM, which indicates SIMNCORE performs appropriately [8, 34].

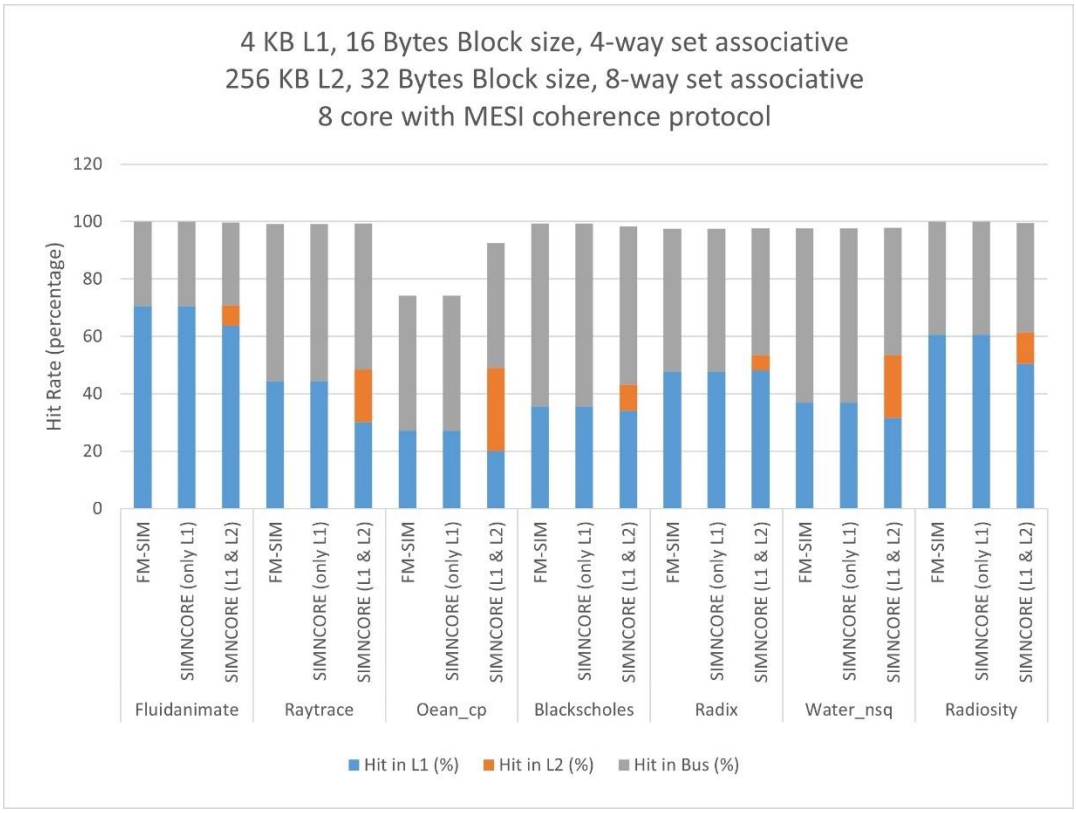

Figure 5.3: Cache hit rate analysis with FM-SIM and SIMNCORE

To evaluate L1 and L2 private cache performance, we tested out SIMNCORE against SMPCache, simulating the same workloads (Figure 5.4). SMPcache is a popular trace-driven simulator that is widely used in academia for symmetric multiprocessor studies. It provides users with a graphical user interface that makes navigating the caching events simple. However, users sometimes find it difficult to understand the precise formats needed to produce the trace file for multicore cache simulation with SMPCache. Additionally, unlike SIMNCORE, it takes a long time to simulate even modest trace files for the detailed graphical display of the simulation. We have collected several SPEC ’92 benchmark programs (Hydro, Nasa7, Cexp, Mdljd, Ear, Comp, Wave, Swm, and UComp) from the SMPcache library [36] and compared the outputs received from both the simulators. The cache configuration includes a 2 KB L1 cache with a block size of 32 bytes

and 4-way set-associativity, and a 64 KB L2 cache with a block size of 32 bytes and 8-way set-associativity for each core. The system is considered to have 4 cores with the MESI coherence policy. The Global Cache miss rate denotes the miss rate in the private caches which consists of the L1 and L2 cache. The cache miss rates for every benchmark program found in SIMNCORE and SMPcache are almost equal, which makes it safe to say that SIMNCORE's accuracy is acceptable like other recognized simulators.

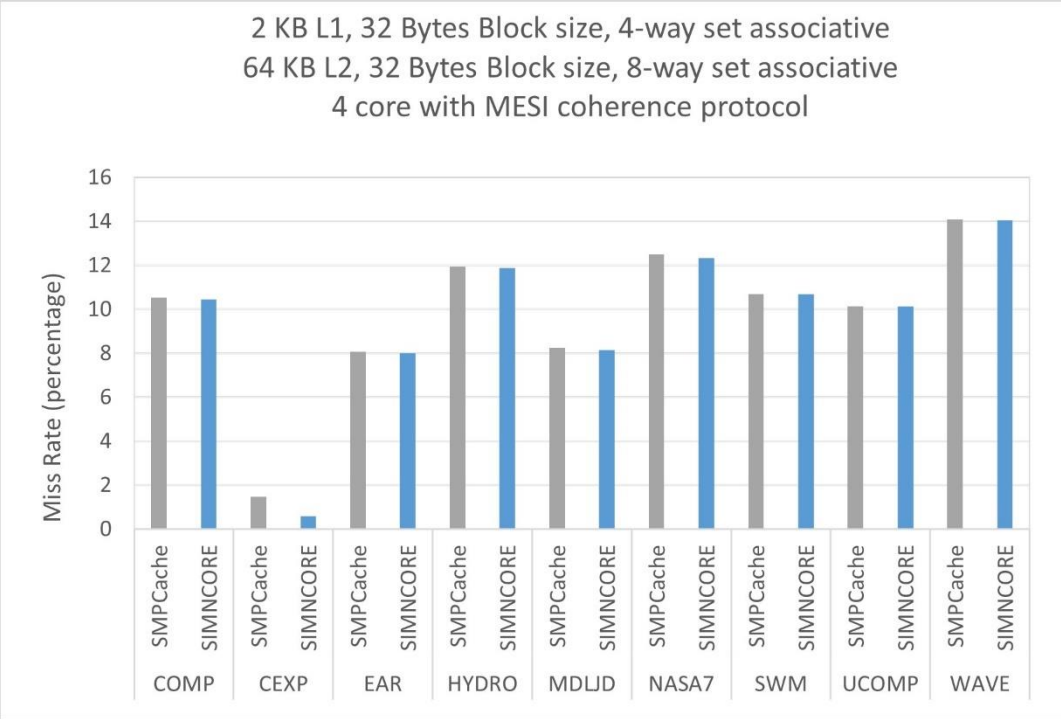

Figure 5.4: Global cache miss rate analysis in SMPCache and SIMNCORE

Figures 5.5 to 5.76 depict the experimental findings for the following multicore/manycore cache parameters: 1) varying number of cores (4-core and 8-core); 2) varying cache sizes for L1 cache (4 KB, 8 KB, 16 KB, 32 KB, and 64 KB); 3) varying cache sizes for L2 cache (128 KB, 256 KB, 512 KB, 1024 KB, and 2048 KB); 4) varying block sizes (32 bytes and 64 bytes); 5) varying coherence protocol (MESI and Firefly); 6) Set associativity (4-way and 8-way).

Figures 5.5 (5.5(a) to 5.5(u)) show the cache miss rates of different benchmark programs from two perspectives: 1) Miss rate in Private Cache, and 2) Miss rate globally. Figure 5.5(a) shows the private cache miss rate of four different programs with MESI and FRIREFLY cache coherence policies. Private cache miss rates for the benchmark programs are significantly higher for the MESI protocol compared to the FIREFLY protocol, as invalidation often occurs in caches after cache write updates occur in the MESI protocol. Figures 5.5(a) to 5.5(u) show the private cache miss rates of the benchmark programs under MESI and FIREFLY protocols with different L1 and L2 cache parameters. Usually, if the cache sizes are increased, cache miss rates decrease. But sometimes, with increasing cache sizes, we didn't observe any improvement in the cache miss rates (e.g., Figure 5.5(c)). In those cases, every L1 cache has a larger L2 cache (256 KB). At that moment, increasing L1 cache doesn't reflect a significant change in the private cache sizes (L1 and L2), thus not improving cache miss rates.

At times, increasing the core number increases the private cache miss rate but decreases the global cache miss rate. Considering Figure 5.5(b) with Figure 5.5(n), we can observe that 8-core caches have better private cache miss rates than 16-core caches. In contrast, analyzing Figures 5.6(b) and 5.6(n), we can see that the global cache miss rates are very low for 16-core cache memory compared to that of 8-core.

Again, due to the nature of the data (memory addresses) in the trace files, the cache hit or miss rate saturates at times. When similar data/addresses (temporal and spatial) are called by multiple cores simultaneously, a large number of invalidations occur at the private cache level and limit cache hit rates in private caches (e.g., Balckscholes in Figure 5.5(g)).

Global cache miss rates for different programs are shown in Figures 5.6 (5.6(a) to 5.6(u)). In Figure 5.6(a), the global cache miss rate of four different programs with MESI and FRIREFLY cache coherence policies is shown. In contrast with the private cache miss rate from Figure 5.5, global miss rates for each program are almost the same for two different coherence protocols. Though, in MESI protocol, when private caches encounters a lot of cache misses, later those data are found in the bus, improving overall hit rates of the cache memory.

Figures 5.7(a) to 5.7(m) show the Bus traffic for each cache configuration mentioned in the figures. Bus traffic represents the number of times cores access the bus for sending invalidation signals, accessing other cores' caches, updating states after write operations, and so on. Figure 5.7(a) shows the bus traffic of four different programs with MESI and FRIREFLY cache coherence protocols. Unlike MESI, the Firefly protocol doesn't have 'Invalid', which results in a lower number of accesses in the bus for updating states. With the MESI protocol, write updates and data invalidation signals often require access to the bus; which in turn creates higher bus traffic. The global miss rate can be close to zero at times, which is why it is not visible in the figure (e.g., Figure 5.6(c), for Blackscholes and Water_nsq.).

In Figure 5.8(a), the data transfer count among the caches of four different programs using the MESI and Firefly cache coherence protocols is shown. With the MESI protocol, due to higher cache miss rates in private caches, data is often sought in other cores' caches, which leads to higher data transfer rates among cores. Figures 5.8(a) to 5.8(m) show the data transfer count among caches

of the benchmark programs under the MESI and FIREFLY protocols with varying cache parameters.

Results from Figures 5.9(a) to 5.9(d) show the effect of the number of cores on the bus traffic for each benchmark program. For each case, with an increasing number of cores in the processor, we observe an increase in bus traffic (except Ocean_cp with a 16-core processor (Figure 5.9(c)). From these figures, we can see that 8-core caches have less BUS traffic than 16-core caches. The global miss rates in 16-core caches (Figure 5.6(o)) are much less than those in 8-core caches (Figure 5.6(c)), which indicates 16-core works better than 8-core.

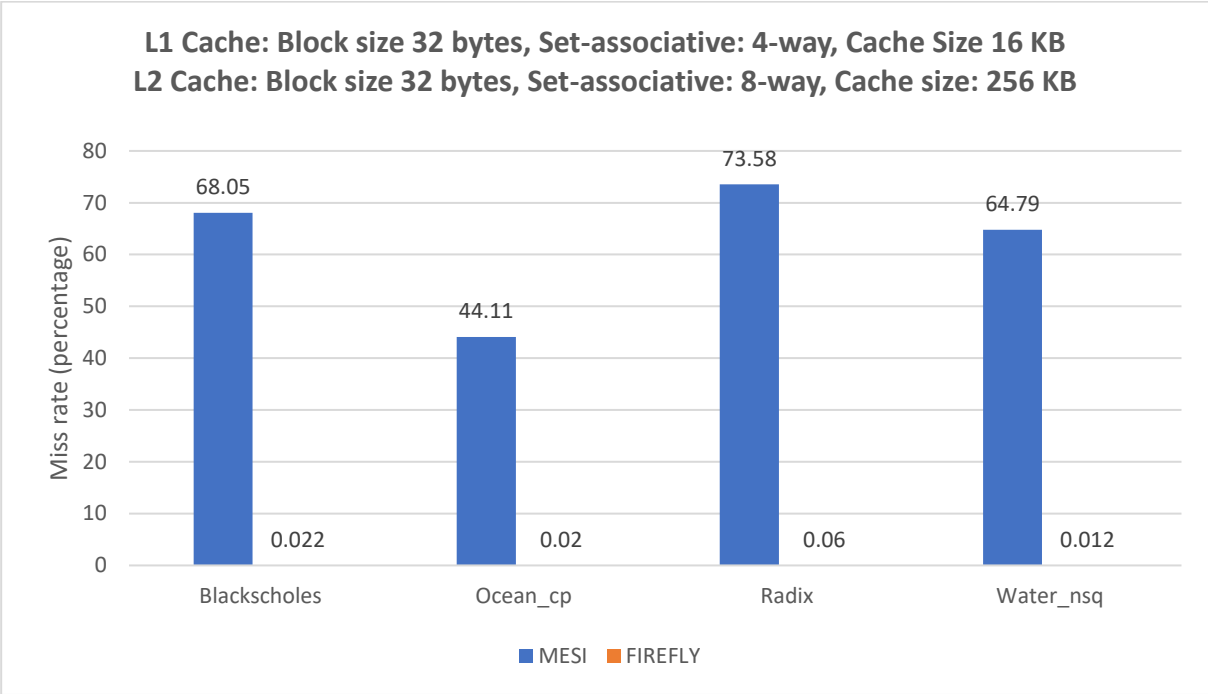

Figure 5.5(a): Comprison between MESI and FIREFLY Protocols regarding Private Cache Miss rates (16-core)

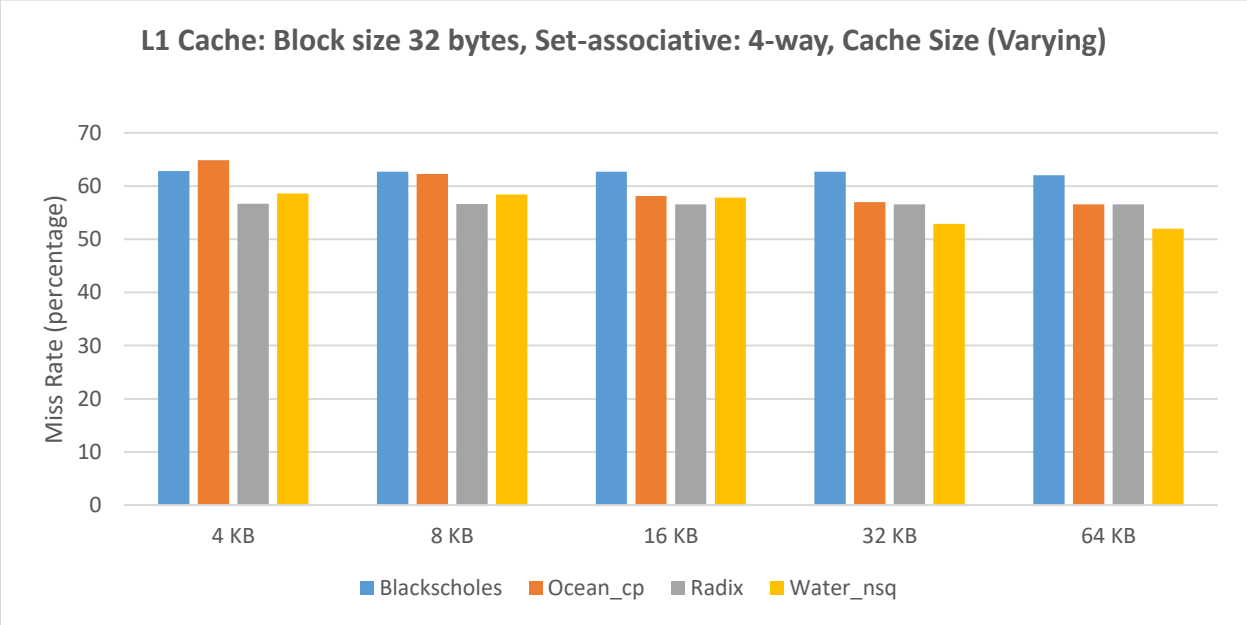

Figure 5. 5(b): Miss Rate in L1 Private Cache (Varying L1 cache, Block size 32 bytes, 8-core, protocol: MESI)

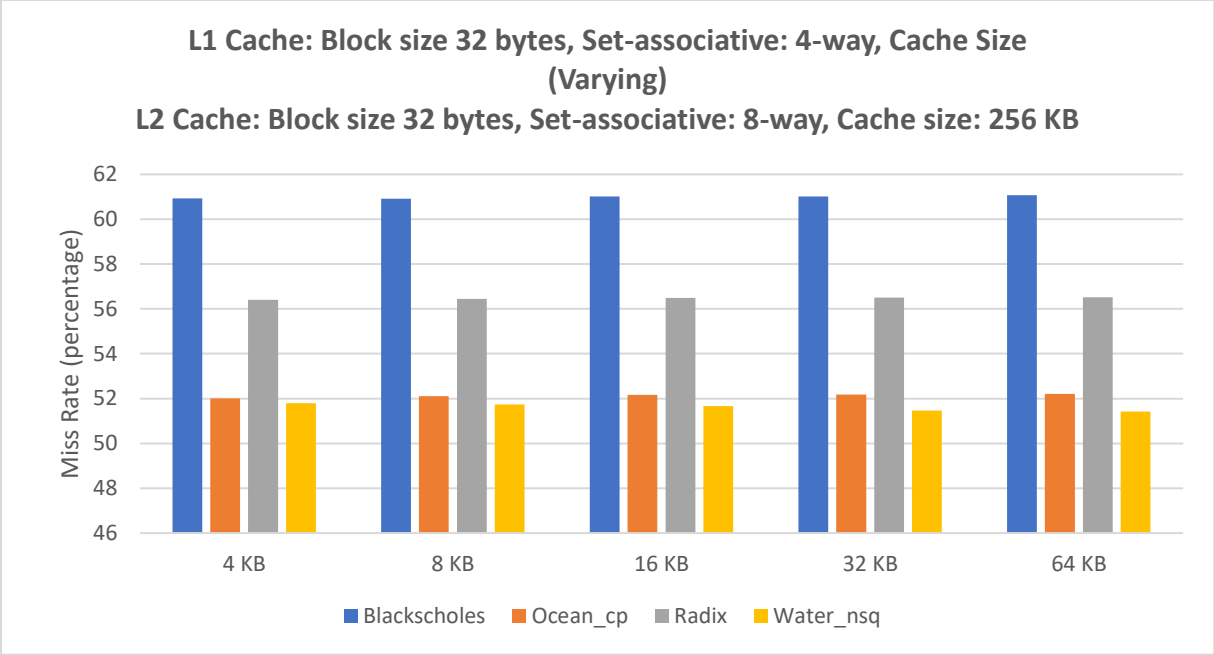

Figure 5. 5(c): Miss Rate in L1 & L2 Private Cache (Varying L1 cache, 256 KB L2 cache, Block size 32 bytes, 8-core, protocol: MESI)

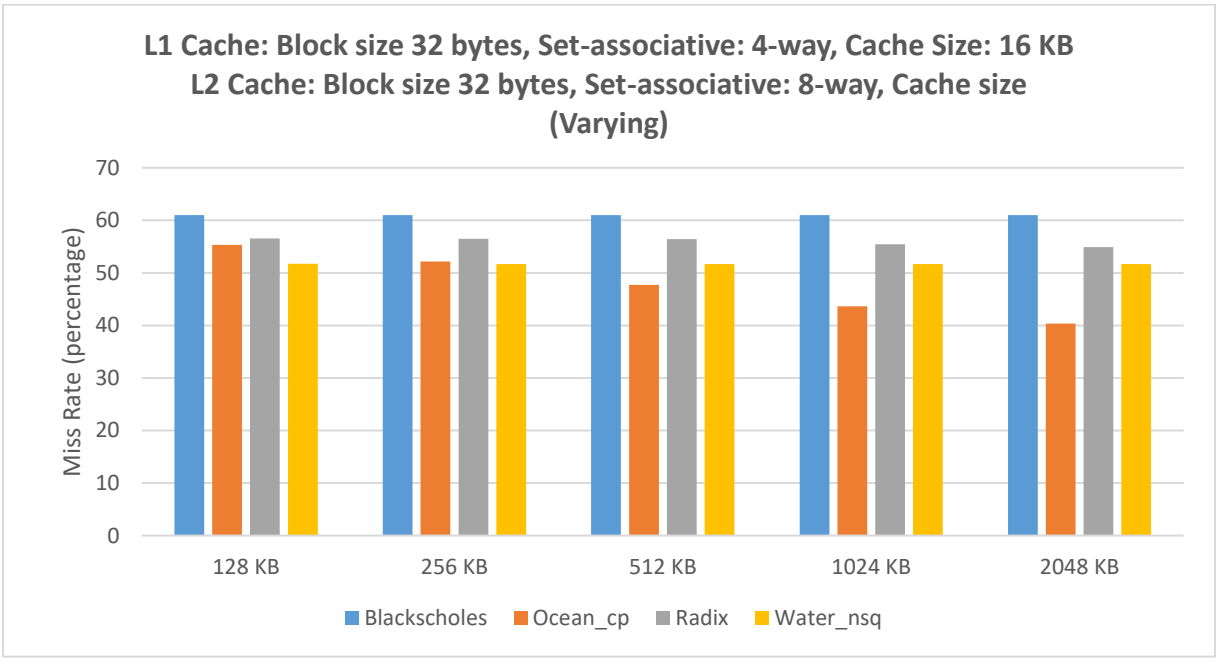

Figure 5. 5(d): Miss Rate in L1 & L2 Private Cache (16 KB L1 cache, varying L2 cache, Block size 32 bytes, 8-core, protocol: MESI)

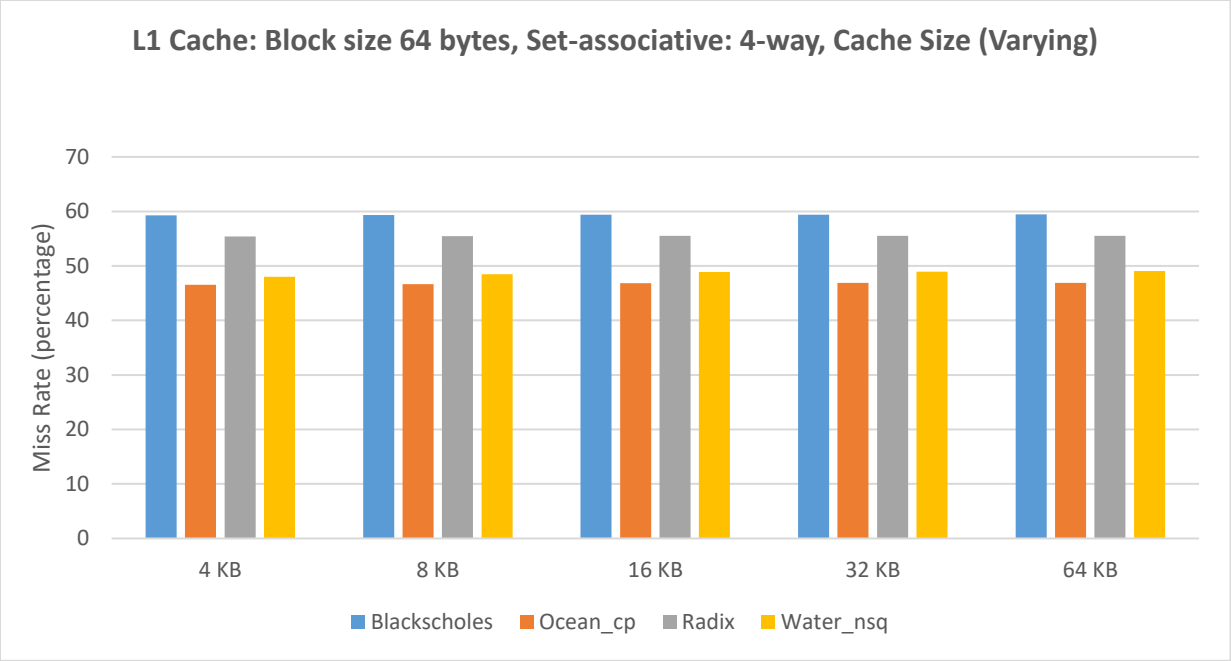

Figure 5. 5(e): Miss Rate in L1 Private Cache (Varying L1 cache, Block size 64 bytes, 8-core, protocol: MESI)

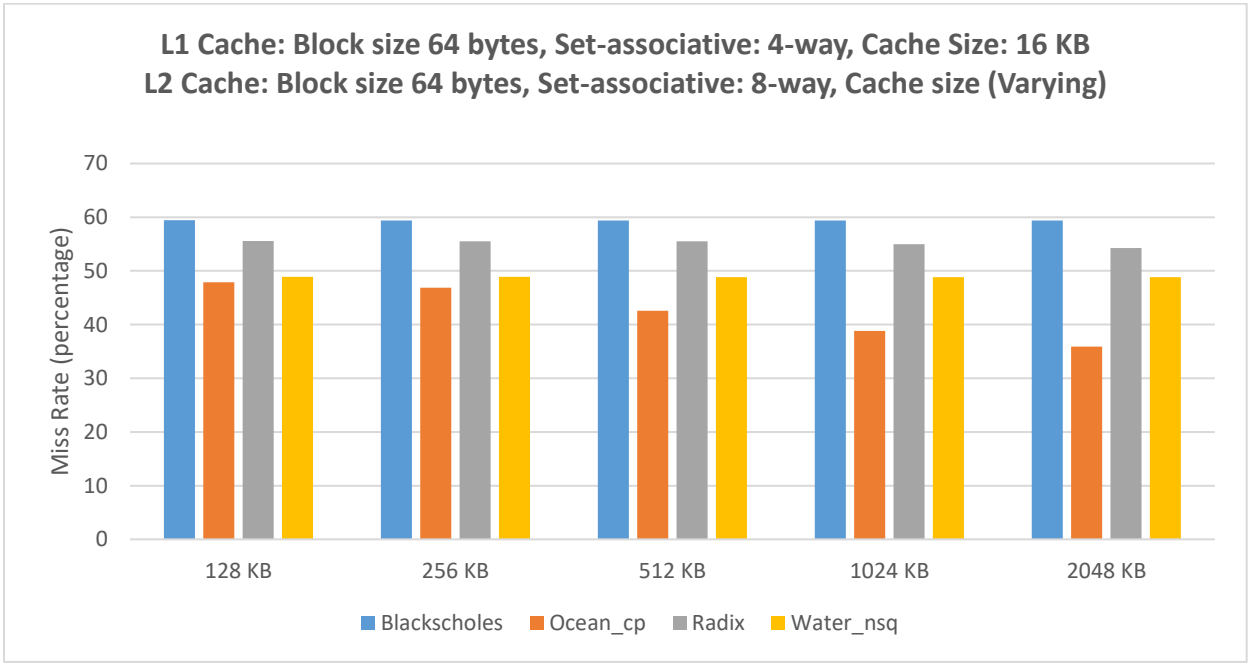

Figure 5. 5(f): Miss Rate in L1 & L2 Private Cache (Varying L1 cache, 256 KB L2 cache, Block size 64 bytes, 8-core, protocol: MESI)

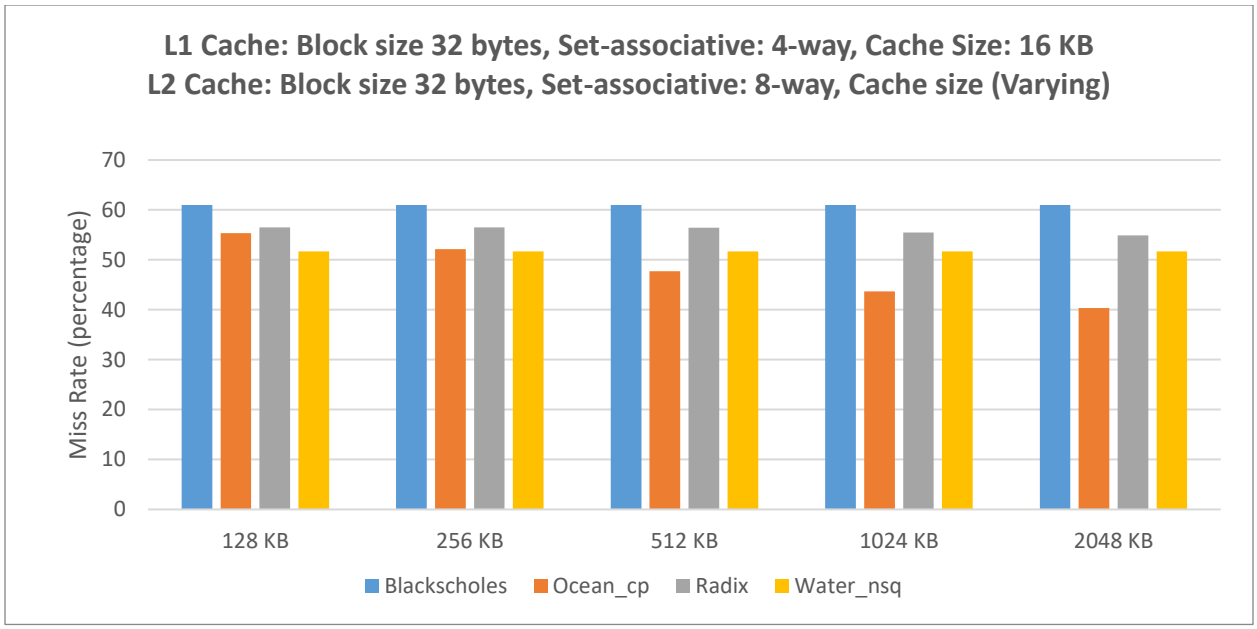

Figure 5. 5(g): Miss Rate in L1 & L2 Private Cache (16 KB L1 cache, varying L2 cache, Block size 64 bytes, 8-core, protocol: MESI)

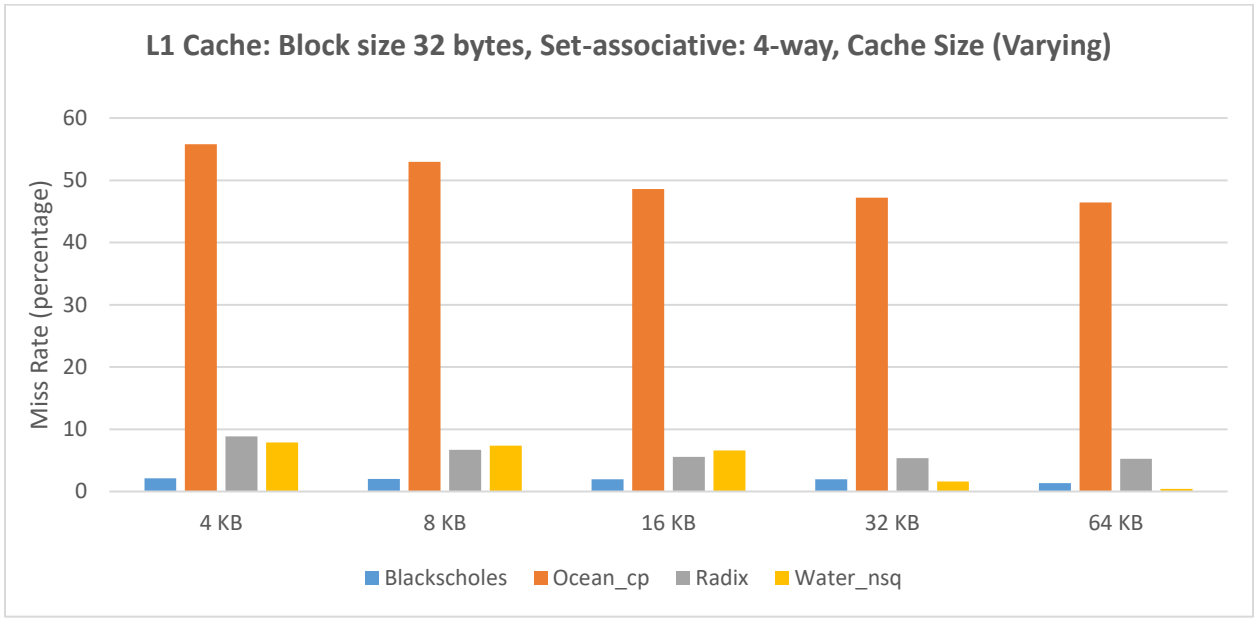

Figure 5. 5(h): Miss Rate in L1 Private Cache (Varying L1 cache, Block size 32 bytes, 8-core, protocol: FIREFLY)

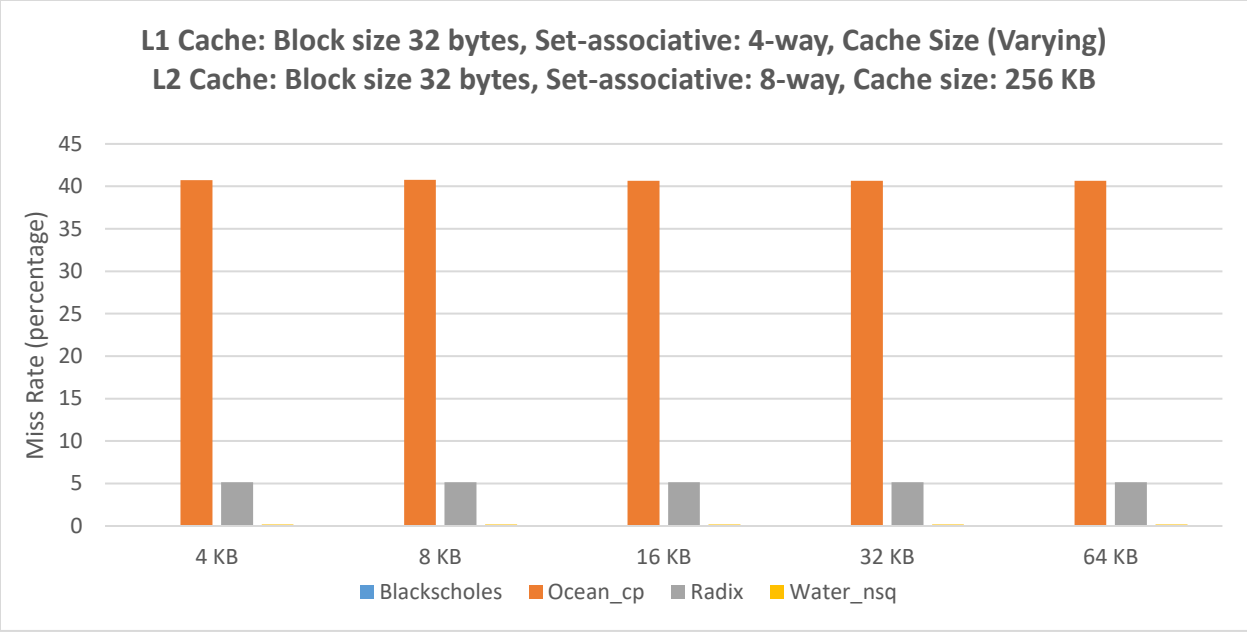

Figure 5. 5(i): Miss Rate in L1 & L2 Private Cache (Varying L1 cache, 256 KB L2 cache, Block size 32 bytes, 8-core, protocol: FIREFLY)

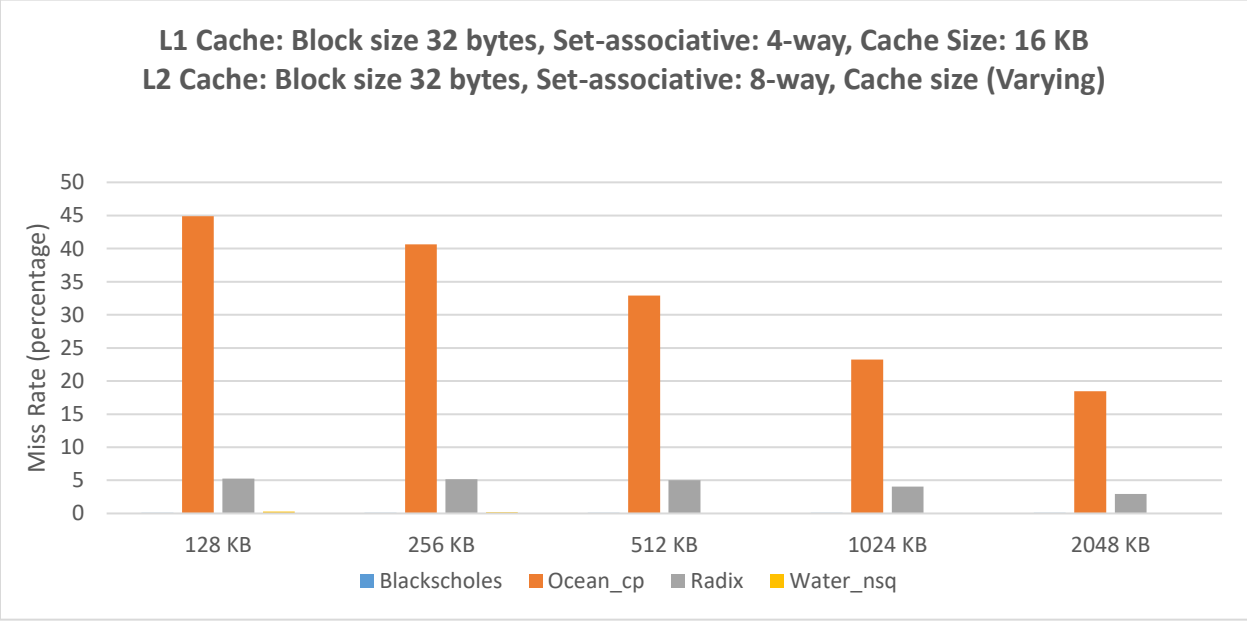

Figure 5. 5(j): Miss Rate in L1 & L2 Private Cache (16 KB L1 cache, varying L2 cache, Block size 32 bytes, 8-core, protocol: FIREFLY)

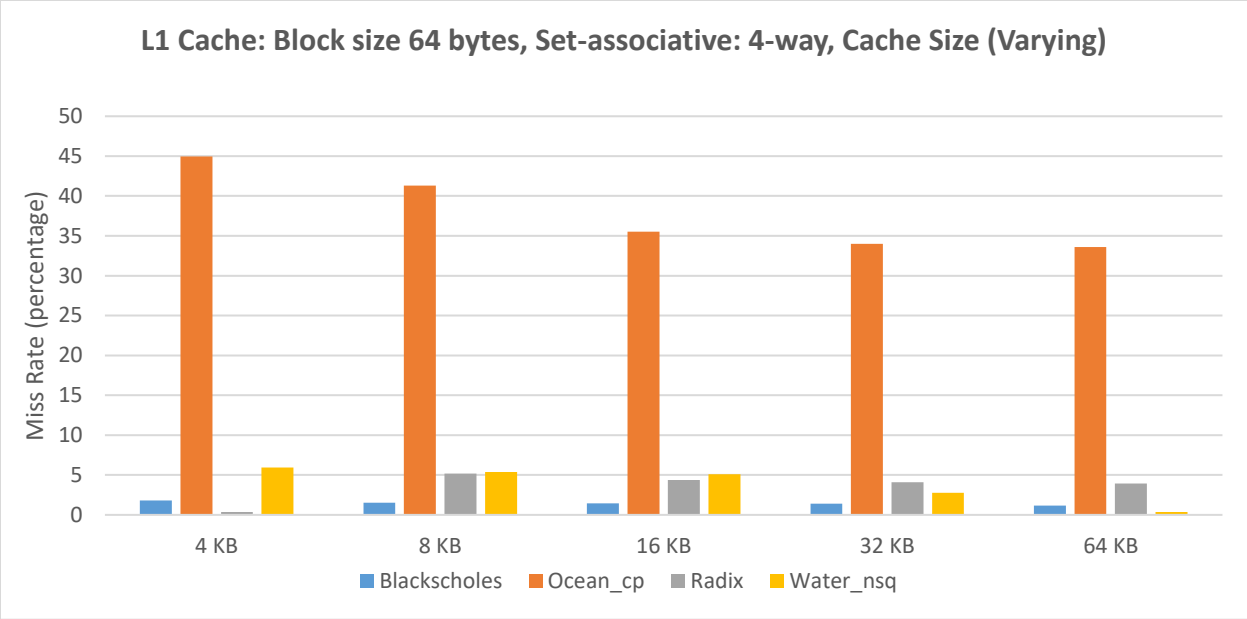

Figure 5. 5(k): Miss Rate in L1 Private Cache (Varying L1 cache, Block size 64 bytes, 8-core, protocol: FIREFLY)

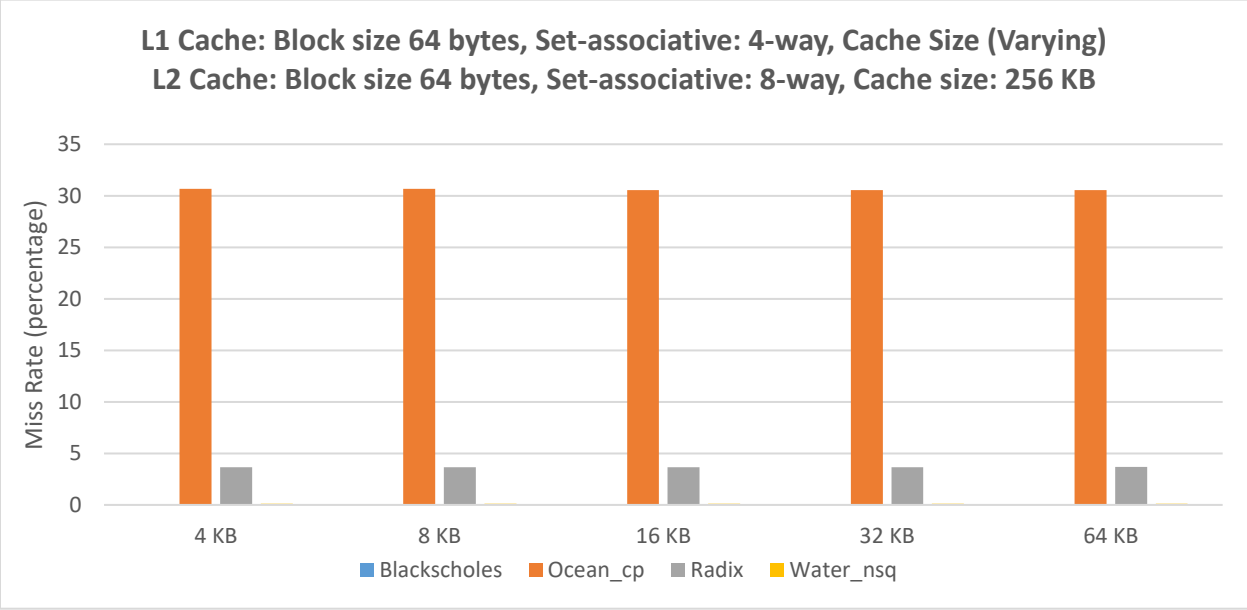

Figure 5. 5(l): Miss Rate in L1 & L2 Private Cache (Varying L1 cache, 256 KB L2 cache, Block size 64 bytes, 8-core, protocol: FIREFLY)

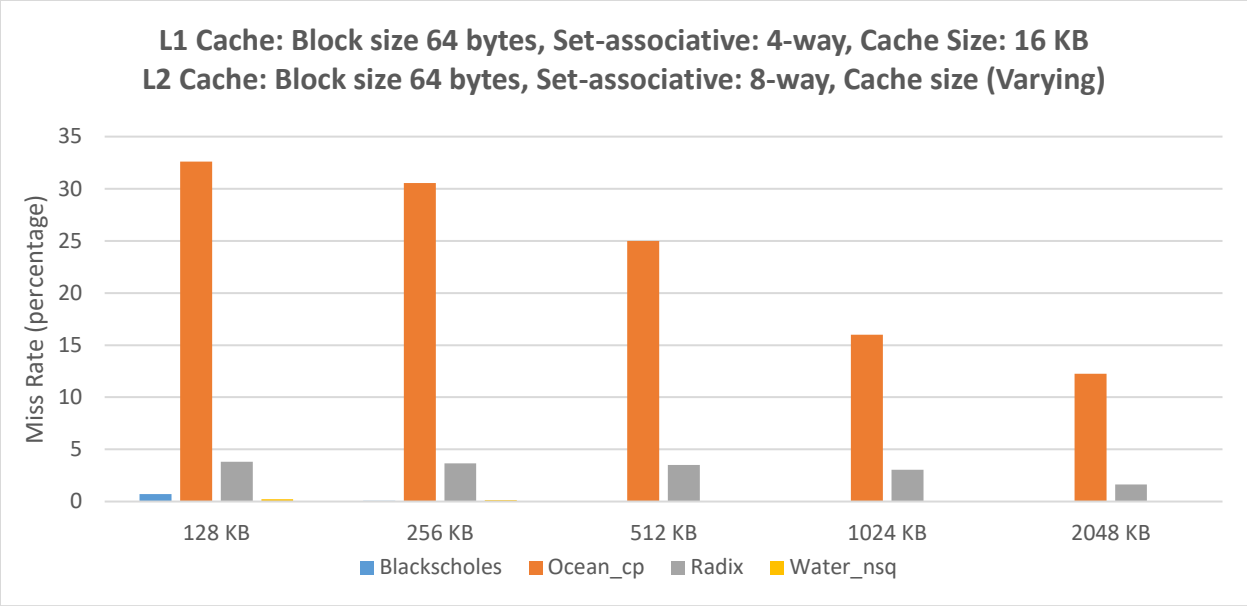

Figure 5. 5(m): Miss Rate in L1 & L2 Private Cache (16 KB L1 cache, varying L2 cache, Block size 64 bytes, 8-core, protocol: FIREFLY)

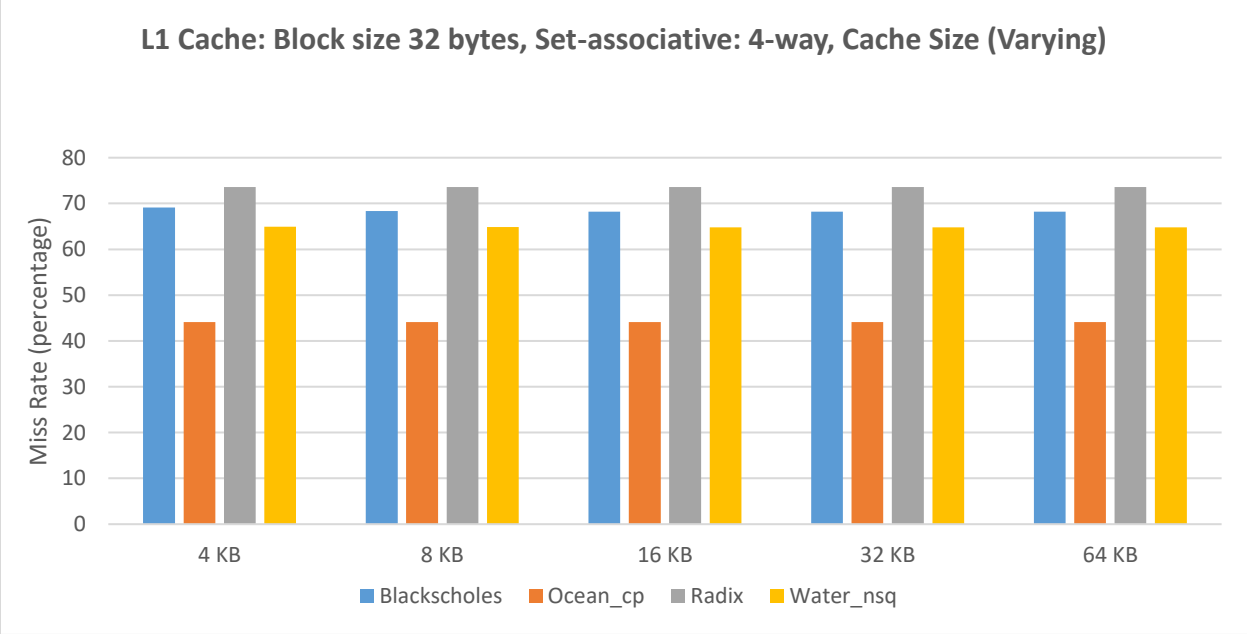

Figure 5. 5(n): Miss Rate in L1 Private Cache (Varying L1 cache, Block size 32 bytes, 16-core, protocol: MESI)

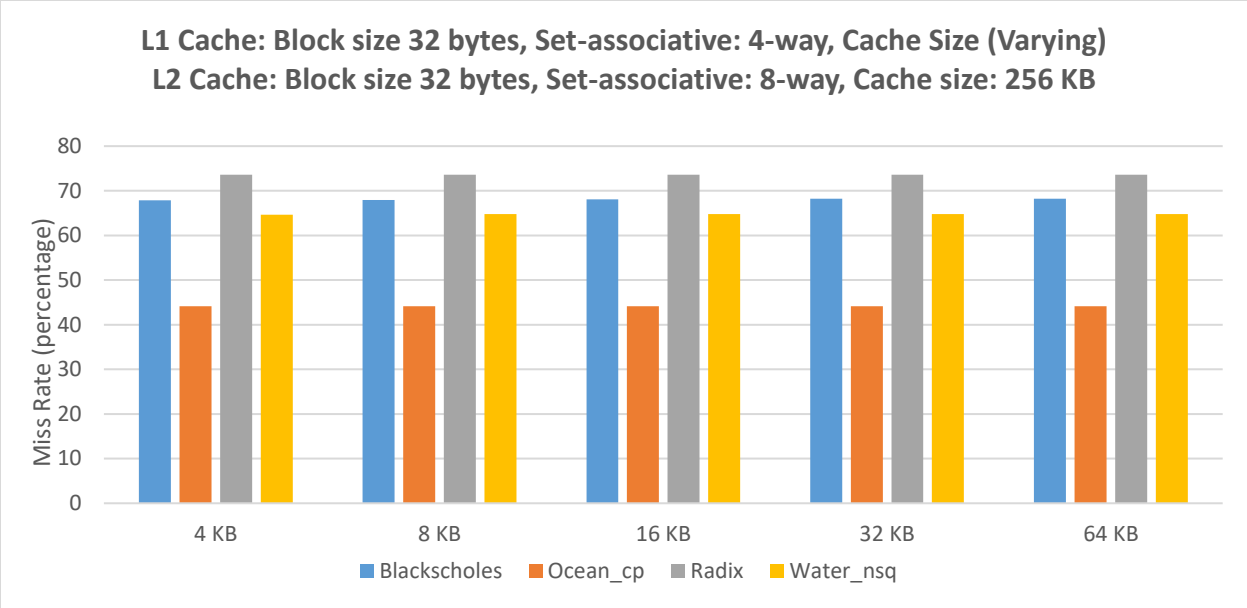

Figure 5. 5(o): Miss Rate in L1 & L2 Private Cache (Varying L1 cache, 256 KB L2 cache, Block size 32 bytes, 16-core, protocol: MESI)

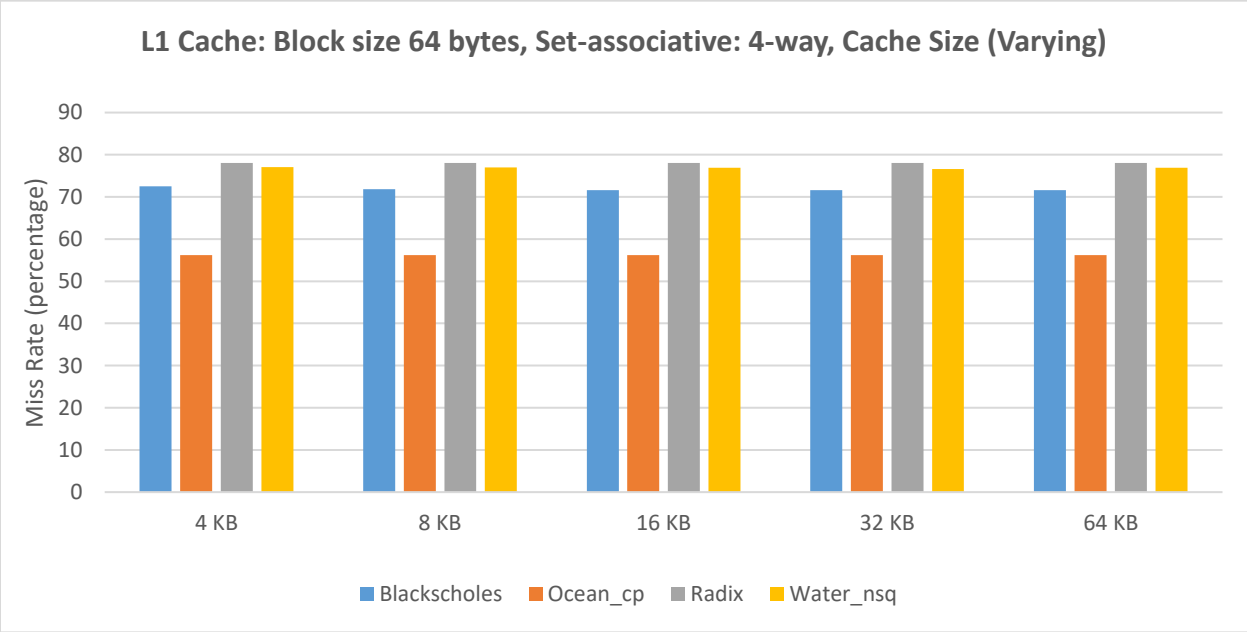

Figure 5. 5(p): Miss Rate in L1 Private Cache (Varying L1 cache, Block size 64 bytes, 16-core, protocol: MESI)

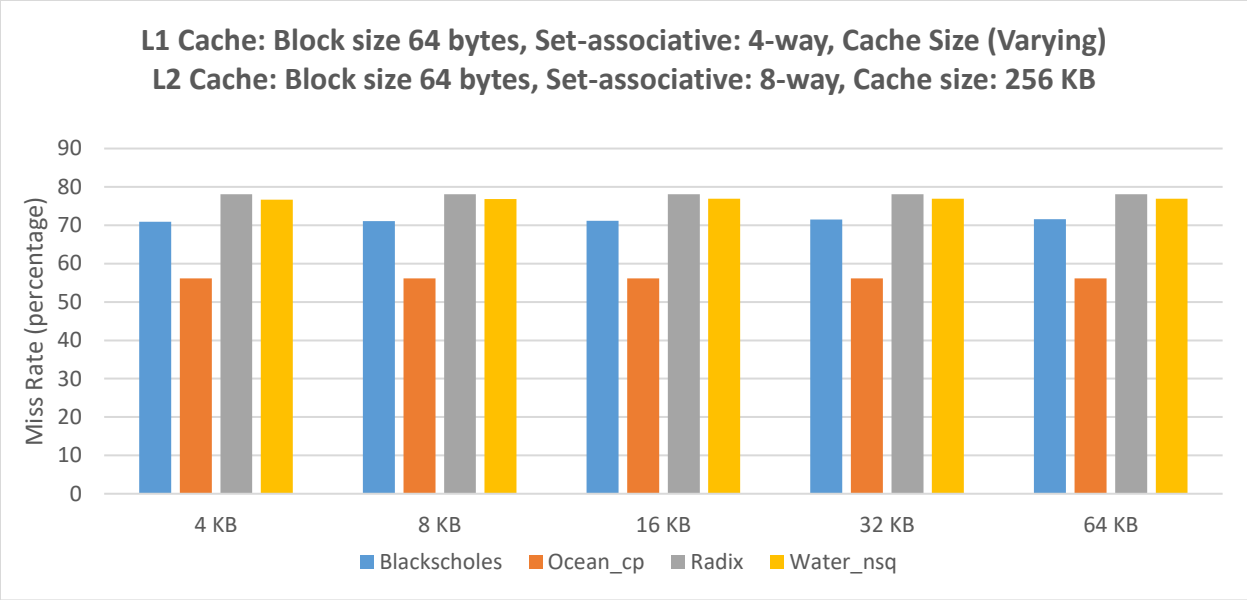

Figure 5. 5(q): Miss Rate in L1 & L2 Private Cache (Varying L1 cache, 256 KB L2 cache, Block size 64 bytes, 16-core, protocol: MESI)

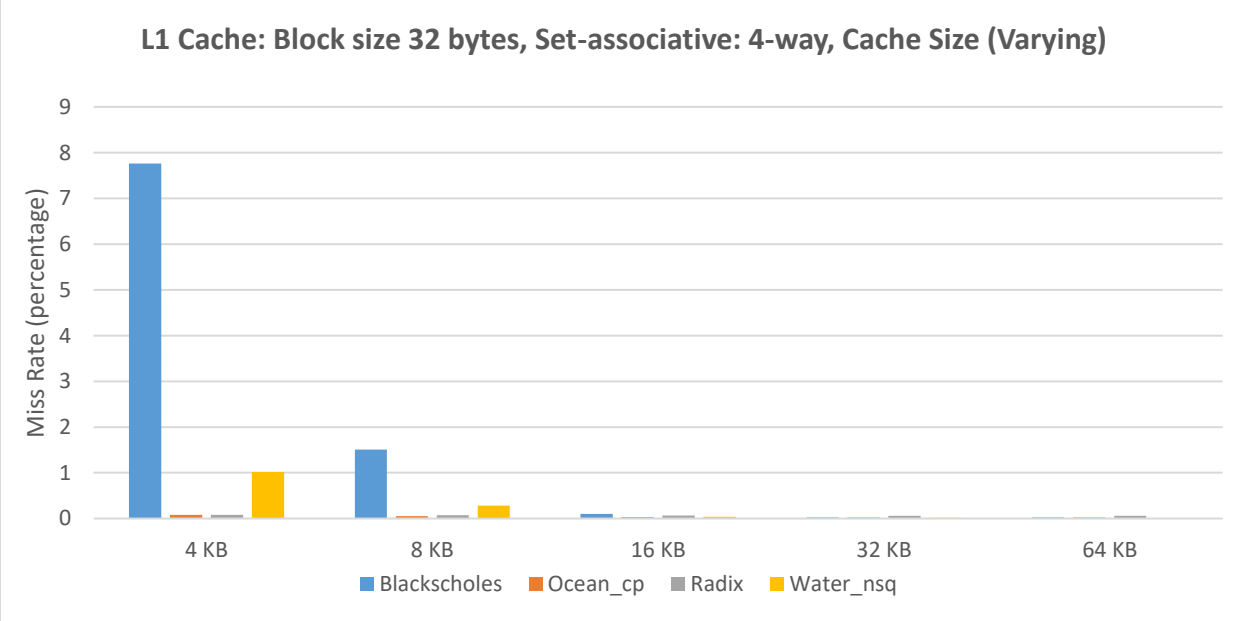

Figure 5.5(r): Miss Rate in L1 Private Cache (Varying L1 cache, Block size 32 bytes, 16-core, protocol: FIREFLY)

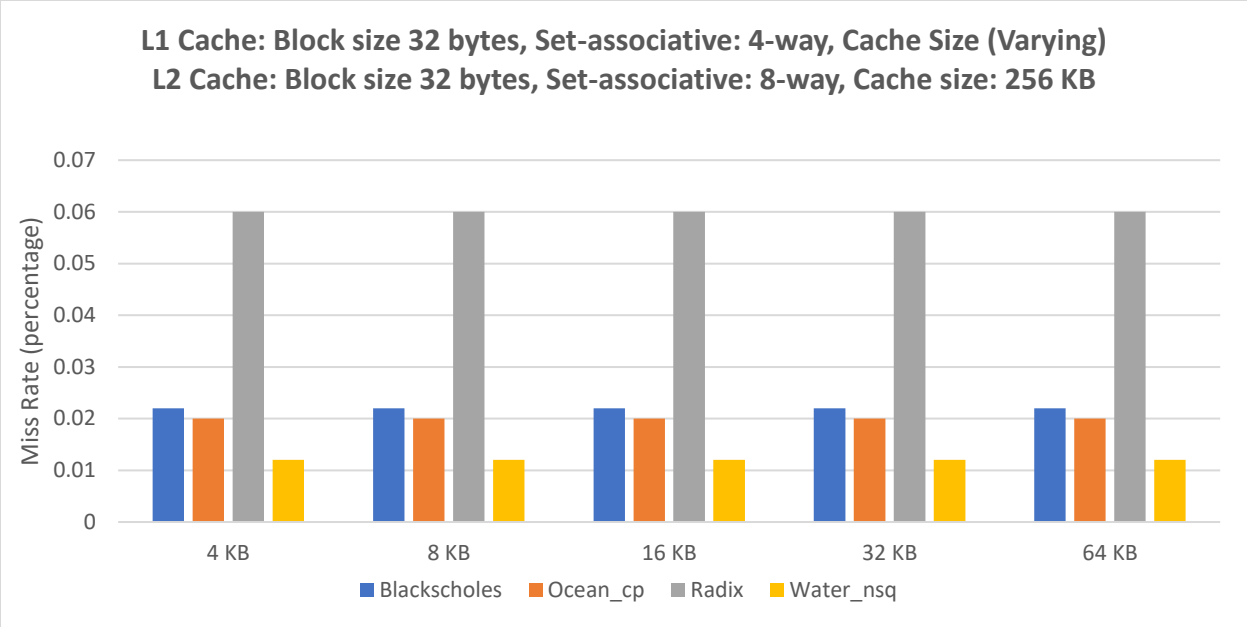

Figure 5. 5(s): Miss Rate in L1 & L2 Private Cache (Varying L1 cache, 256 KB L2 cache, Block size 32 bytes, 16-core, protocol: FIREFLY)

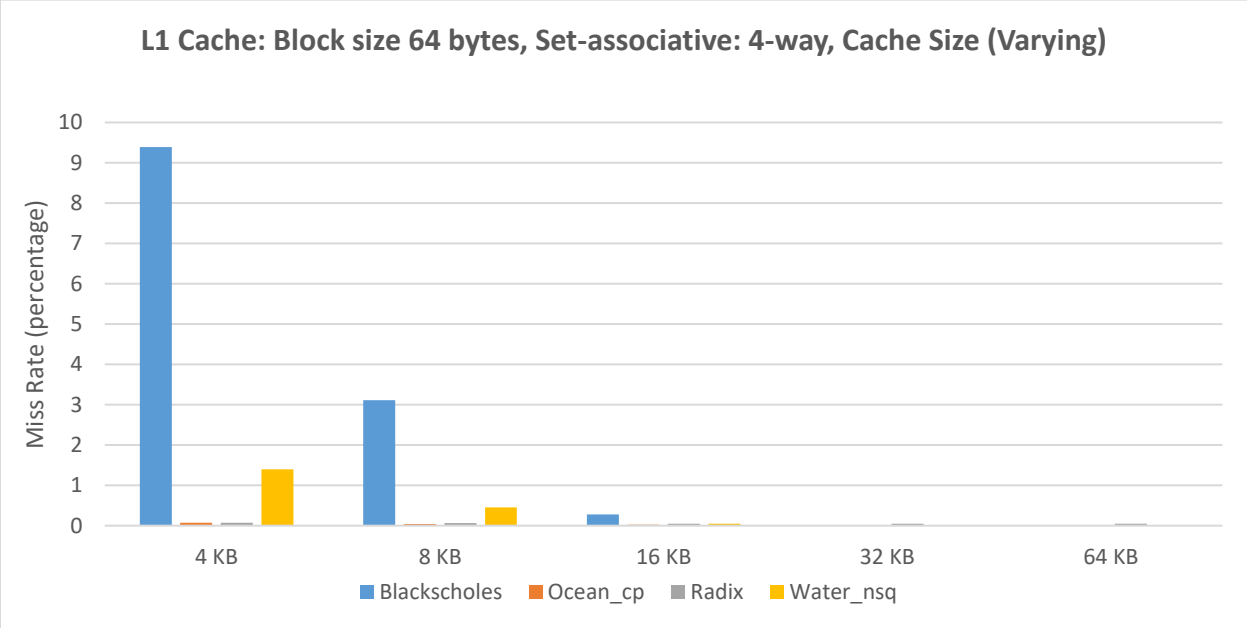

Figure 5. 5(t): Miss Rate in L1 Private Cache (Varying L1 cache, Block size 64 bytes, 16-core, protocol: FIREFLY)

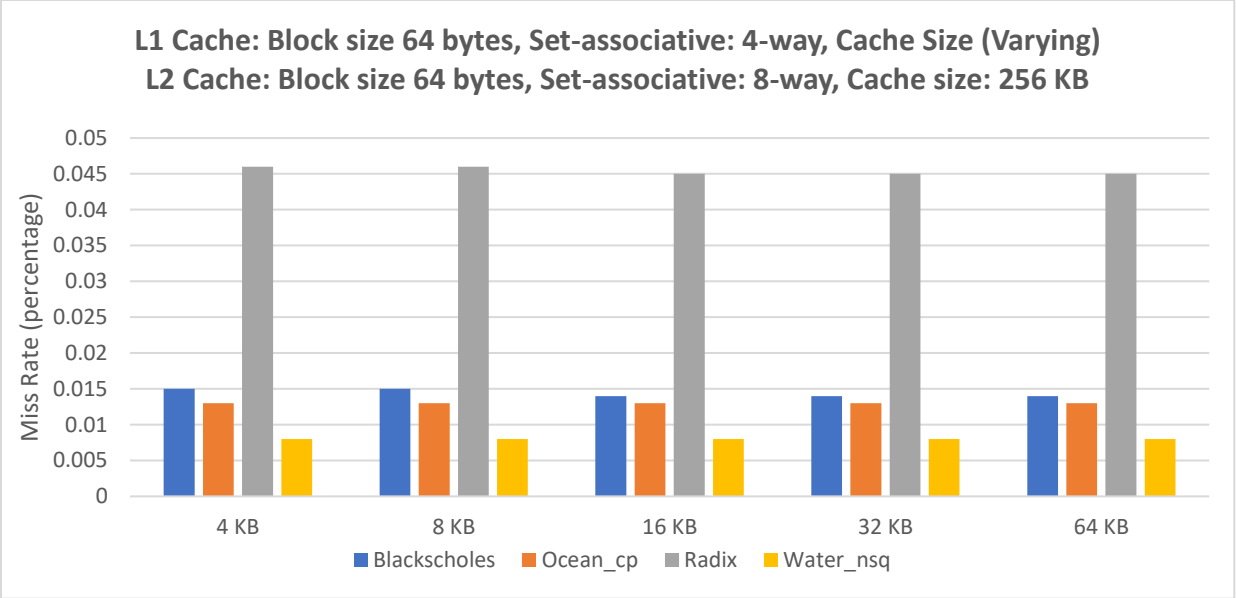

Figure 5. 5(u): Miss Rate in L1 & L2 Private Cache (Varying L1 cache, 256 KB L2 cache, Block size 64 bytes, 16-core, protocol: FIREFLY)

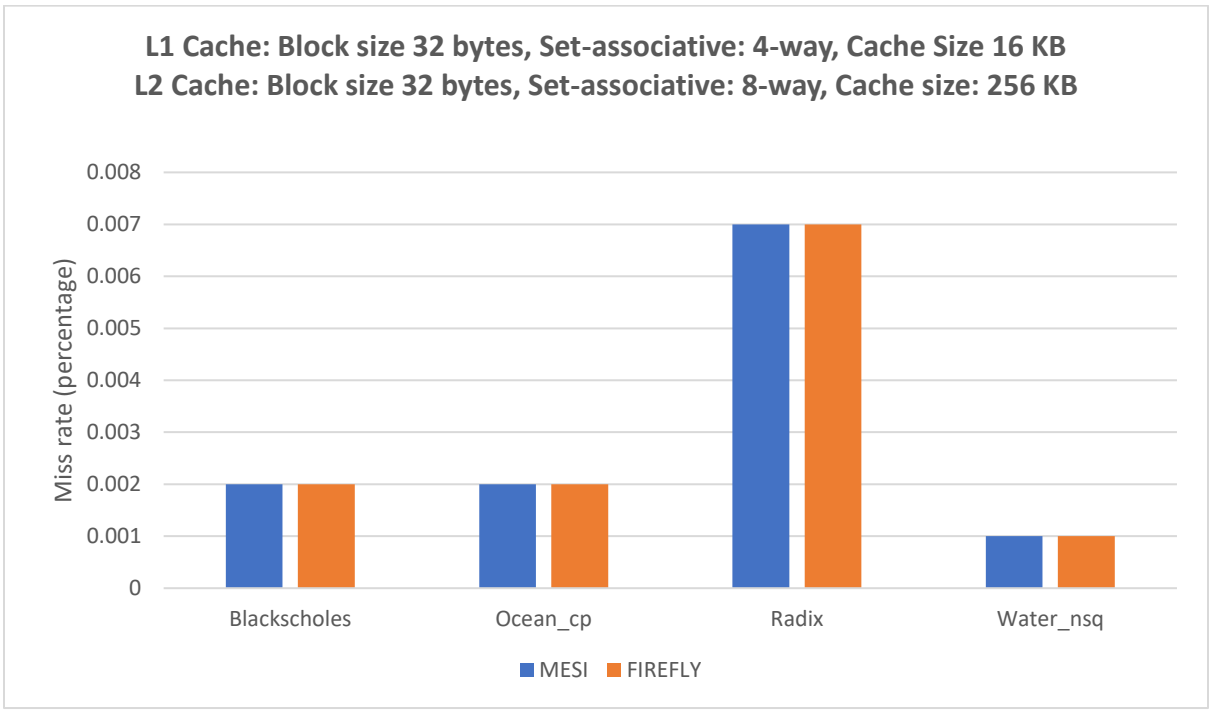

Figure 5.6(a): Comprison between MESI and FIREFLY Protocols regarding Global Cache Miss rates (16-core)

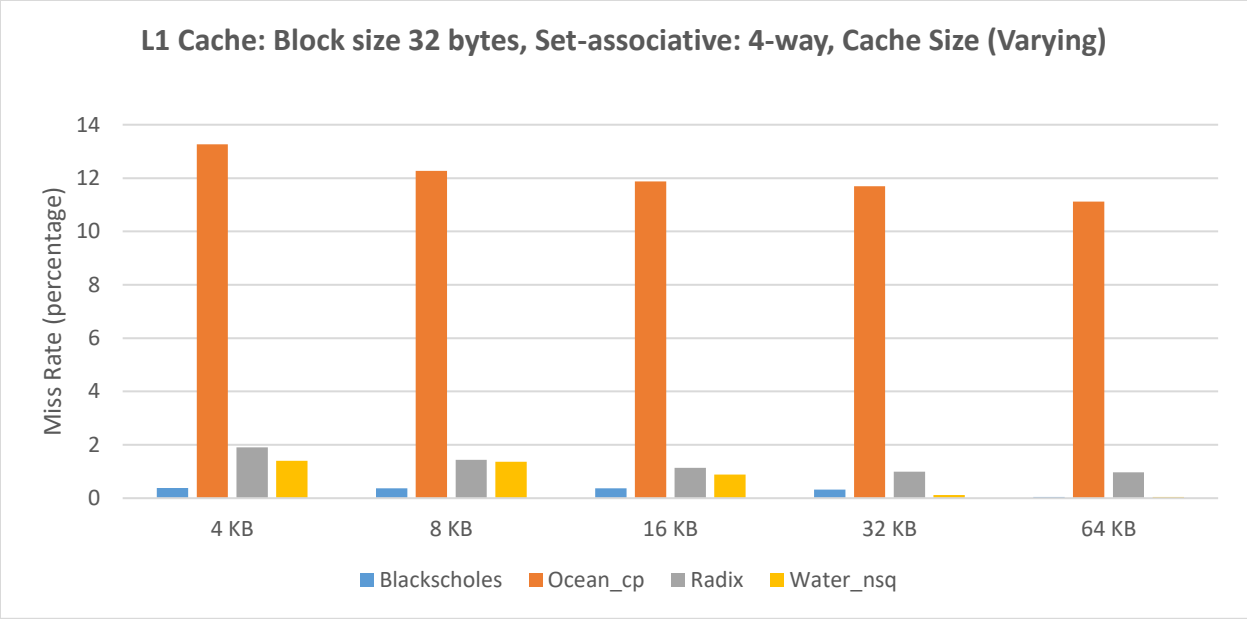

Figure 5. 6(b): Global Miss Rate (Varying L1 cache, Block size 32 bytes, 8-core, protocol: MESI)

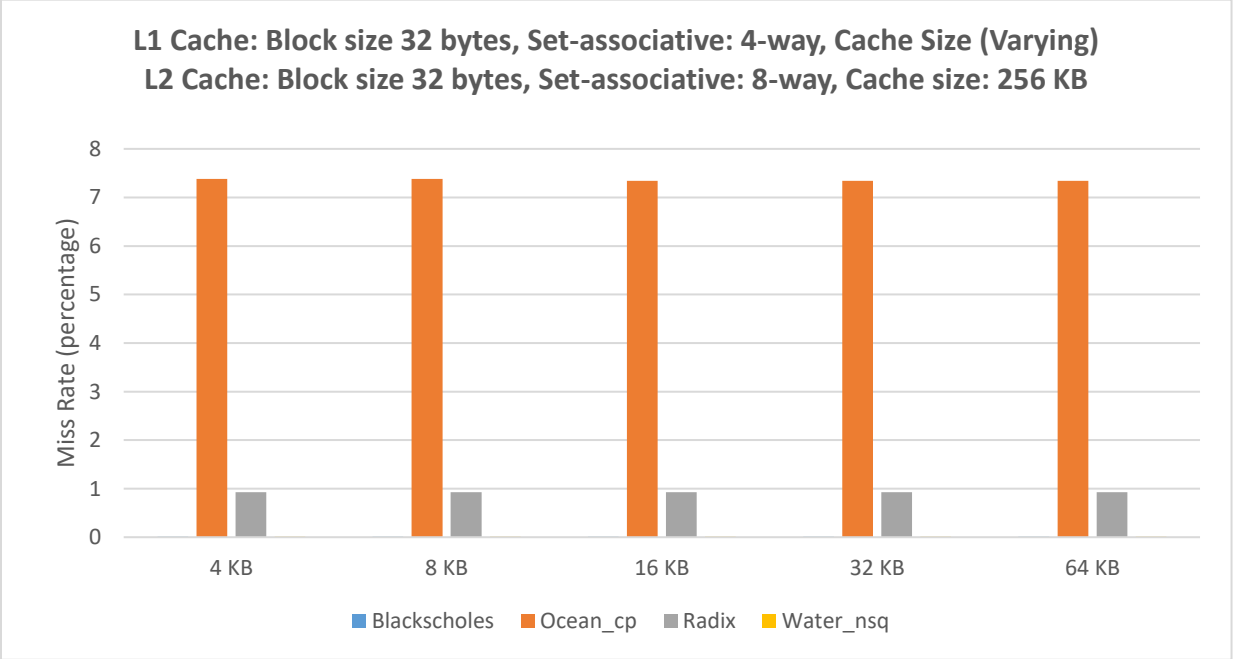

Figure 5. 6(c): Global Miss Rate (Varying L1 cache, 256 KB L2 cache, Block size 32 bytes, 8-core, protocol: MESI)

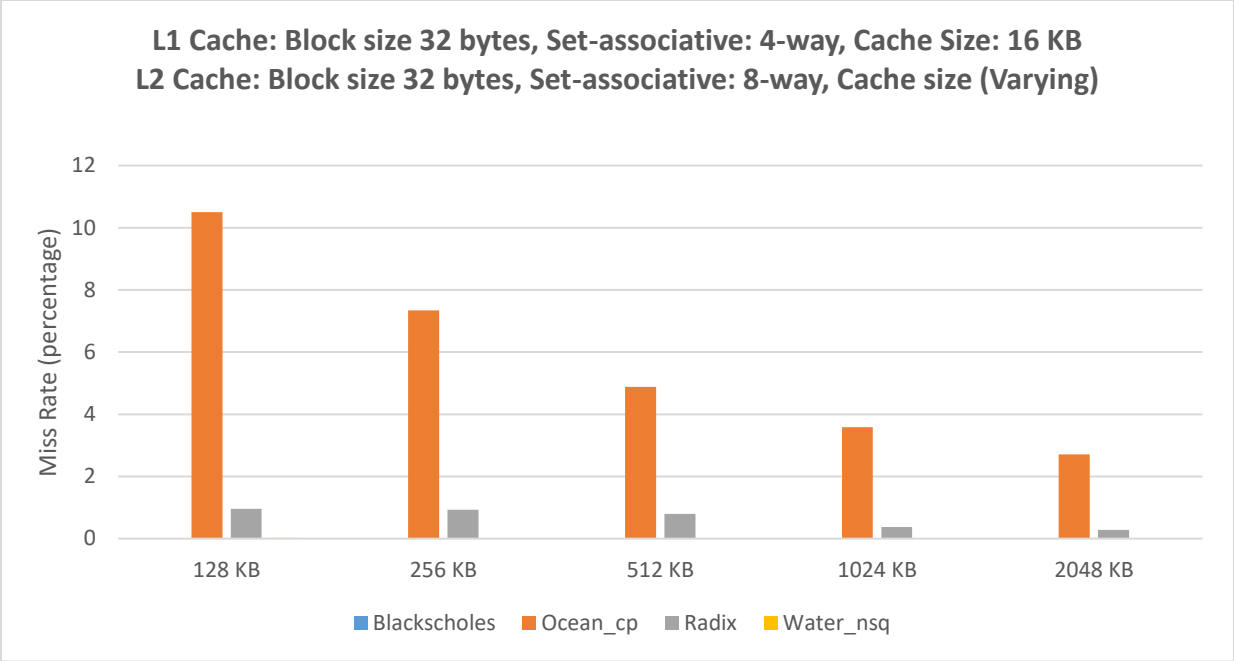

Figure 5. 6(d): Global Miss Rate (16 KB L1 cache, varying L2 cache, Block size 32 bytes, 8-core, protocol: MESI)

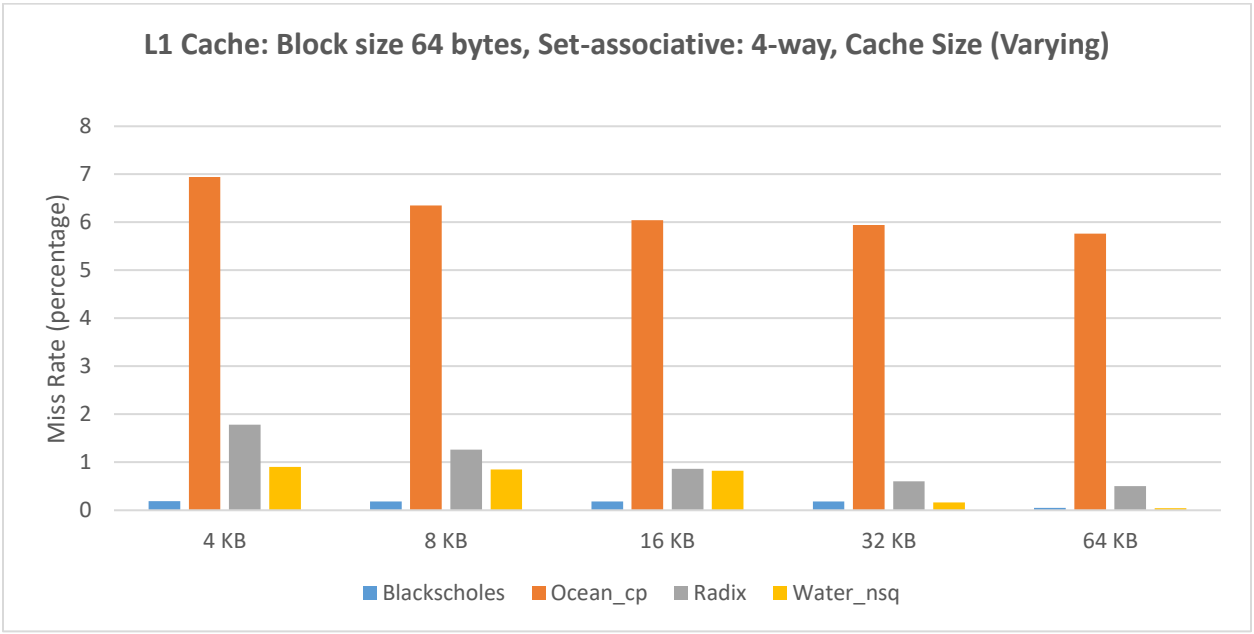

Figure 5. 6(e): Global Miss Rate (Varying L1 cache, Block size 64 bytes, 8-core, protocol: MESI)

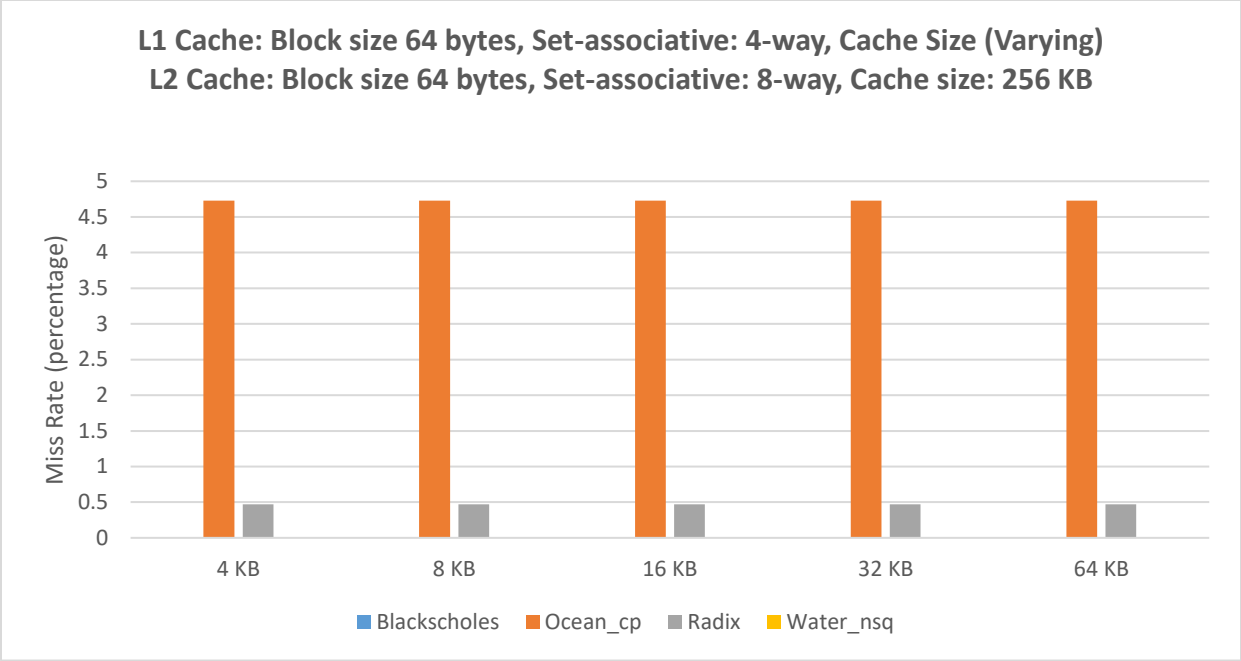

Figure 5. 6(f): Global Miss Rate (Varying L1 cache, 256 KB L2 cache, Block size 64 bytes, 8-core, protocol: MESI)

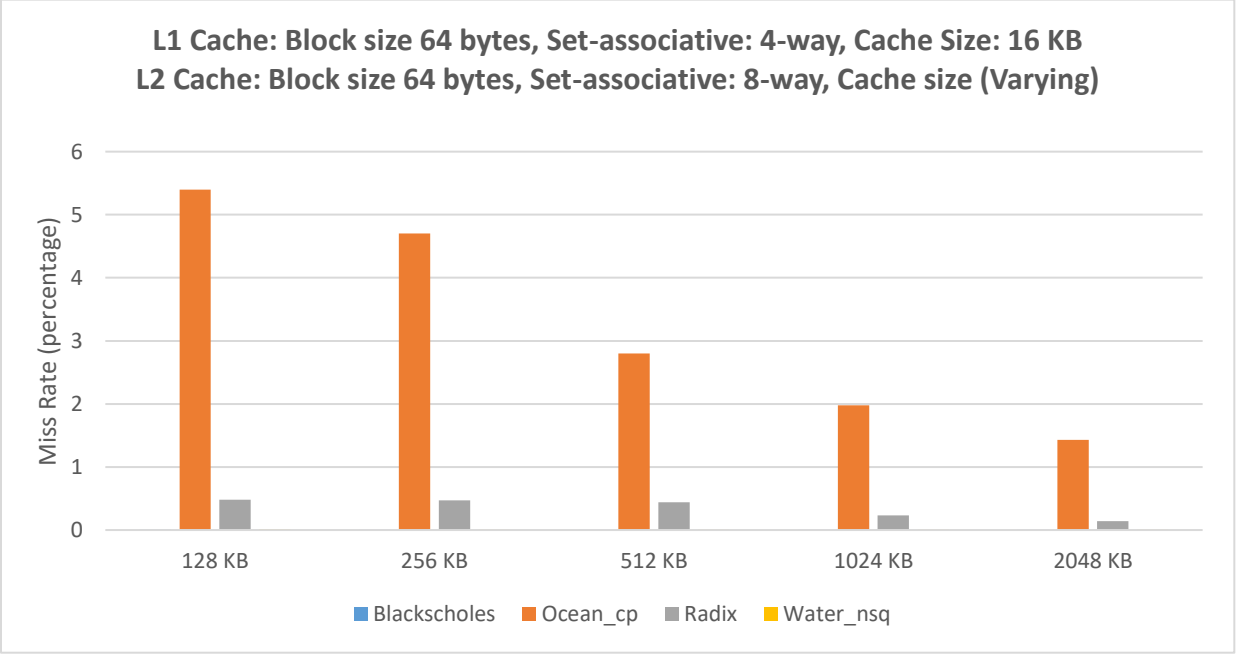

Figure 5. 6(g): Global Miss Rate (16 KB L1 cache, varying L2 cache, Block size 64 bytes, 8-core, protocol: MESI)

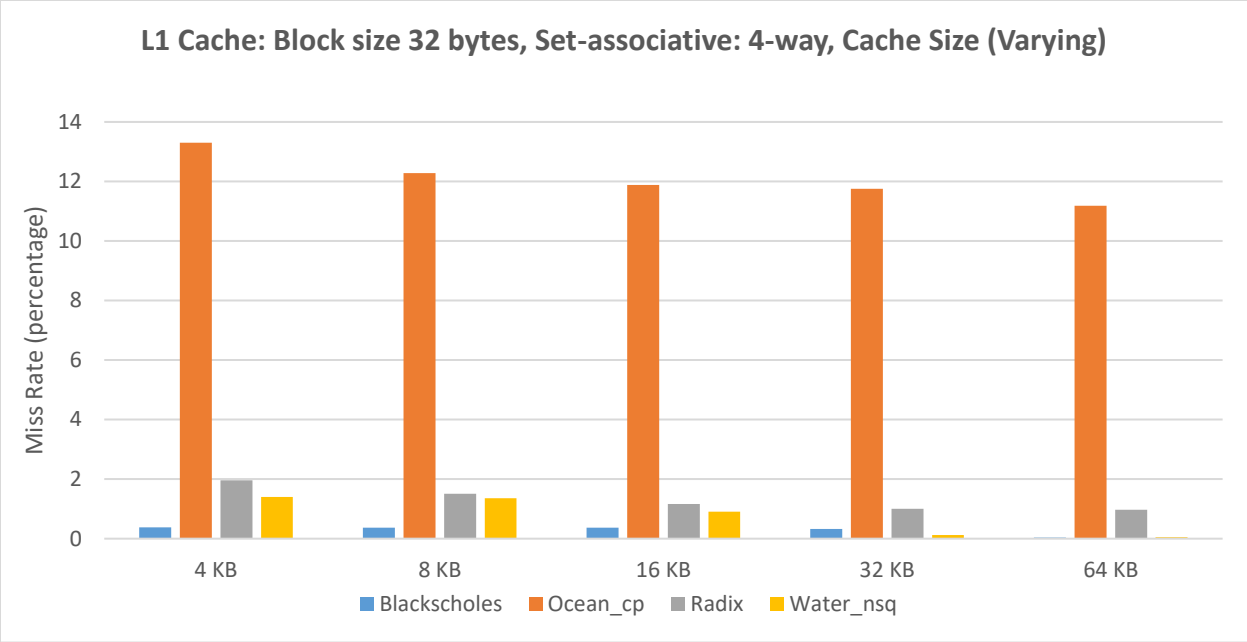

Figure 5. 6(h): Global Miss Rate (Varying L1 cache, Block size 32 bytes, 8-core, protocol: FIREFLY)

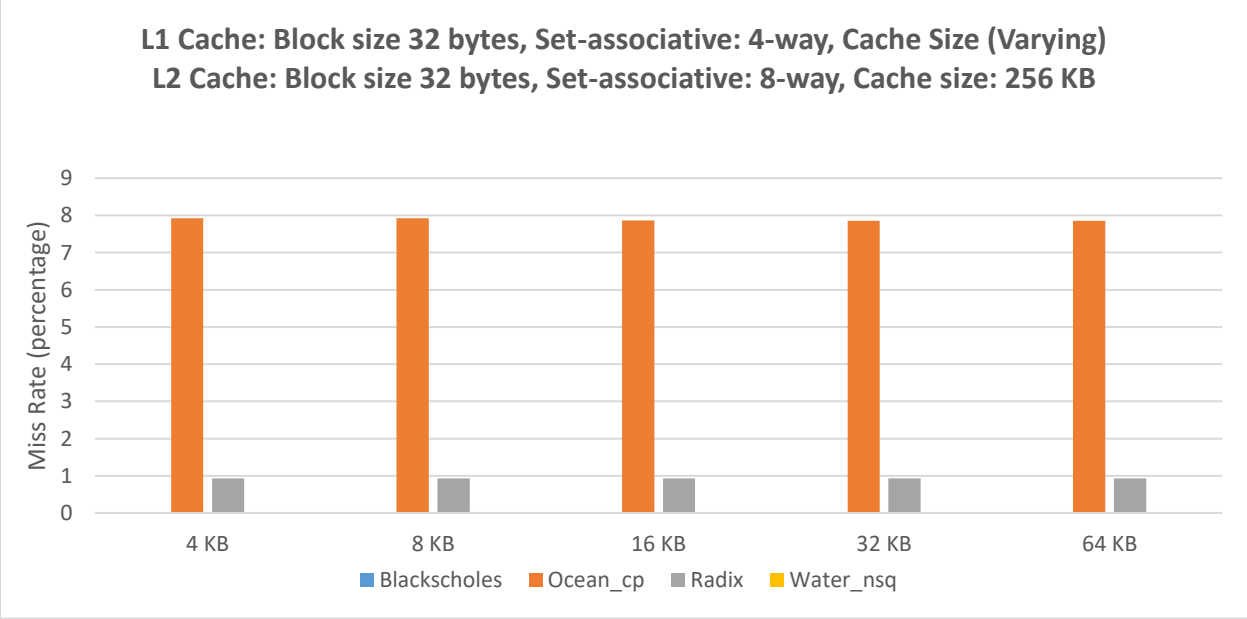

Figure 5. 6(i): Global Miss Rate (Varying L1 cache, 256 KB L2 cache, Block size 32 bytes, 8-core, protocol: FIREFLY)

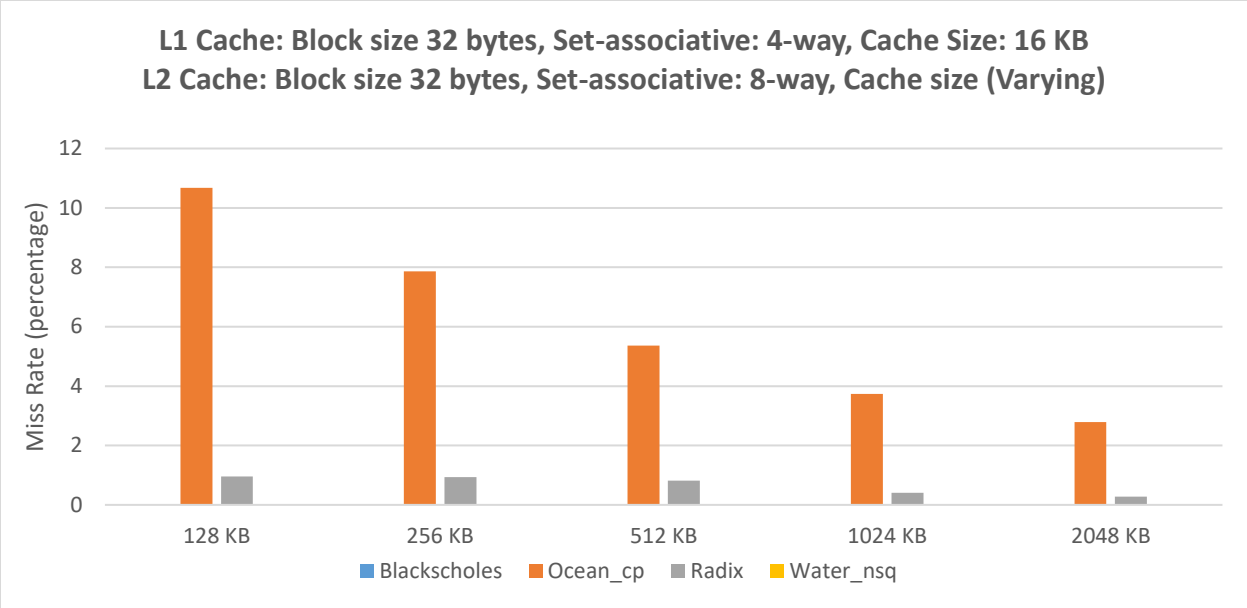

Figure 5. 6(j): Global Miss Rate (16 KB L1 cache, varying L2 cache, Block size 32 bytes, 8-core, protocol: FIREFLY)

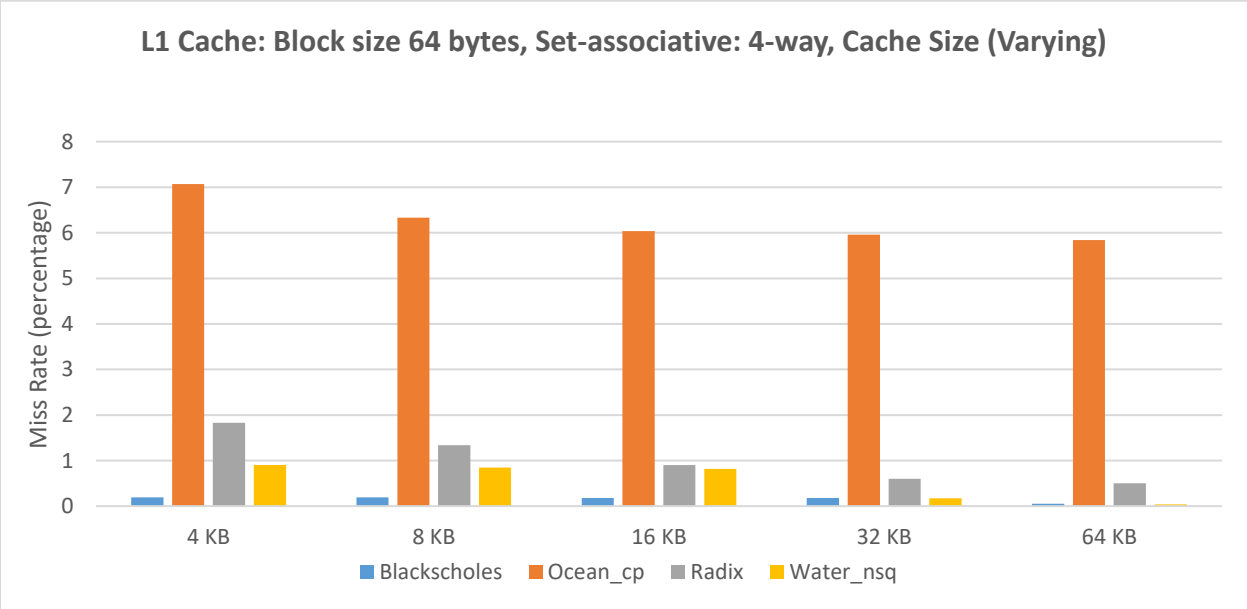

Figure 5. 6(k): Global Miss Rate (Varying L1 cache, Block size 64 bytes, 8-core, protocol: FIREFLY)

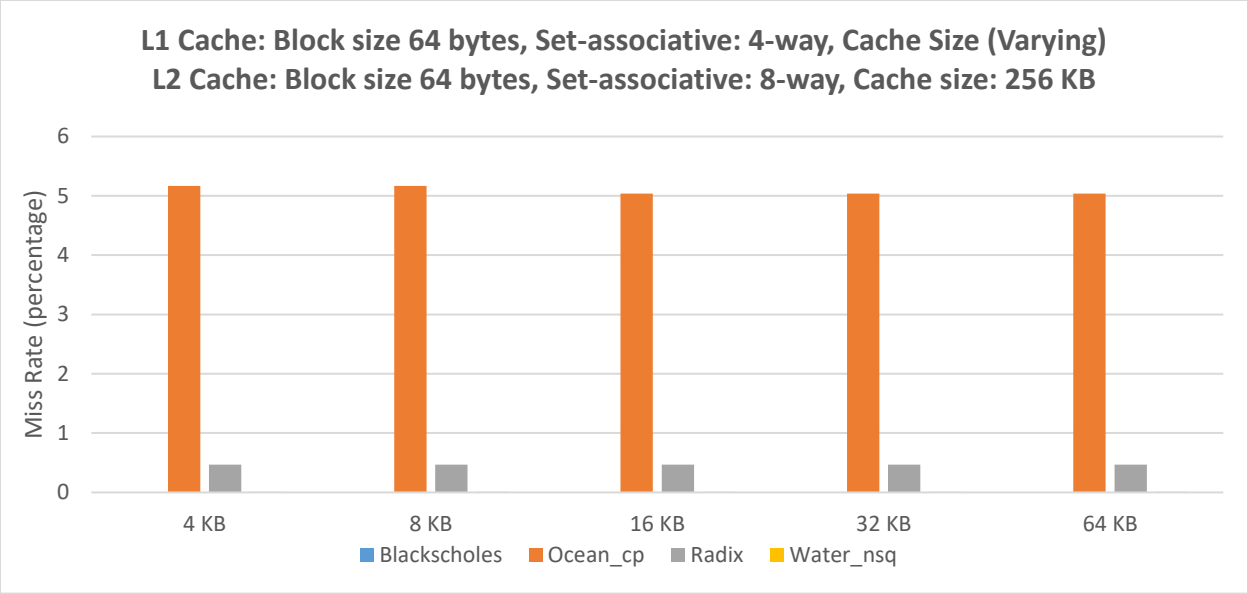

Figure 5. 6(l): Global Miss Rate (Varying L1 cache, 256 KB L2 cache, Block size 64 bytes, 8-core, protocol: FIREFLY)

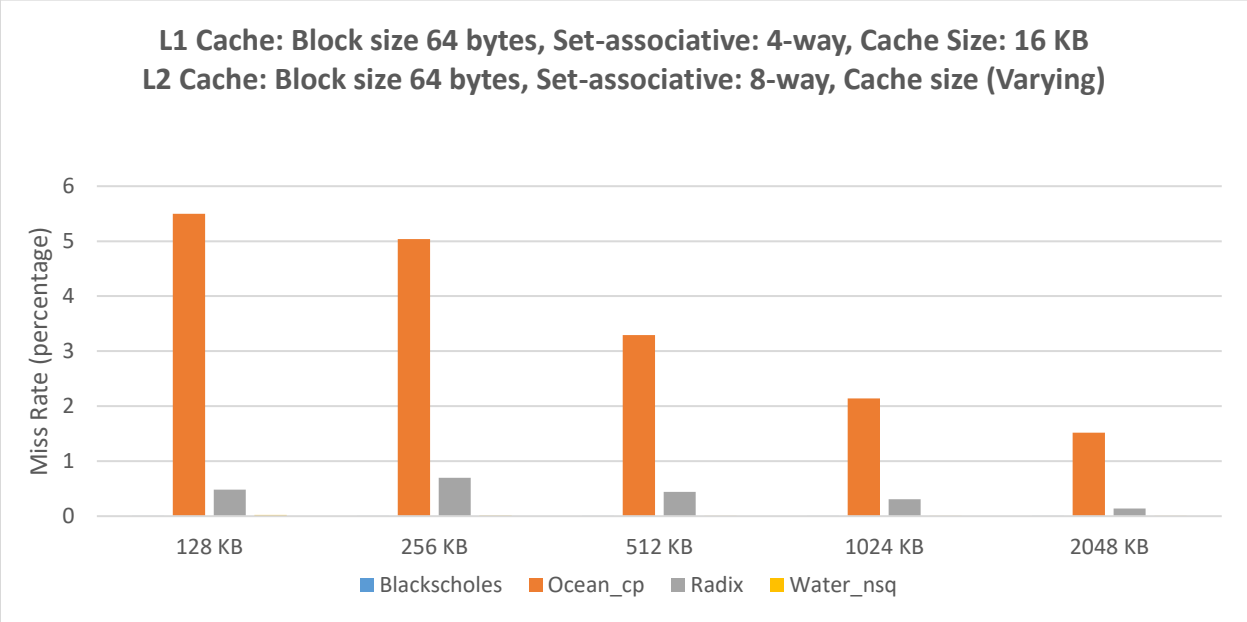

Figure 5. 6(m): Global Miss Rate (16 KB L1 cache, varying L2 cache, Block size 64 bytes, 8-core, protocol: FIREFLY)

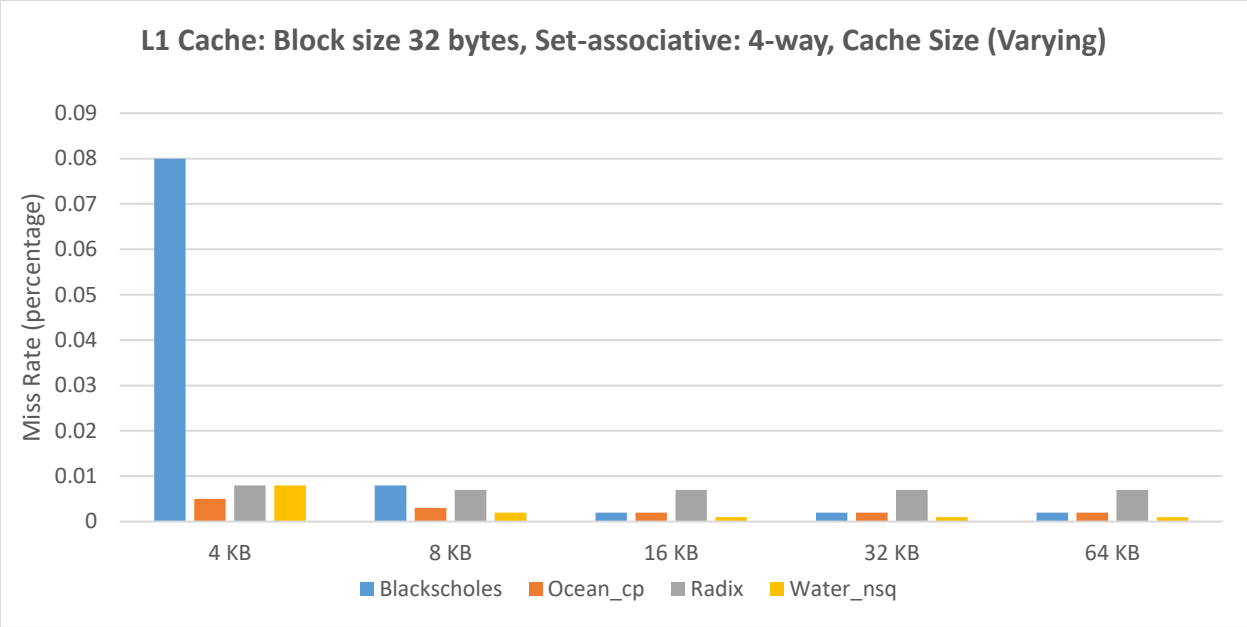

Figure 5. 6(n): Global Miss Rate (Varying L1 cache, Block size 32 bytes, 16-core, protocol: MESI)

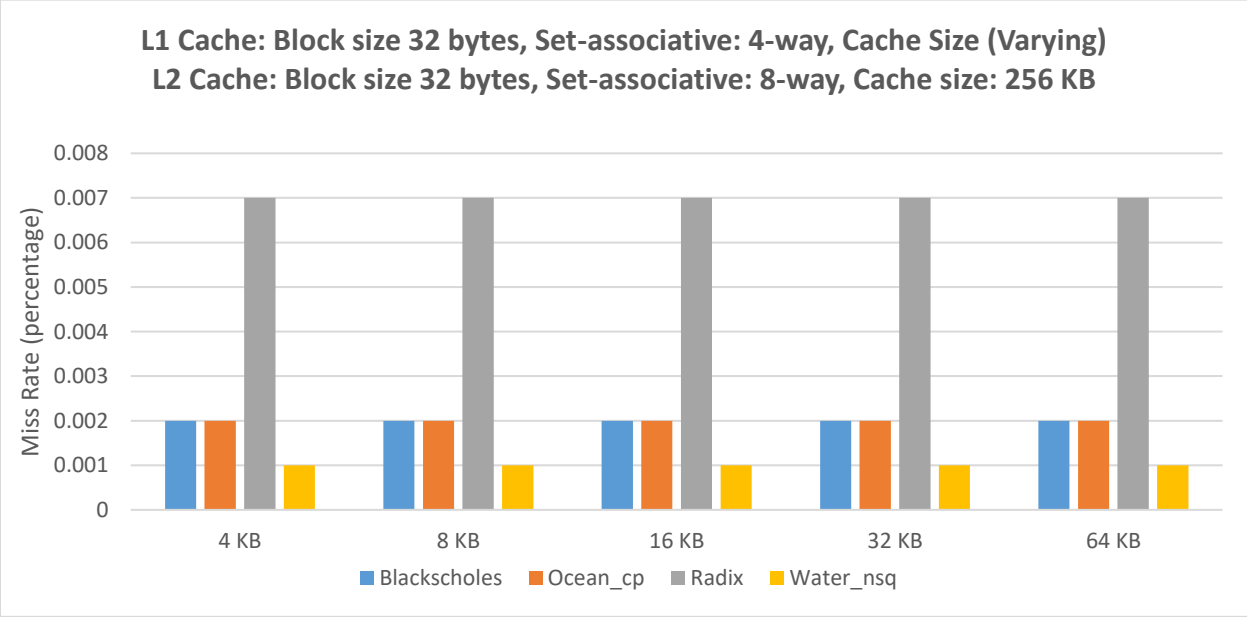

Figure 5. 6(o): Global Miss Rate (Varying L1 cache, 256 KB L2 cache, Block size 32 bytes, 16-core, protocol: MESI)

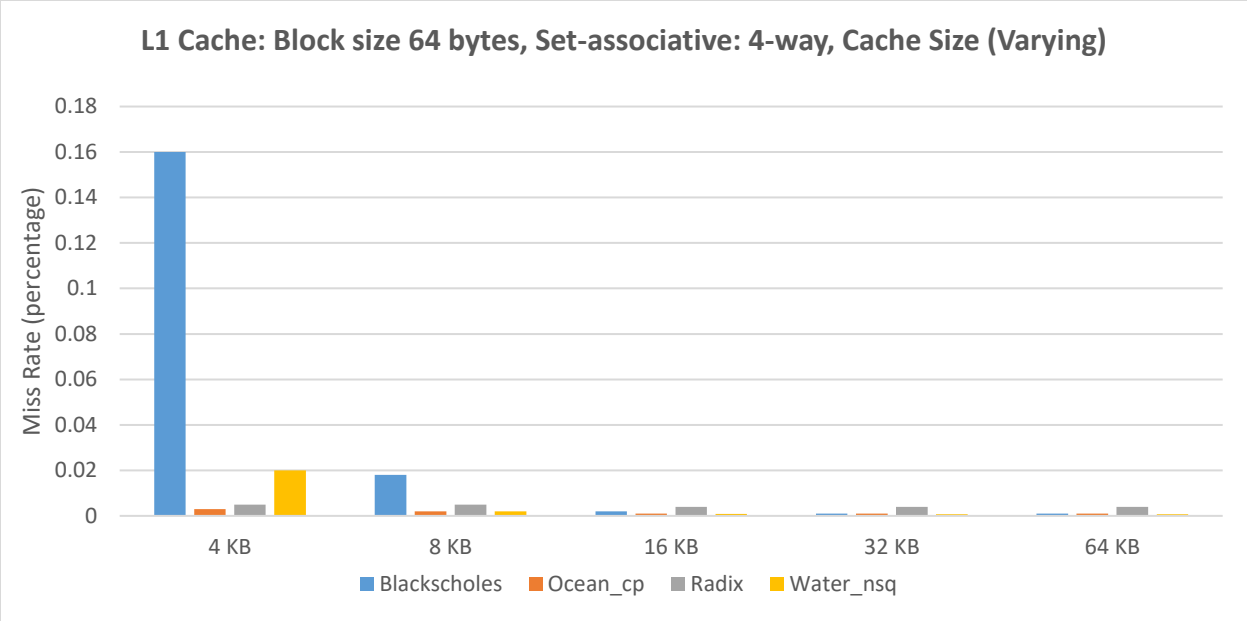

Figure 5. 6(p): Global Miss Rate (Varying L1 cache, Block size 64 bytes, 16-core, protocol: MESI)

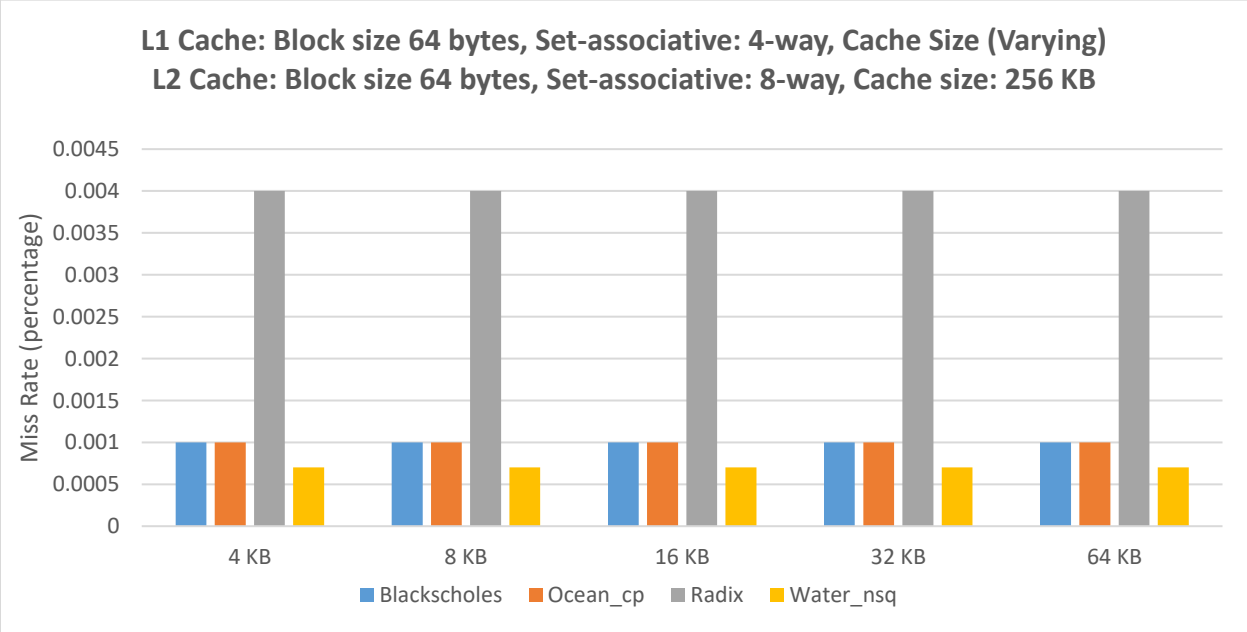

Figure 5. 6(q): Global Miss Rate (Varying L1 cache, 256 KB L2 cache, Block size 64 bytes, 16-core, protocol: MESI)

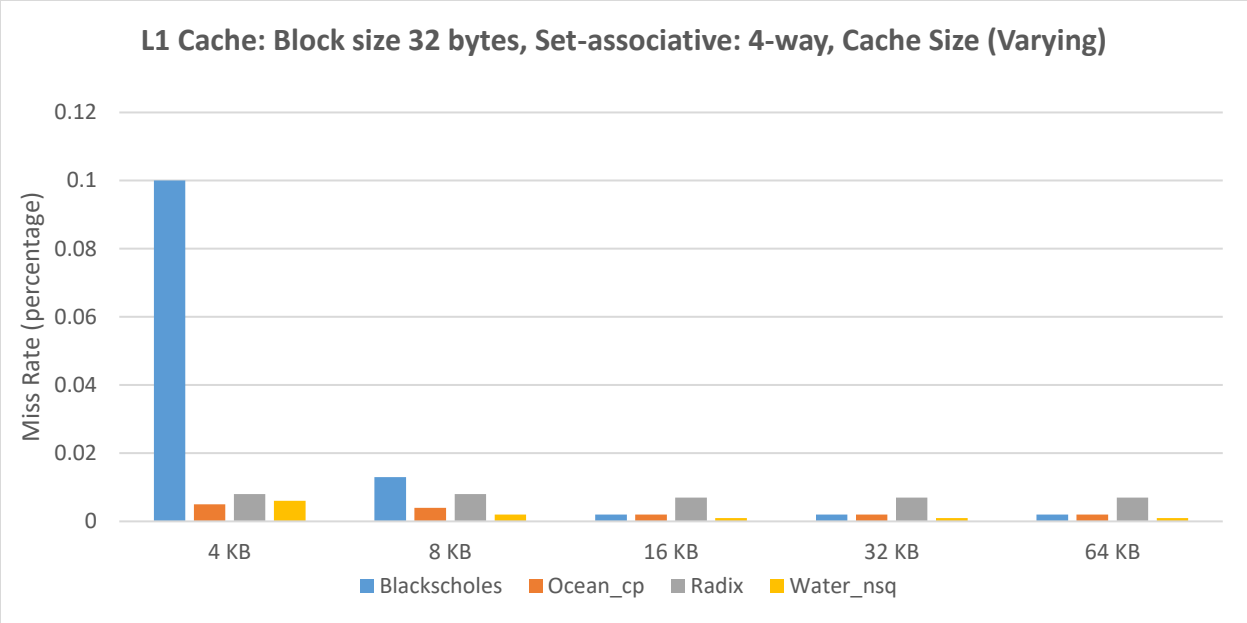

Figure 5. 6(r): Global Miss Rate (Varying L1 cache, Block size 32 bytes, 16-core, protocol: FIREFLY)

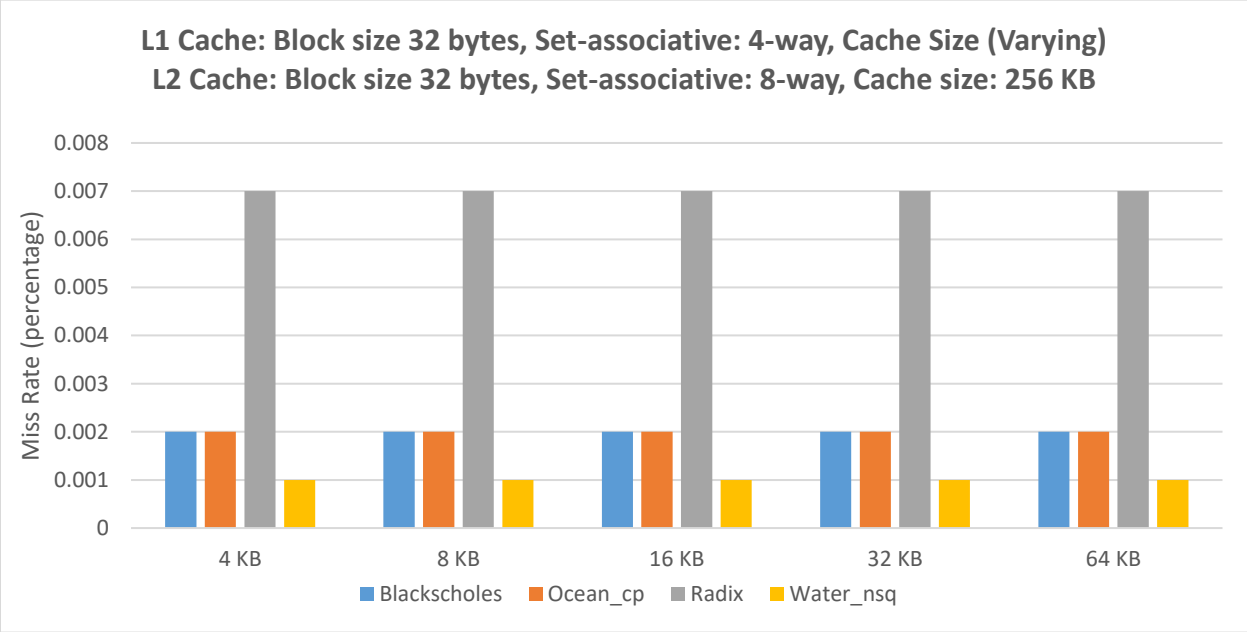

Figure 5. 6(s): Global Miss Rate (Varying L1 cache, 256 KB L2 cache, Block size 32 bytes, 16-core, protocol: FIREFLY)

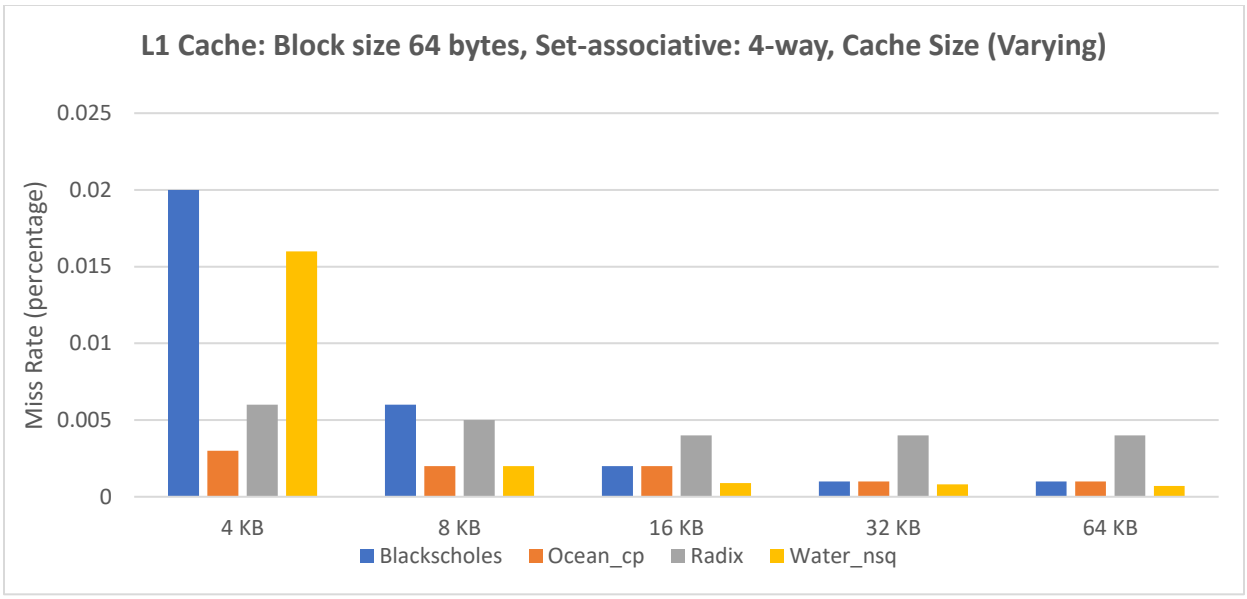

Figure 5. 6(t): Global Miss Rate (Varying L1 cache, Block size 64 bytes, 16-core, protocol: FIREFLY)

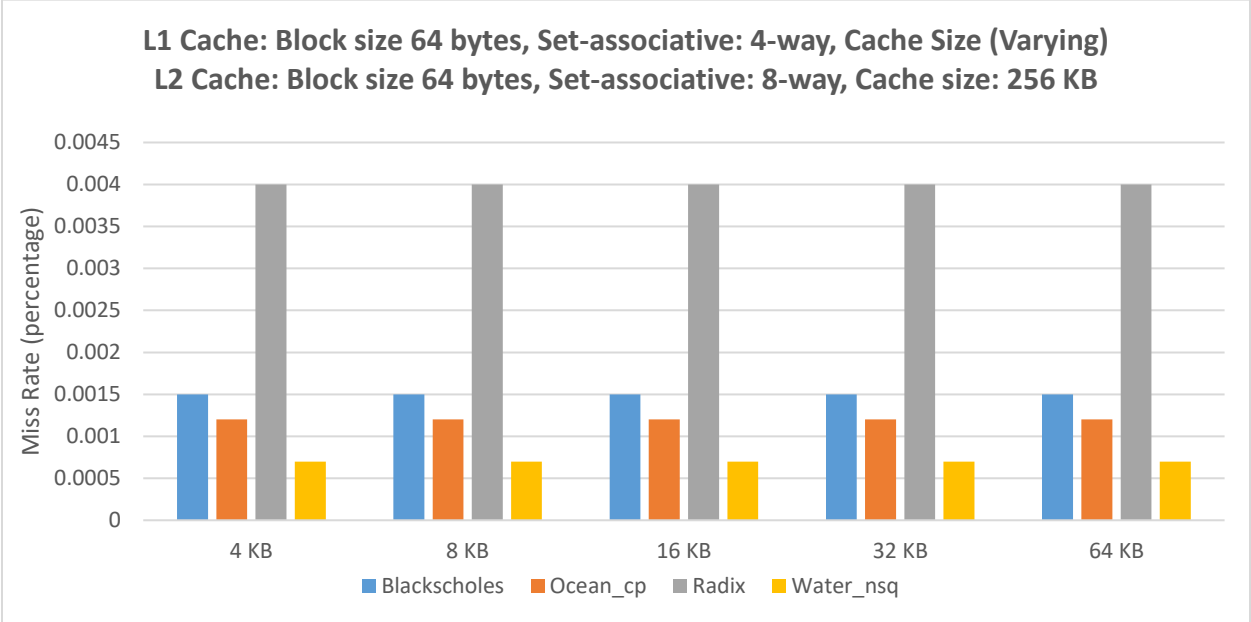

Figure 5. 6(u): Global Miss Rate (Varying L1 cache, 256 KB L2 cache, Block size 64 bytes, 16-core, protocol: FIREFLY)

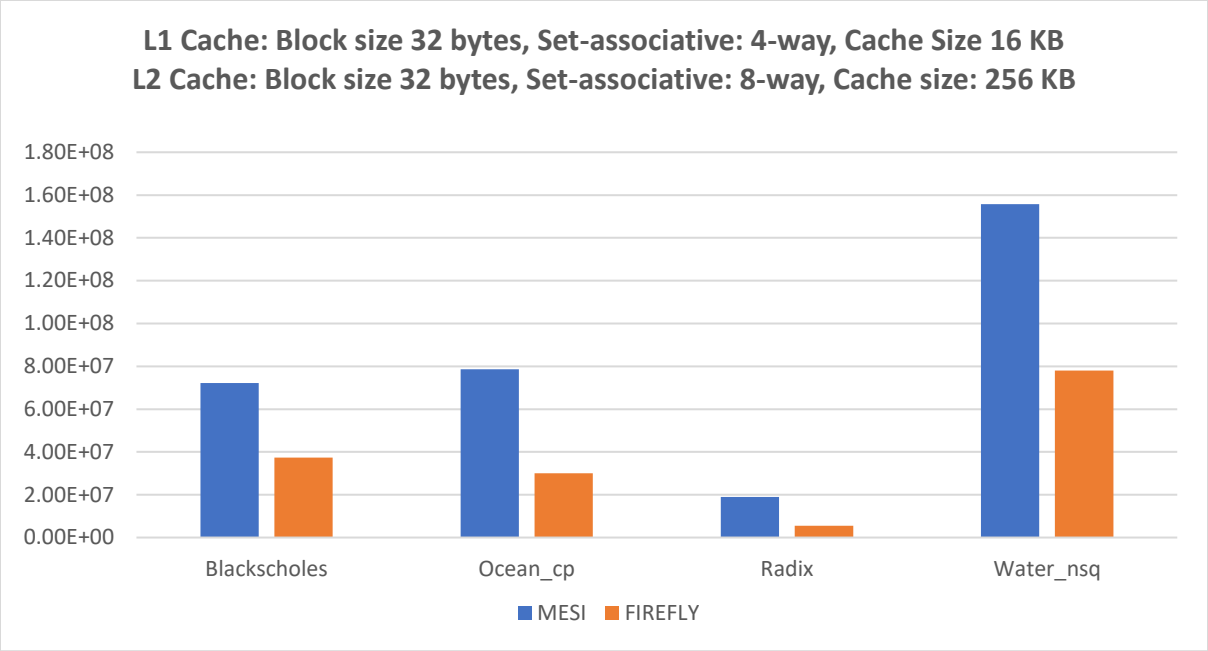

Figure 5.7(a): Comprison between MESI and FIREFLY Protocols regarding # Bus Traffic (16-core)

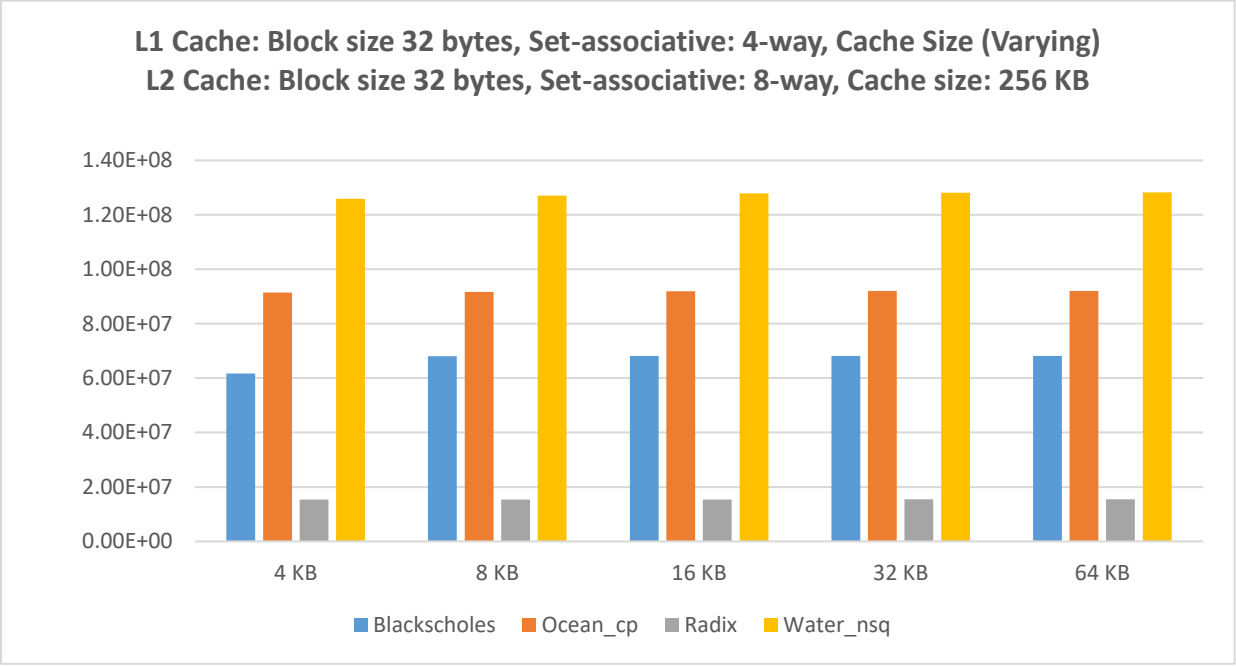

Figure 5. 7(b): # Bus Traffic (Varying L1 cache, 256 KB L2 cache, Block size 32 bytes, 8-core, protocol: MESI)

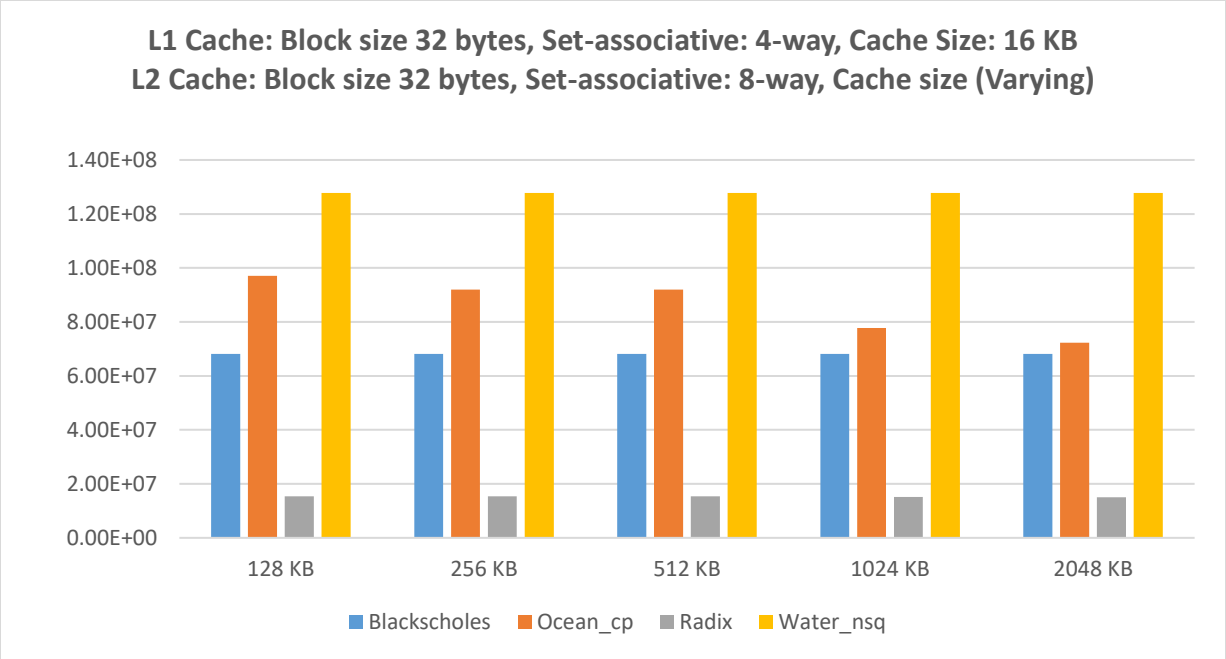

Figure 5. 7(c): # Bus Traffic (16 KB L1 cache, varying L2 cache, Block size 32 bytes, 8-core, protocol: MESI)

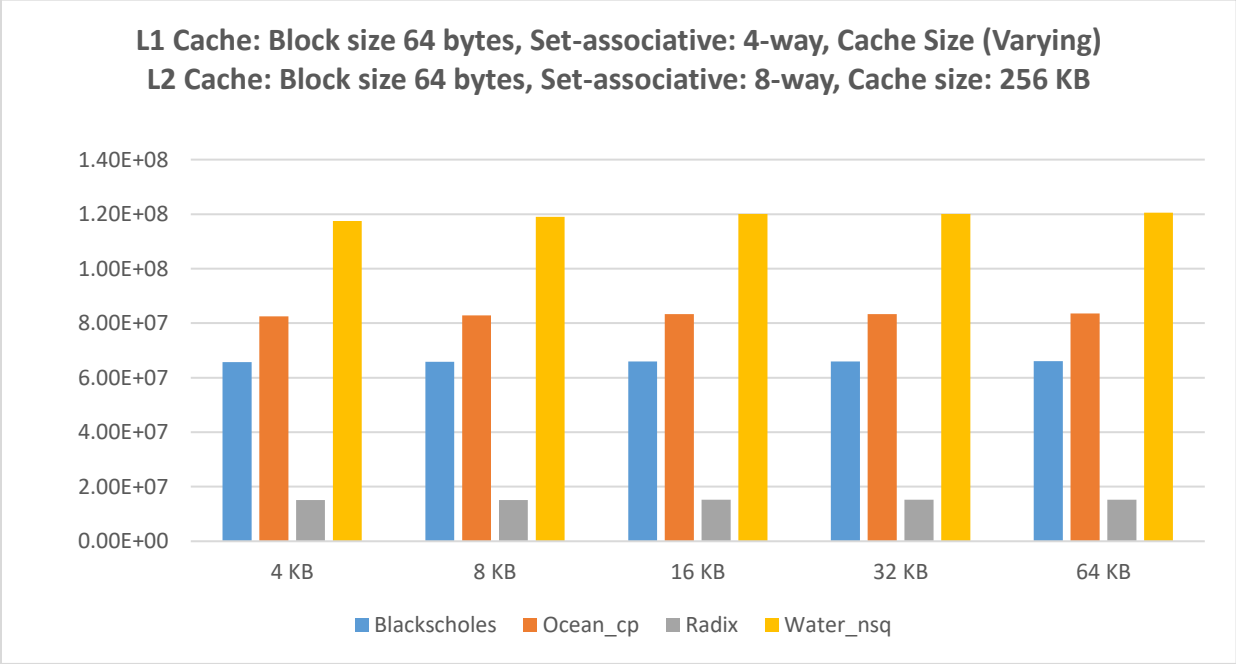

Figure 5. 7(d): # Bus Traffic (Varying L1 cache, 256 KB L2 cache, Block size 64 bytes, 8-core, protocol: MESI)

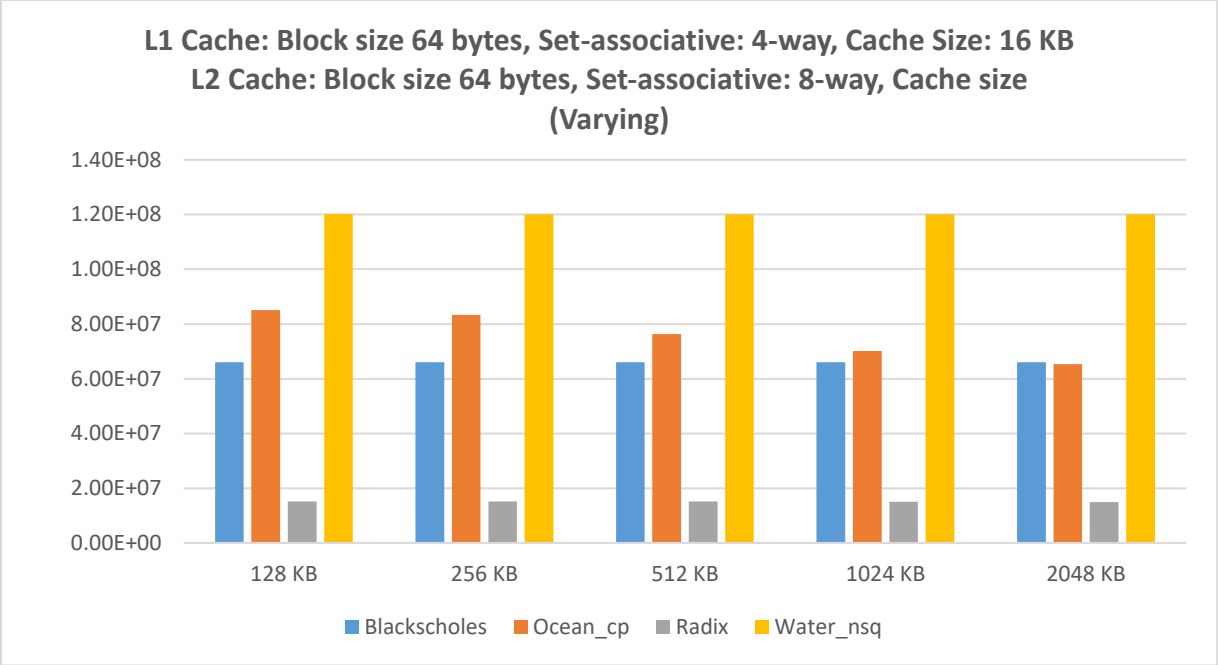

Figure 5. 7(e): # Bus Traffic (16 KB L1 cache, varying L2 cache, Block size 64 bytes, 8-core, protocol: MESI)

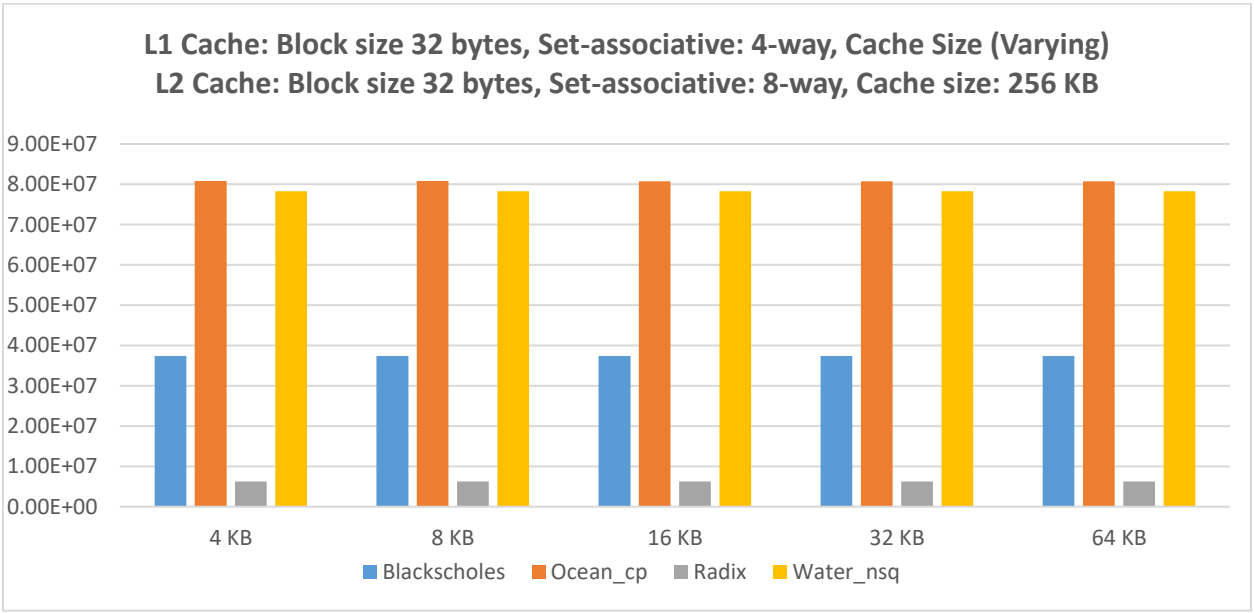

Figure 5. 7(f): # Bus Traffic (Varying L1 cache, 256 KB L2 cache, Block size 32 bytes, 8-core, protocol: FIREFLY)

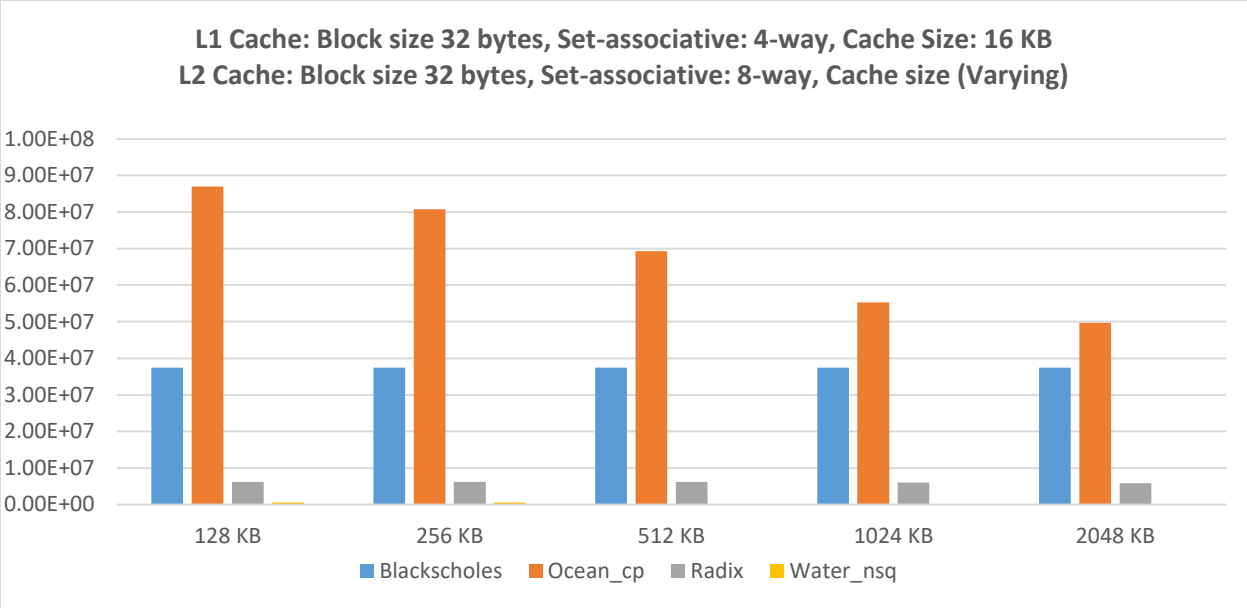

Figure 5. 7(g): # Bus Traffic (16 KB L1 cache, varying L2 cache, Block size 32 bytes, 8-core, protocol: FIREFLY)

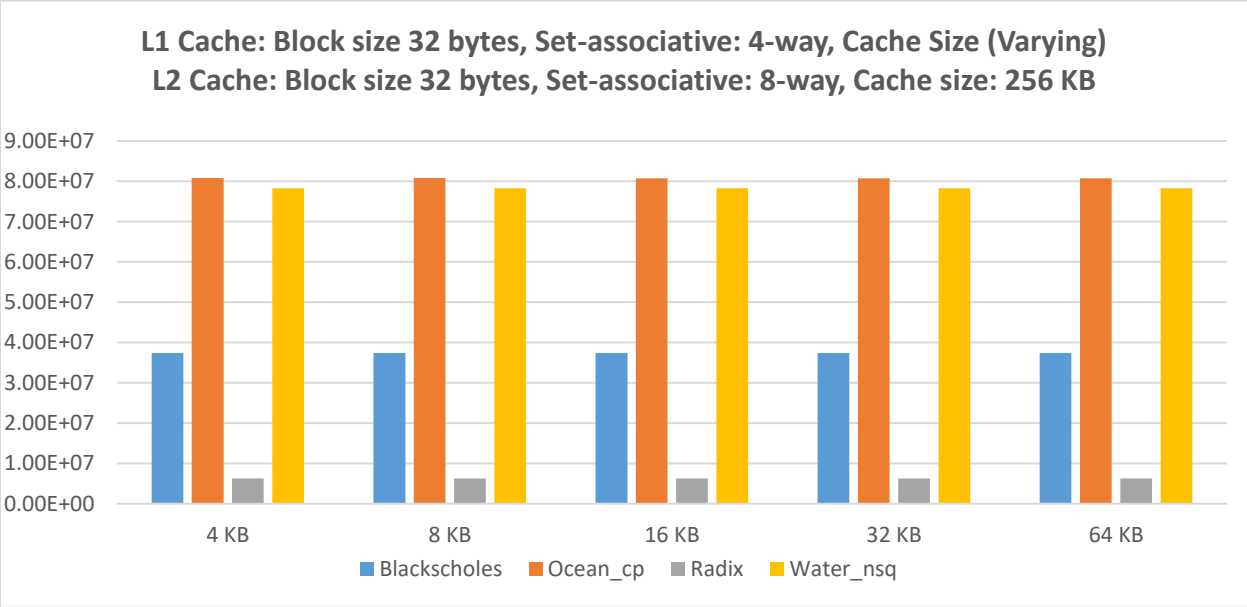

Figure 5. 7(h): # Bus Traffic (Varying L1 cache, 256 KB L2 cache, Block size 64 bytes, 8-core, protocol: FIREFLY)

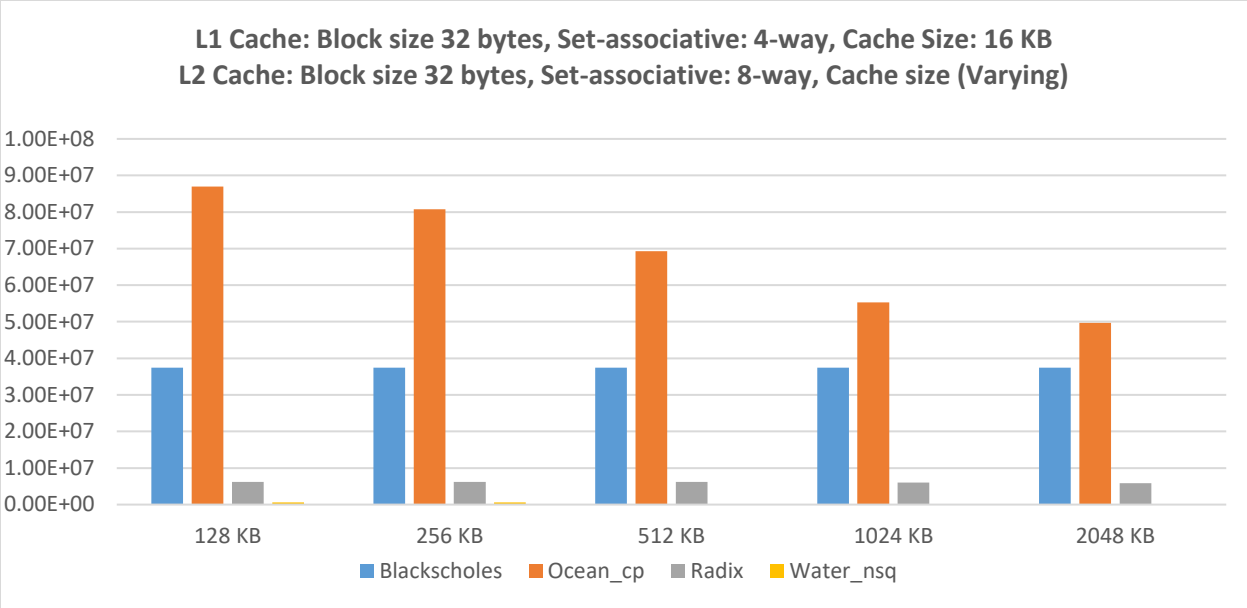

Figure 5. 7(i): # Bus Traffic (16 KB L1 cache, varying L2 cache, Block size 64 bytes, 8-core, protocol: FIREFLY)

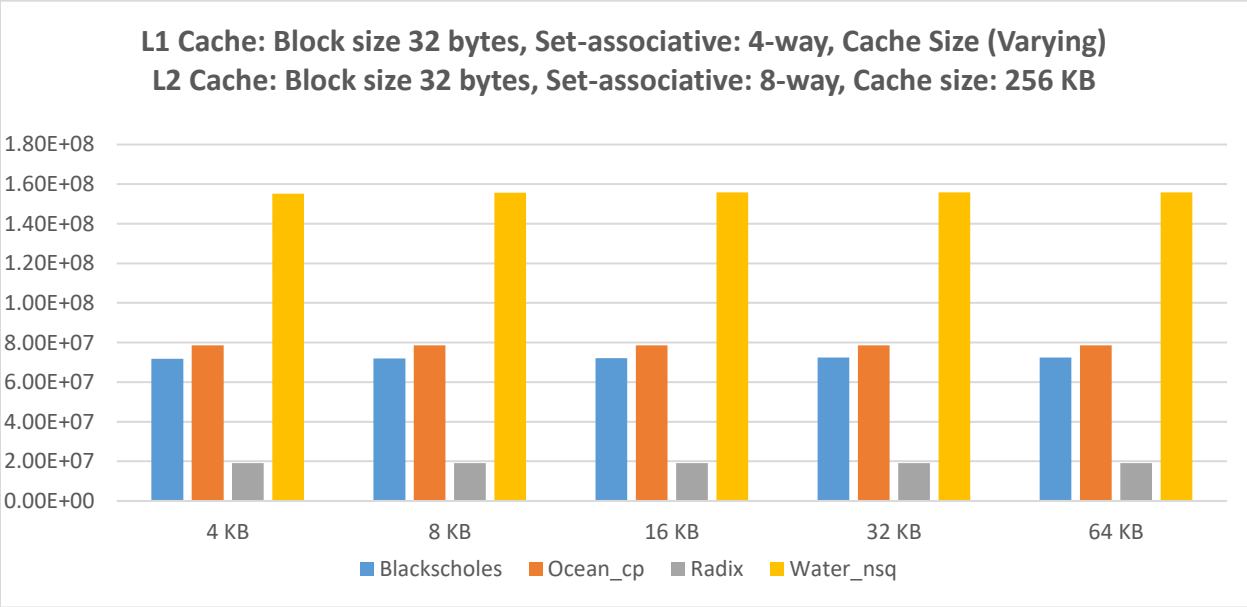

Figure 5. 7(j): # Bus Traffic (Varying L1 cache, 256 KB L2 cache, Block size 32 bytes, 16-core, protocol: MESI)

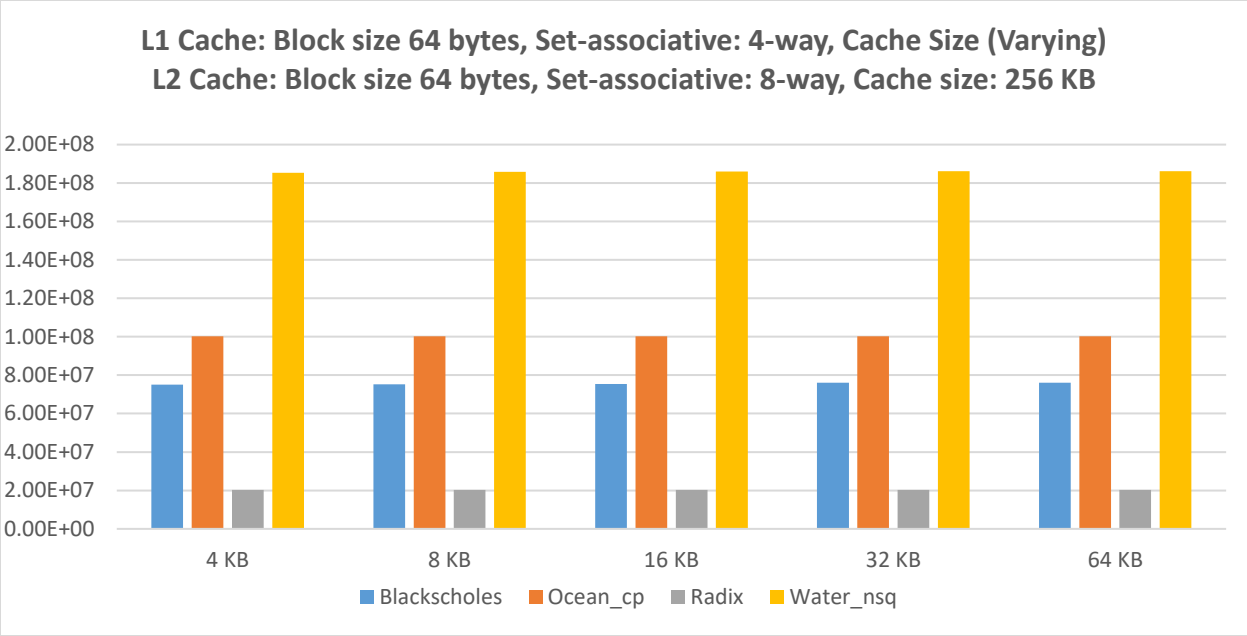

Figure 5. 7(k): # Bus Traffic (Varying L1 cache, 256 KB L2 cache, Block size 64 bytes, 16-core, protocol: MESI)

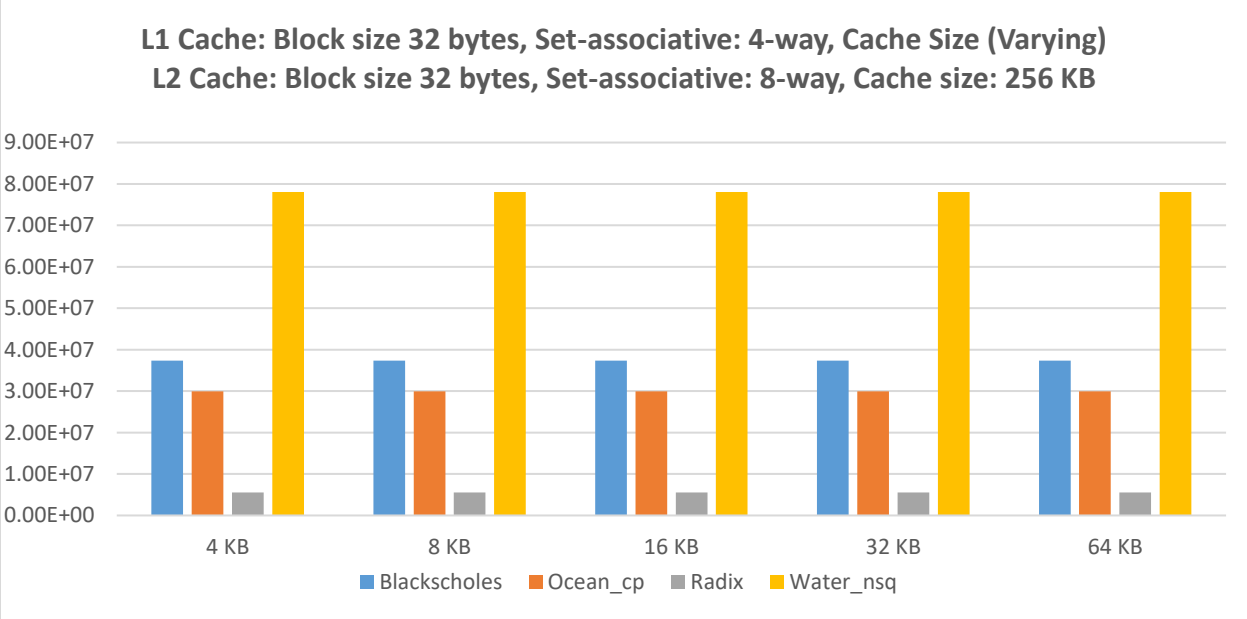

Figure 5. 7(l): # Bus Traffic (Varying L1 cache, 256 KB L2 cache, Block size 32 bytes, 16-core, protocol: FIREFLY)

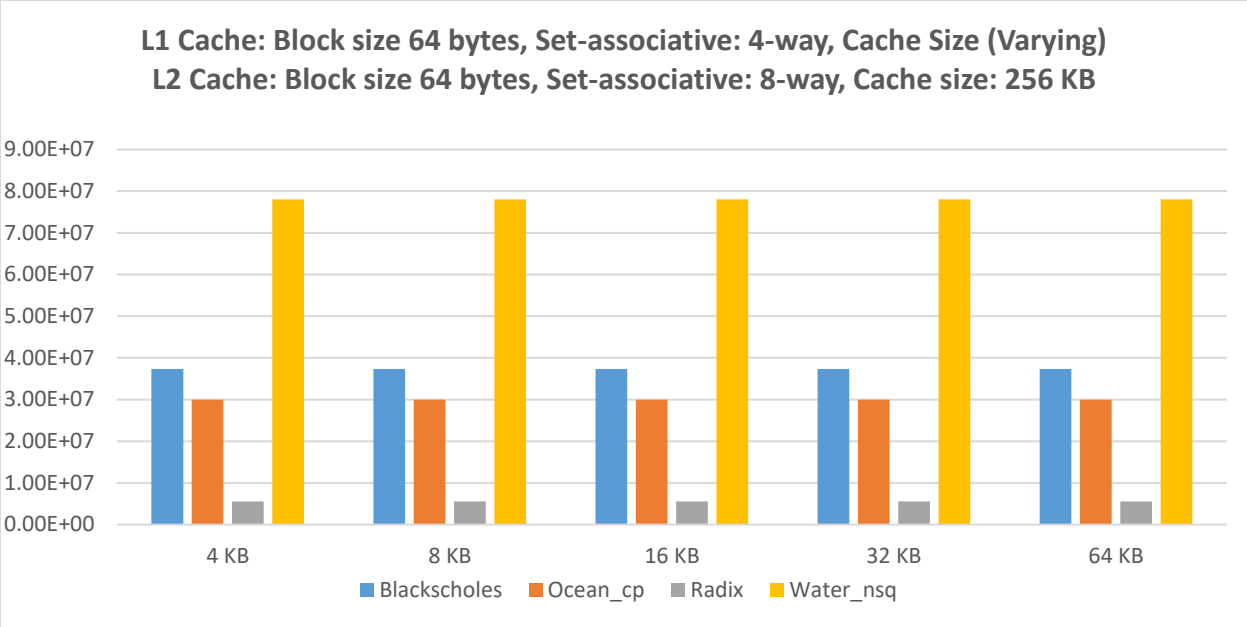

Figure 5. 7(m): # Bus Traffic (Varying L1 cache, 256 KB L2 cache, Block size 64 bytes, 16-core, protocol: FIREFLY)

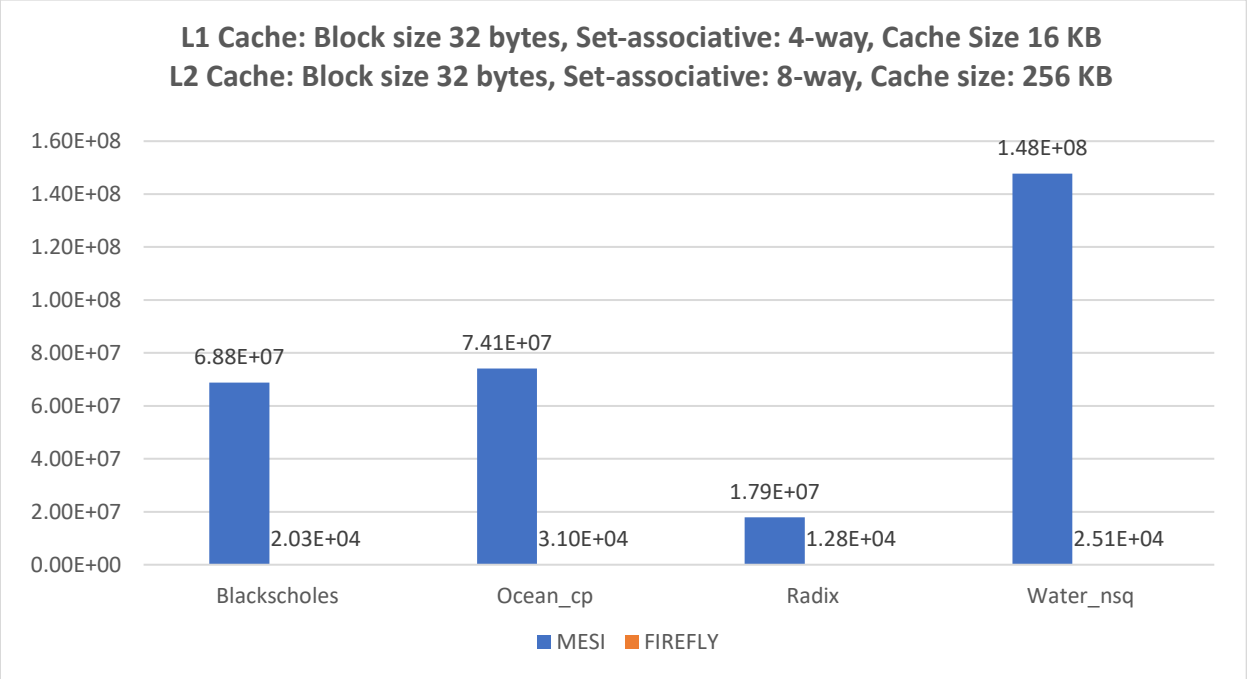

Figure 5.8(a): Comprison between MESI and FIREFLY Protocols regarding # Data Transfer among caches (16-core)

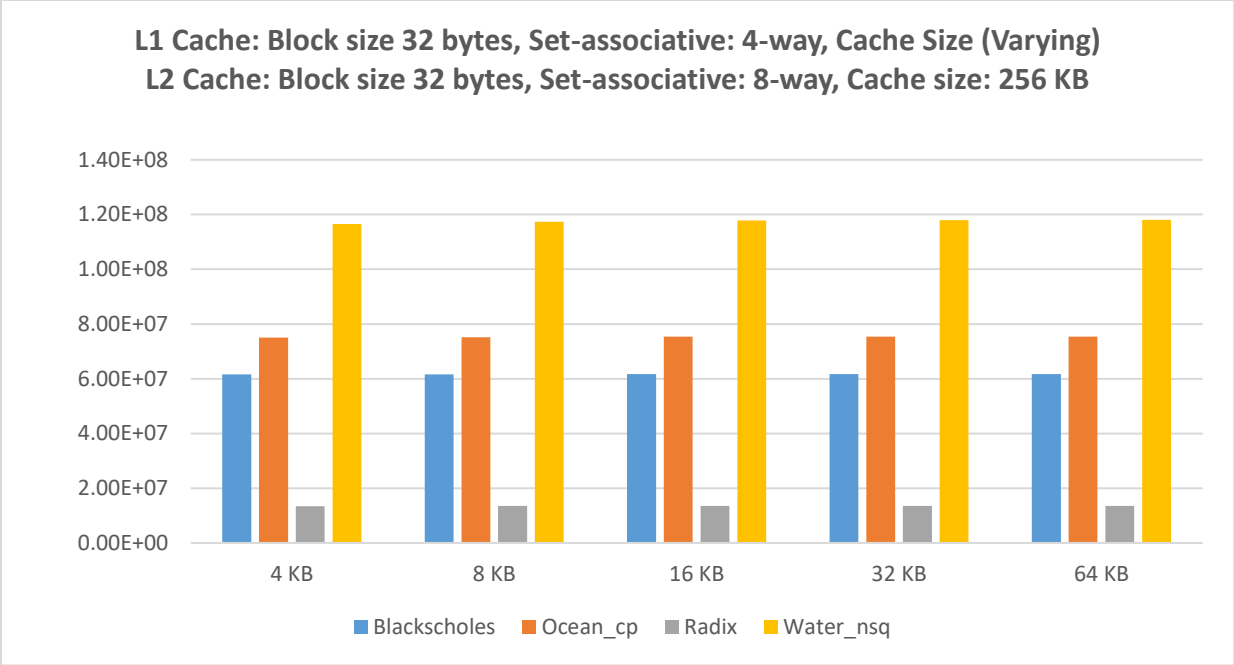

Figure 5. 8(b): # Data Transfer among caches (Varying L1 cache, 256 KB L2 cache, Block size 32 bytes, 8-core, protocol: MESI)

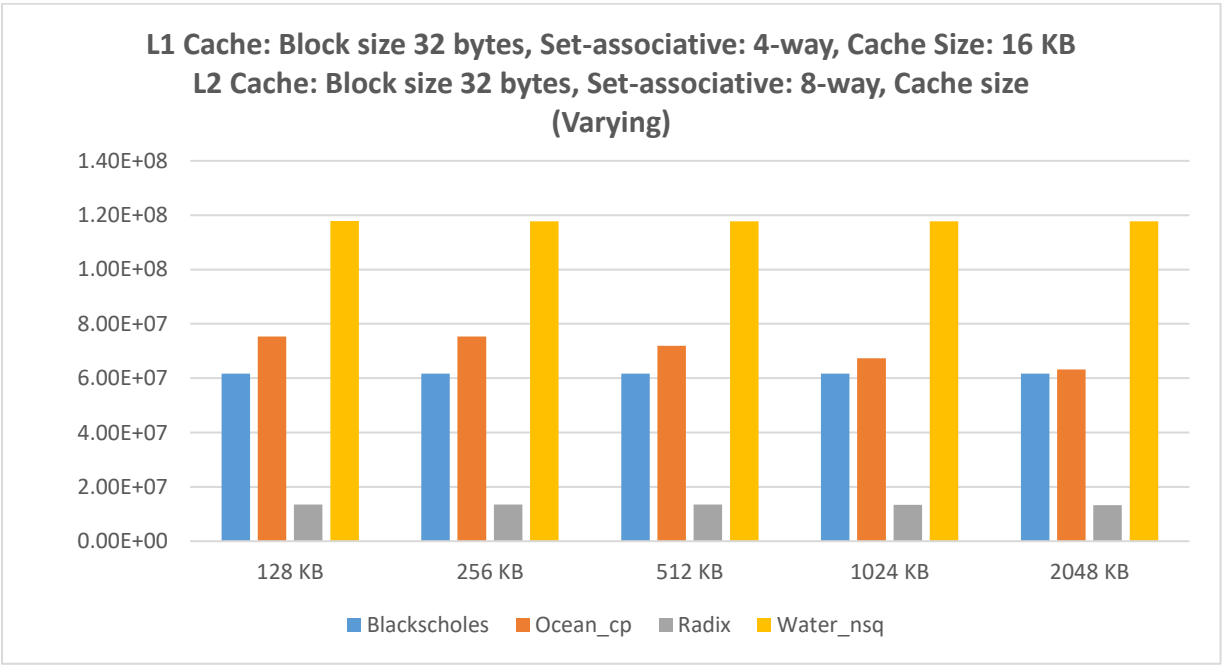

Figure 5. 8(c): # Data Transfer among caches (16 KB L1 cache, varying L2 cache, Block size 32 bytes, 8-core, protocol: MESI)

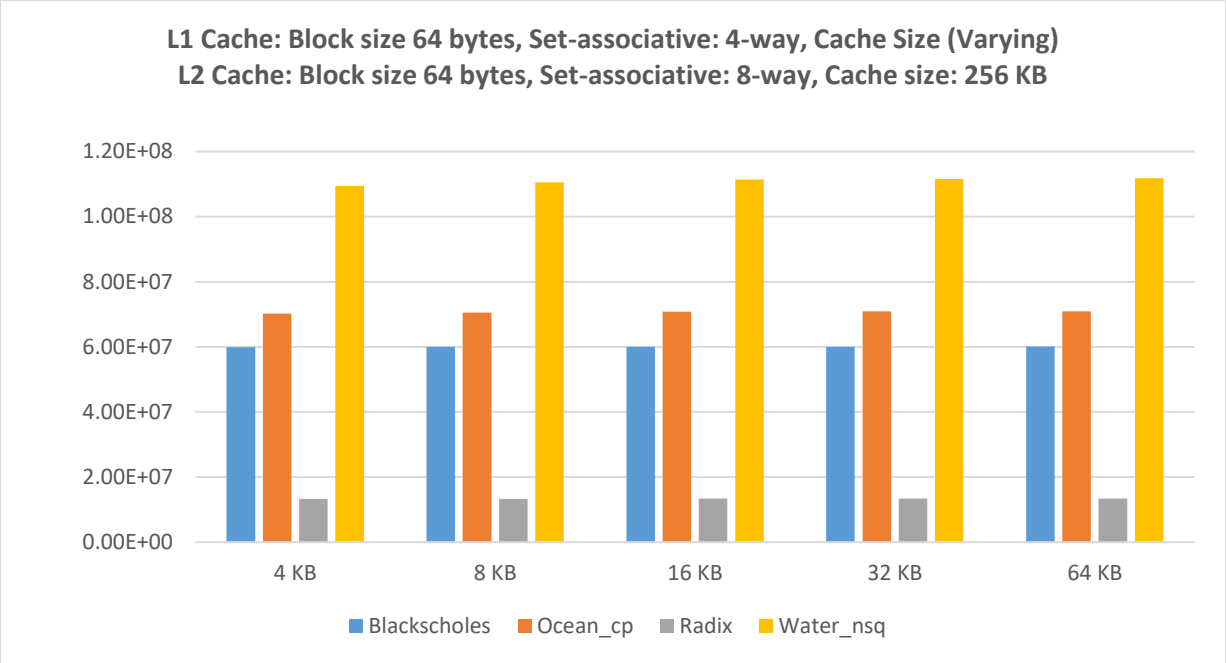

Figure 5. 8(d): # Data Transfer among caches (Varying L1 cache, 256 KB L2 cache, Block size 64 bytes, 8-core, protocol: MESI)

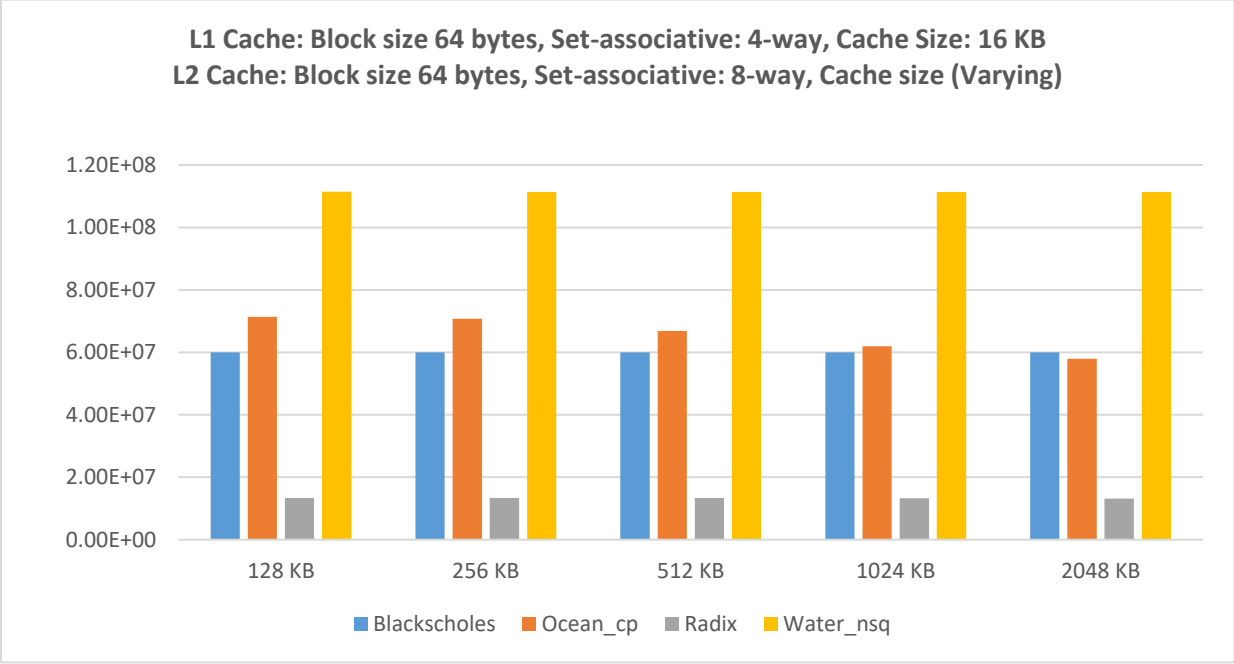

Figure 5. 8(e): # Data Transfer among caches (16 KB L1 cache, varying L2 cache, Block size 64 bytes, 8-core, protocol: MESI)

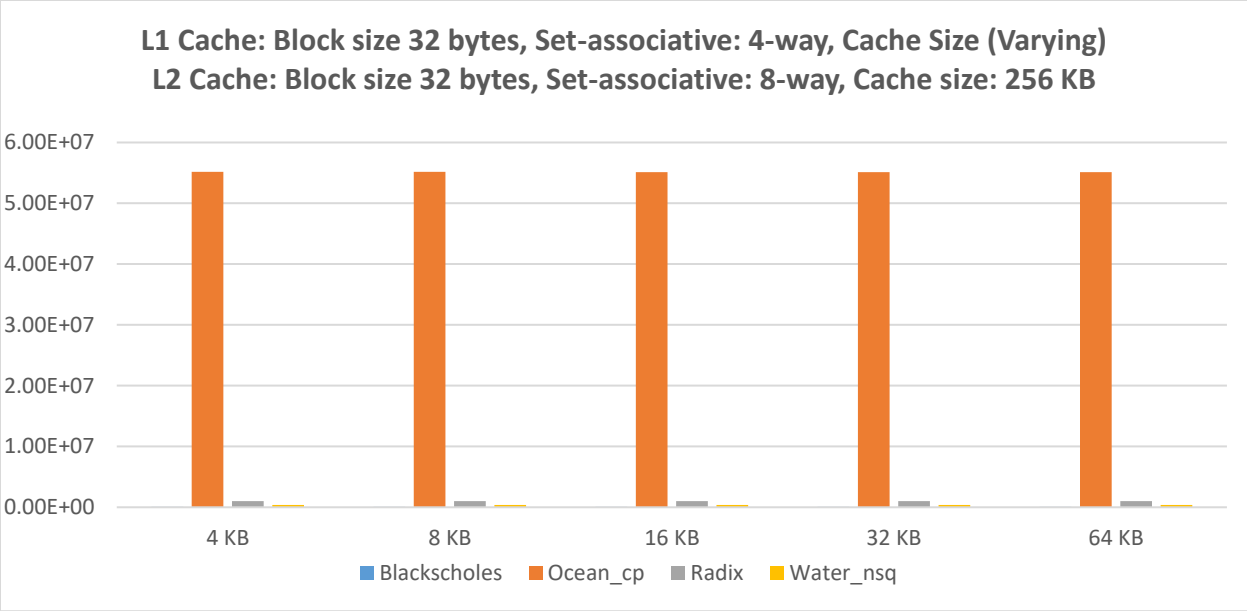

Figure 5. 8(f): # Data Transfer among caches (Varying L1 cache, 256 KB L2 cache, Block size 32 bytes, 8-core, protocol: FIREFLY)

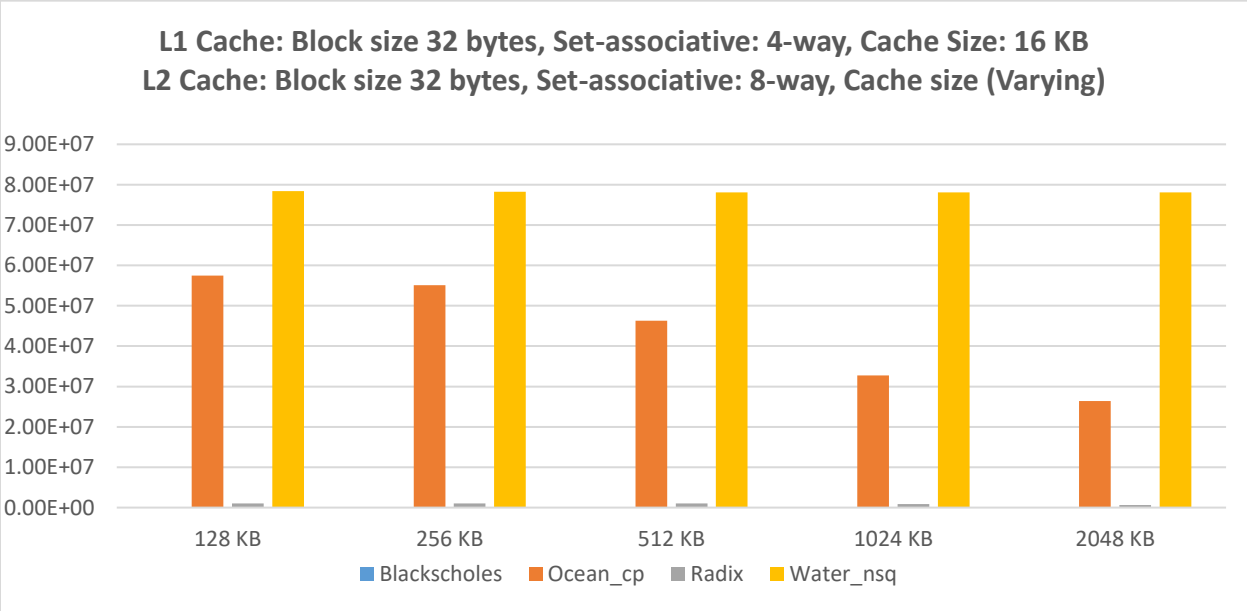

Figure 5. 8(g): # Data Transfer among caches (16 KB L1 cache, varying L2 cache, Block size 32 bytes, 8-core, protocol: FIREFLY)

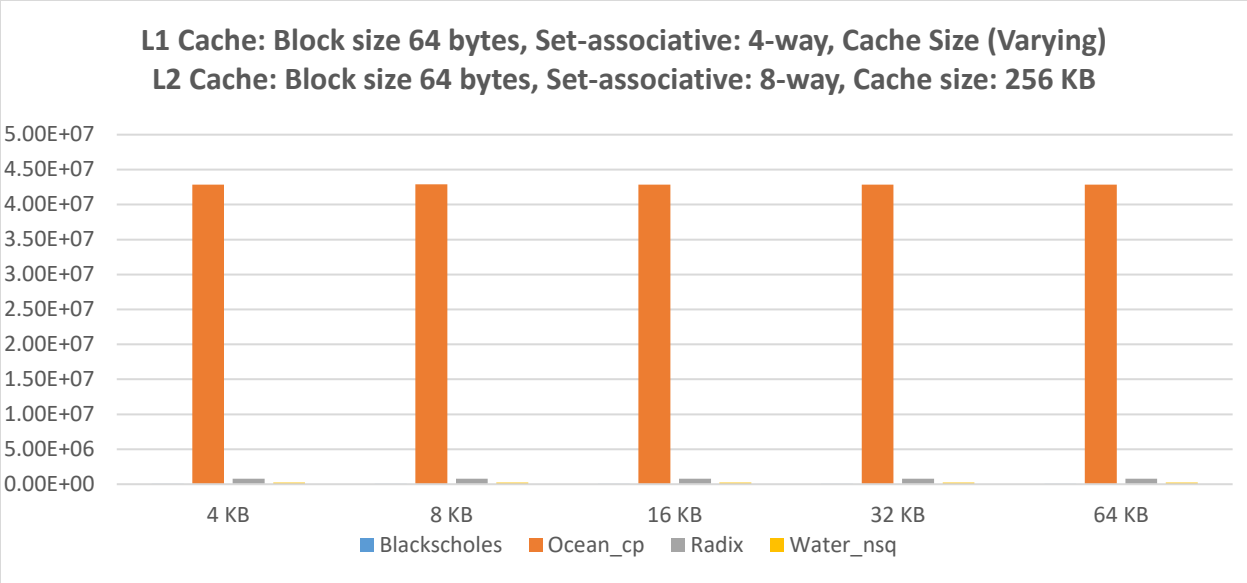

Figure 5. 8(h): # Data Transfer among caches (Varying L1 cache, 256 KB L2 cache, Block size 64 bytes, 8-core, protocol: FIREFLY)

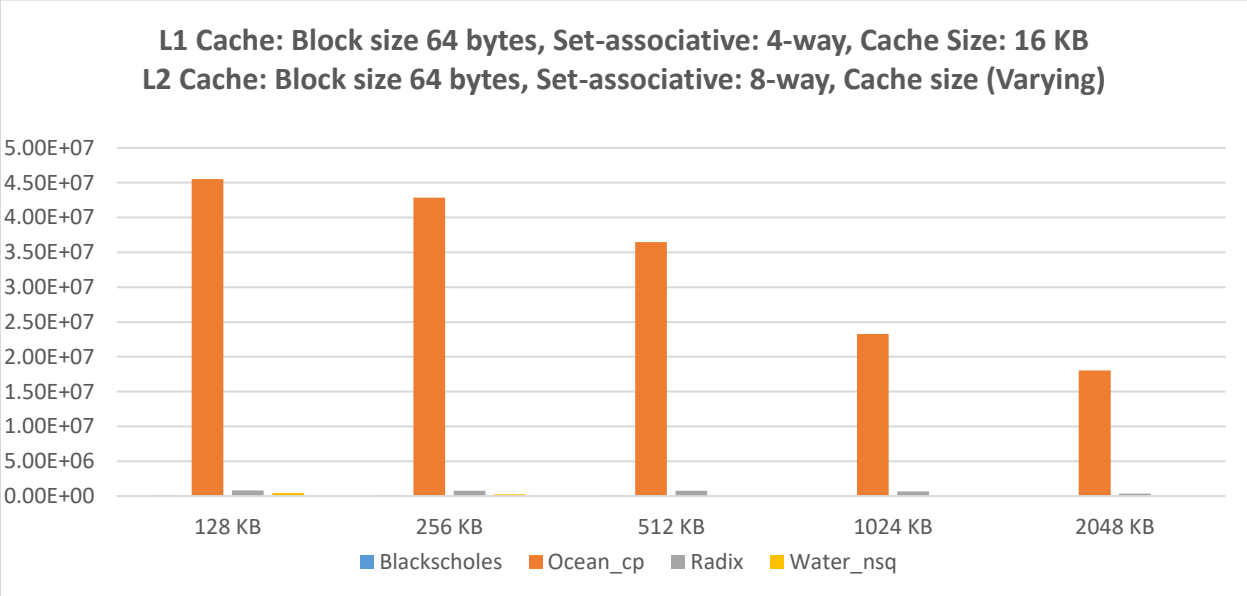

Figure 5. 8(i): # Data Transfer among caches (16 KB L1 cache, varying L2 cache, Block size 64 bytes, 8-core, protocol: FIREFLY)

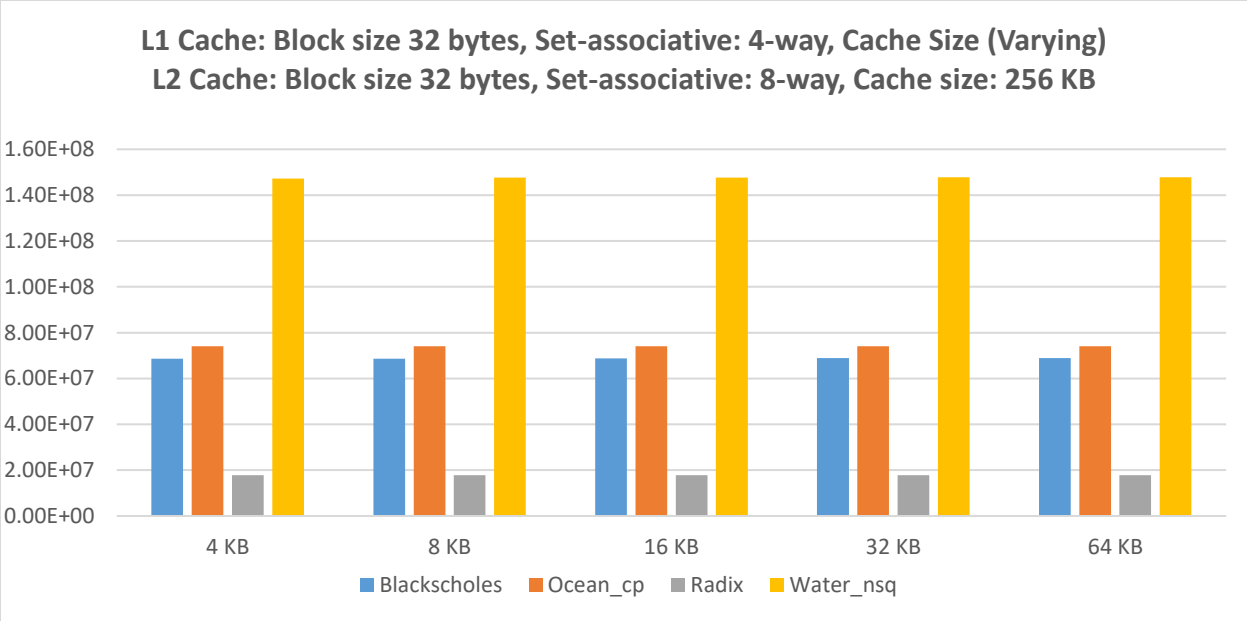

Figure 5. 8(j): # Data Transfer among caches (Varying L1 cache, 256 KB L2 cache, Block size 32 bytes, 16-core, protocol: MESI)

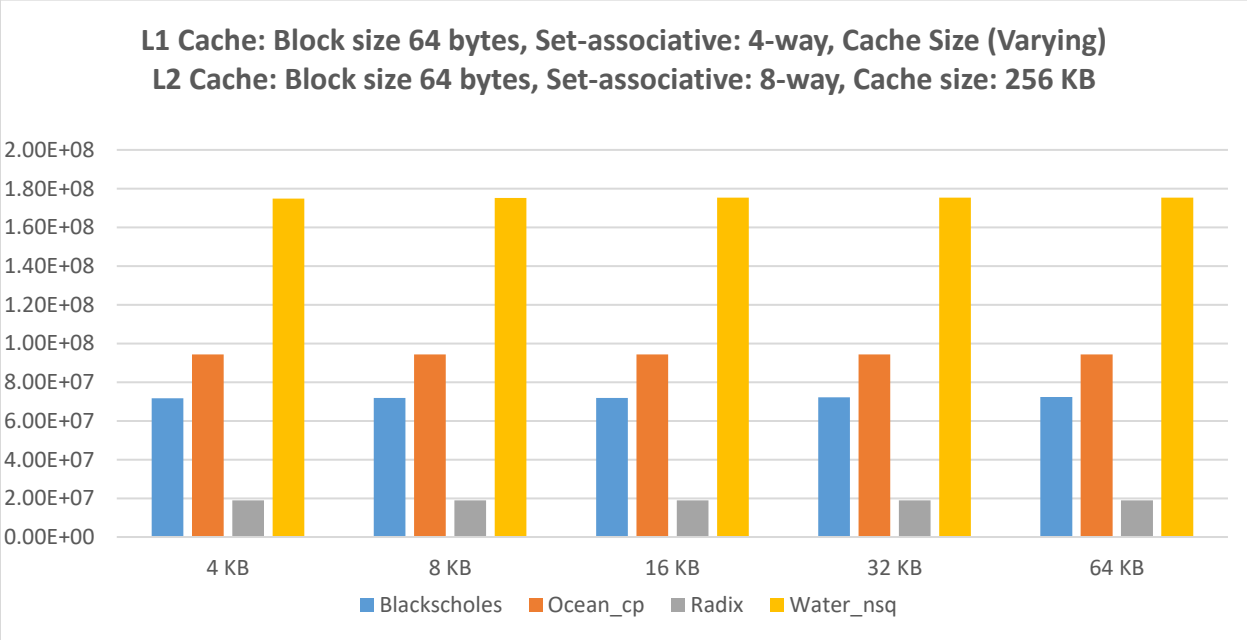

Figure 5. 8(k): # Data Transfer among caches (Varying L1 cache, 256 KB L2 cache, Block size 64 bytes, 16-core, protocol: MESI)

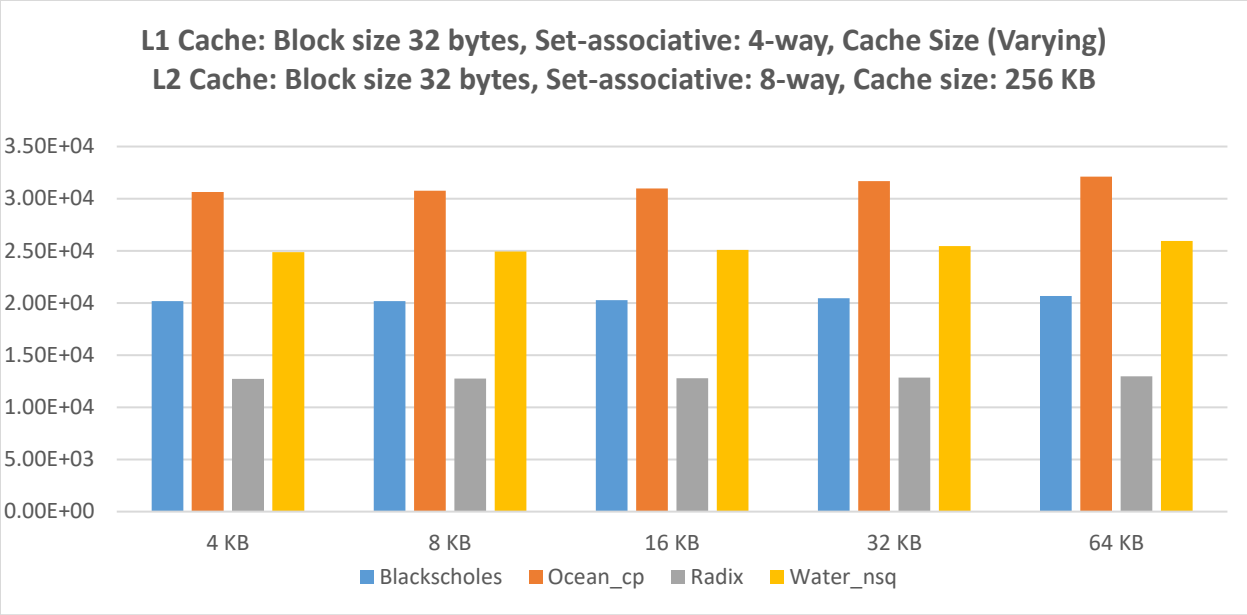

Figure 5. 8(l): # Data Transfer among caches (Varying L1 cache, 256 KB L2 cache, Block size 32 bytes, 16-core, protocol: FIREFLY)

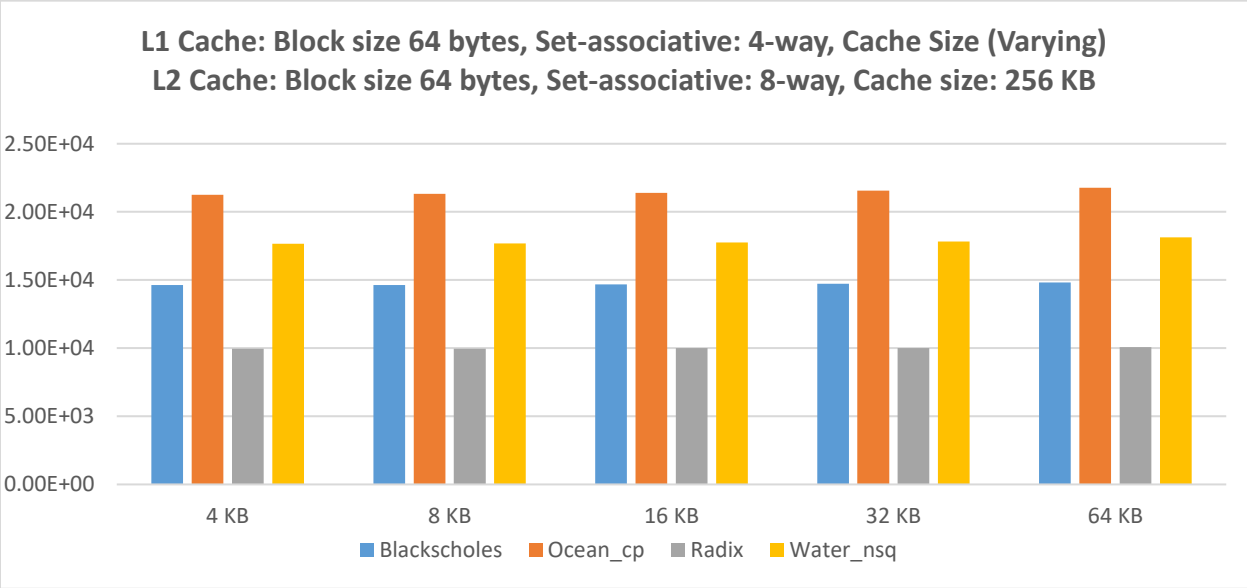

Figure 5. 8(m): # Data Transfer among caches (Varying L1 cache, 256 KB L2 cache, Block size 64 bytes, 16-core, protocol: FIREFLY)

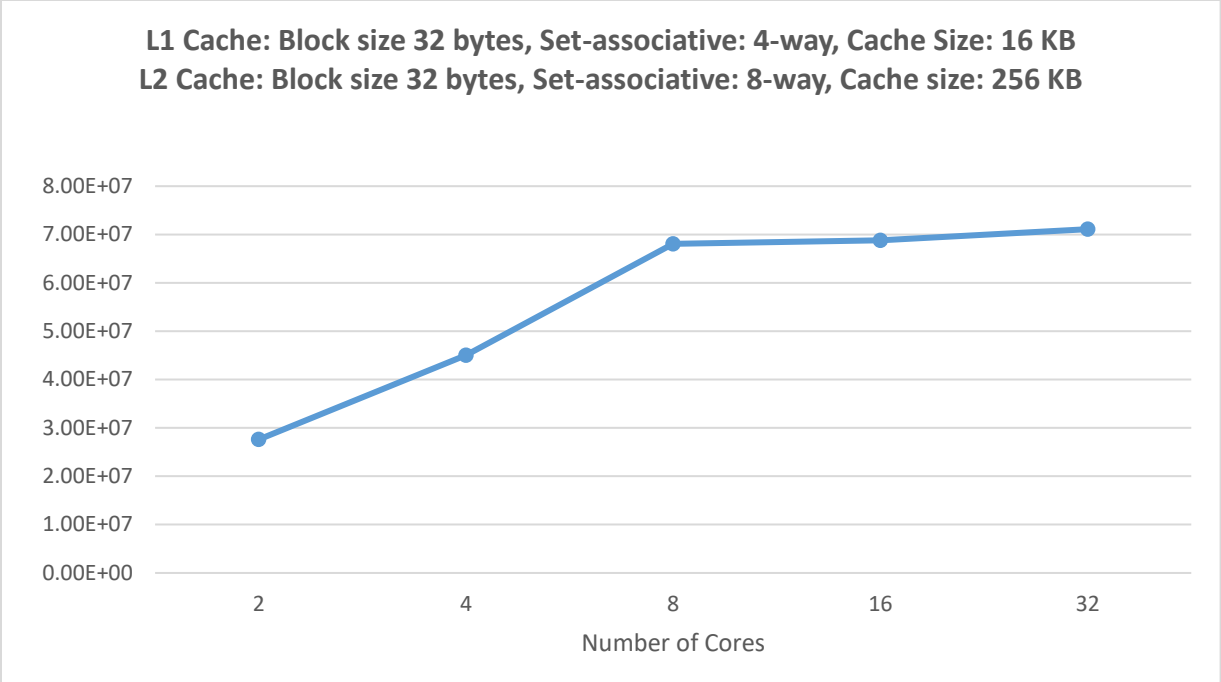

Figure 5.9(a): Effect of number of cores on Bus traffic (program: Blackscholes, protocol: MESI)

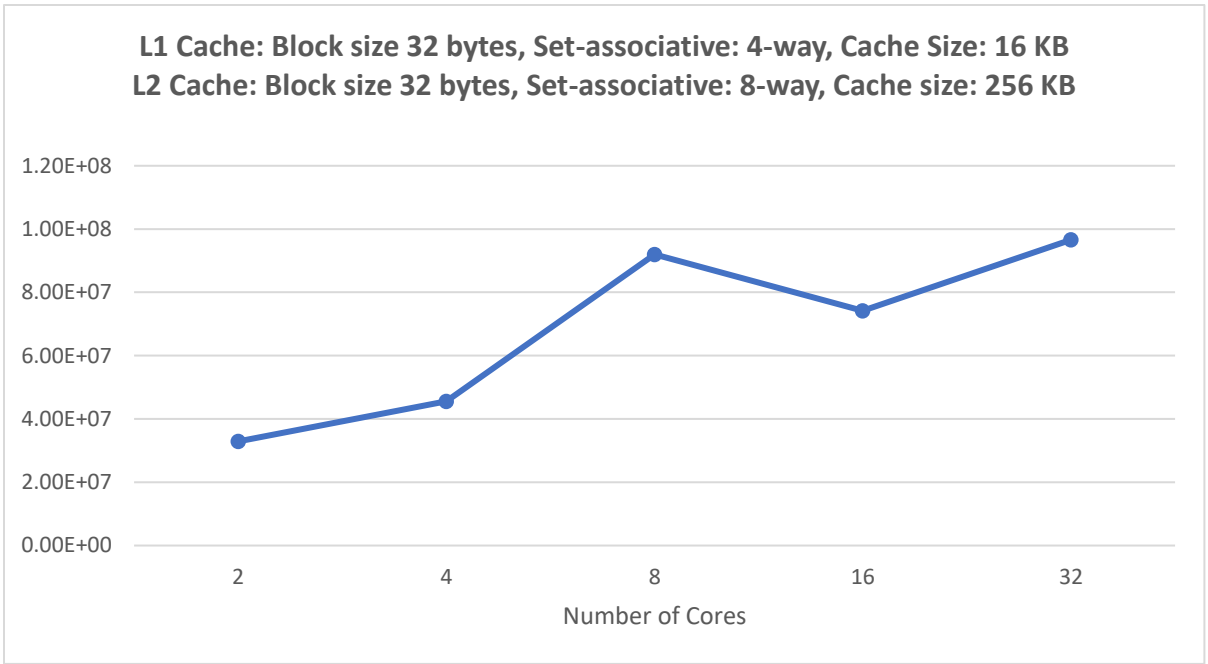

Figure 5.9(b): Effect of number of cores on Bus traffic (program: Ocean_cp, protocol: MESI)

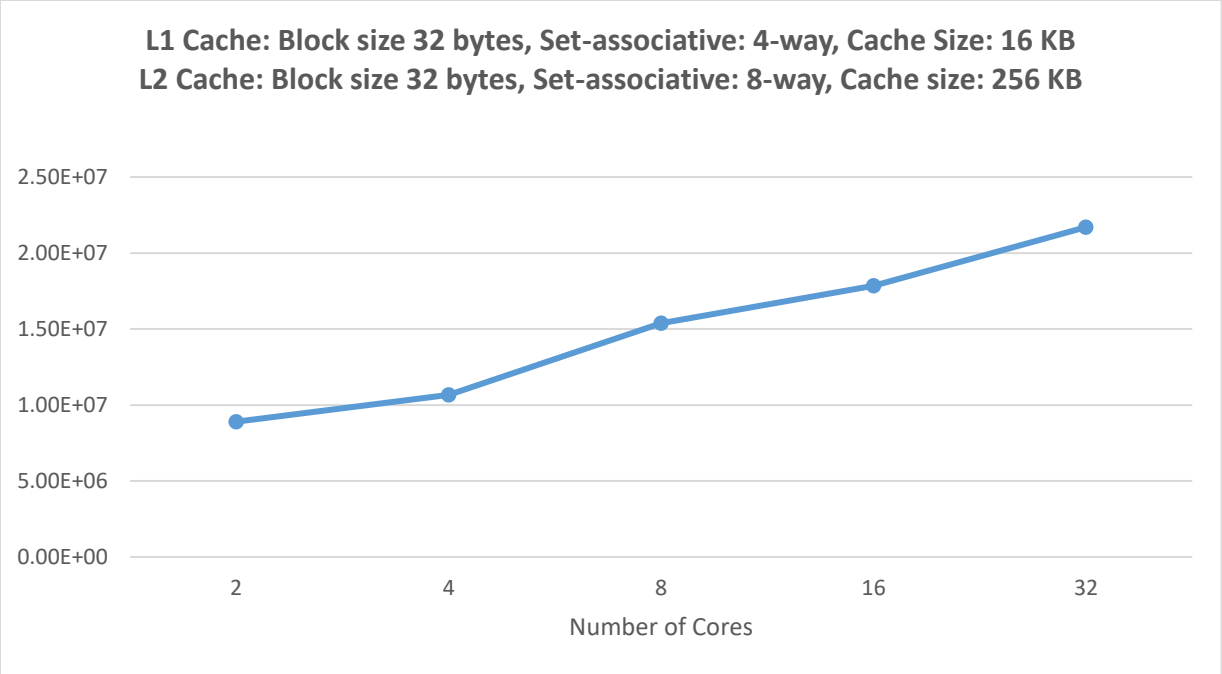

Figure 5.9(c): Effect of number of cores on Bus traffic (program: Radix, protocol: MESI)

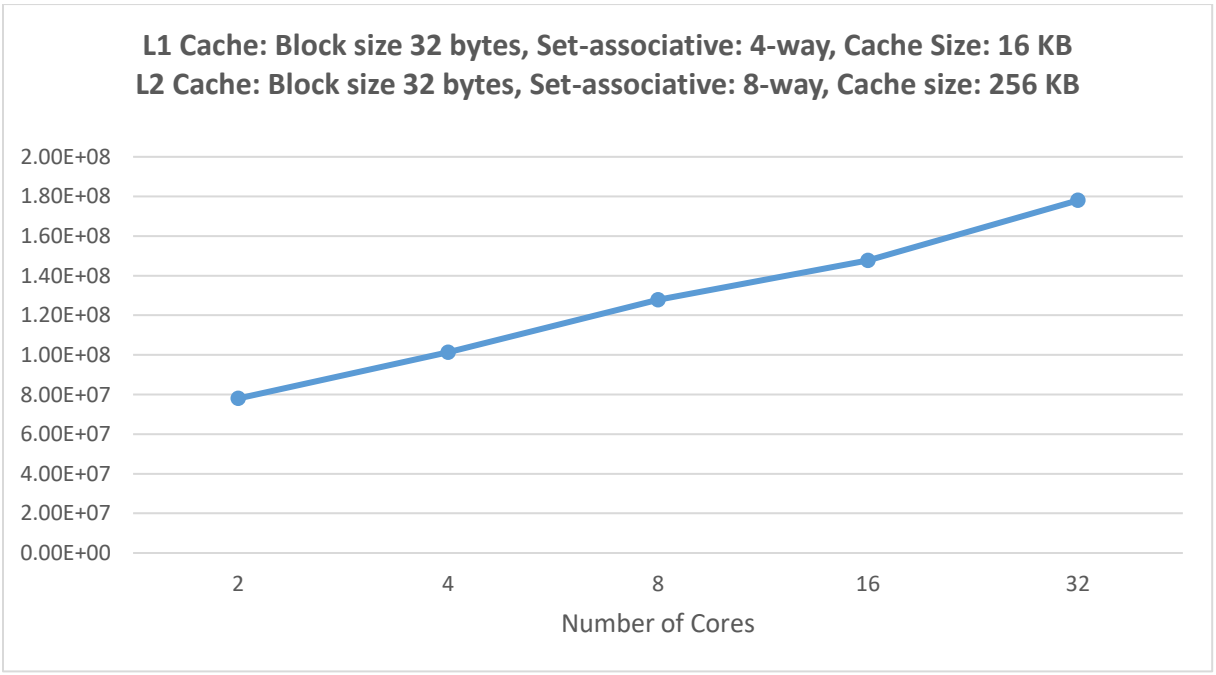

Figure 5.9(d): Effect of number of cores on Bus traffic (program: Water_nsq, protocol: MESI)

CHAPTER VI

CONCLUSION

An extended work environment is one of the main concerns of this proposed simulator. Our simulator grasps a fair portion of motivation from the propositions of other cache simulators. However, our vision is to improve the work perimeter from a single level to a multi-level cache for manycore systems to surpass the precursor. We expect that this simulator will be useful for creating various cache memory designs for manycore processors that are currently being developed. Several benchmark programs were utilized to simulate and assess manycore cache architectures in the experimental results. Our results demonstrate that the proposed cache simulator functions correctly. As the simulator is very customizable in nature, it is also expected that the practitioner will introduce new functions to it and widen the capability of the simulator. A comprehensive tutorial guide is going to be added with SIMNCORE source code, explaining the necessary installation and instrumentation procedures. The simulator will be released as an open-source program, so that it can be available for the academic community, hoping to be a helpful tool for students and researchers.

6.1 Future Areas for Development

While the proposed simulator shows major success in manycore cache memory simulation, there is a lot of scope for improving the simulator to widen its capabilities. The inclusion of L3 cache in SIMNCORE will be a substantial improvement to the simulation method. At present, the last level cache is ‘Private’ to each core. We intend to add the last level of cache as "Shared" among

the core in the near future. Simulations can be run on distant devices as well if SIMNCORE can be updated to a browser-based simulator. Even though SIMNCORE comes with a number of well-known cache coherence protocols, other coherence protocols (like DRAGON [37]) can be added. We also believe that a Python-based version of our suggested simulator can be developed with a better user-friendly interface.

REFERENCES

[1] Y. Xing, F. Liu, N. Xiao, Z. Chen, and Y. Lu, “Capability for Multi-Core and Many-Core Memory Systems: A Case-Study with Xeon Processors,” *IEEE Access*, vol. 7, pp. 47655–47662, 2019.

[2] C. J. Hughes, C. Kim, and Y. K. Chen, “Performance and energy implications of many-core caches for throughput computing,” *IEEE Micro*, vol. 30, no. 6, pp. 25–35, 2010.

[3] Xing Han, Jiang Jiang, Yuzhuo Fu, and Chang Wang, “ Reconfigurable Many-Core Processor with Cache Coherence”, 17th CCF Conference, NCCET 2013, July 2013.

[4] M. A. V. Rodriguez, J. M. S. Pérez, R. M. de la Montaña, and F. A. Z. Gallardo, “Simulation of Cache Memory Systems on Symmetric Multiprocessors with Educational Purposes,” *Proc. First Int. Congr. Qual. Tech. Educ. Innov.*, vol. 3, pp. 47–59, 2000.

[5] R. A. Uhlig and T. N. Mudge, “Trace-driven memory simulation: a survey,” *ACM Comput. Surv.*, vol. 29, no. 2, pp. 128–170, 1997.

[6] T. Austin, E. Larson, and D. Ernest, “SimpleScalar: An infrastructure for computer system modeling,” *Computer (Long. Beach. Calif).*, vol. 35, no. 2, pp. 59–67, 2002.

[7] M. A. Vega Rodriguez, J. M. Sanchez Perez, and J. A. Gomez Pulido, “An educational tool for testing caches on symmetric multiprocessors,” *Microprocessors and Microsystems*, vol. 25, no. 4, pp. 187–194, 2001.

[8] Rano Mal, and Yul Chu, "A Flexible Multi-core Functional Cache Simulator (FM-SIM)". *In Proceedings of the Summer Simulation Multi-Conference,* Article No.: 29, pp 1–12, 2017.

[9] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "PIN: A binary instrumentation tool for computer architecture research and education," *Proc. 2004 Work. Comput. Archit. Educ. WCAE 2004 - Held Conjunction with 31st Int. Symp. Comput. Archit. ISCA 2004*, 2004.

[10] D. Burger, T. M. Austin, and S. W. Keckler, "Recent extensions to the SimpleScalar Tool suite," *Perform. Eval. Rev.*, vol. 31, no. 4, pp. 4–7, 2004.

[11] A. Barai, Y. Arafa, A.-H. Badawy, G. Chennupati, N. Santhi, and S. Eidenbenz, *PPT-Multicore: performance prediction of OpenMP applications using reuse profiles and analytical modeling*, no. 0123456789. Springer US, 2021.

[12] J. Sharkey, "M-Sim: A Flexible, Multithreaded Architectural Simulation Environment," *Tech. Rep. CS-TR-05-DP01, Dep. Comput. Sci. State Univ. New York Binghamt.*, 2005.

[13] D. Madoń, E. Sanchez, and S. Monnier, "A study of a simultaneous multithreaded processor implementation," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 1685 LNCS, pp. 716–726, 1999.

[14] M. Nairag, "Enhancements and applications of the SimpleScalar simulator for undergraduate and graduate computer architecture education," *Proceedings WCAE 2000, Workshop on Computer Architecture Education*, Vancouver, BC, June 10, 2000. Published in *IEEE Computer Architecture Technical Committee Newsletter*, September 2000, pp. 34–41.

[15] R. Ubal, J. Sahuquillo, S. Petit, and P. Lopez, "Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors," pp. 62–68, 2008.

[16] T. Boiński and P. Czarnul, “Optimization of Data Assignment for Parallel Processing in a Hybrid Heterogeneous Environment Using Integer Linear Programming,” *Comput. J.*, no. April, 2021.

[17] N. Binkert *et al.*, “The gem5 simulator,” *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[18] A. A. Abudaqa, T. M. Al-Kharoubi, M. F. Mudawar, and A. Kobilica, “Simulation of ARM and x86 microprocessors using in-order and out-of-order CPU models with Gem5 simulator,” *2018 5th Int. Conf. Electr. Electron. Eng. ICEEE 2018*, pp. 317–322, 2018.

[19] B. K. Reddy, M. J. Walker, D. Balsamo, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett, “Empirical CPU power modelling and estimation in the gem5 simulator,” *2017 27th Int. Symp. Power Timing Model. Optim. Simulation, PATMOS 2017*, vol. 2017-Janua, pp. 1–8, 2017.

[20] M. Alian, U. Darbaz, G. Dozsa, S. Diestelhorst, D. Kim, and N. S. Kim, “Dist-gem5: Distributed simulation of computer clusters,” *ISPASS 2017 - IEEE Int. Symp. Perform. Anal. Syst. Softw.*, pp. 153–162, 2017.

[21] H. R. & S. M. A. P. Anuradha, “Computer Architecture Simulator,” *Int. J. Electr. Electron. Eng. Res.* , vol. 3, no. 1, pp. 297–302, 2013.

[22] O. Matthews *et al.*, “MosaicSim: A Lightweight, Modular Simulator for Heterogeneous Systems,” *Proc. - 2020 IEEE Int. Symp. Perform. Anal. Syst. Software, ISPASS 2020*, no. February, pp. 136–148, 2020.

[23] A. Koshiba, T. Hirofuchi, R. Takano, and M. Namiki, “A Software-based NVM Emulator Supporting Read/Write Asymmetric Latencies,” *IEICE Trans. Inf. Syst.*, vol. E102D, no. 12, pp. 2377–2388, 2019.

[24] C. Fanning and S. Garcia, “Below C level: A student-centered x86-64 simulator,” *Annu. Conf. Innov. Technol. Comput. Sci. Educ. ITiCSE*, pp. 381–387, 2019.

[25] K. Vollmar and P. Sanderson, “MARS: An education-oriented MIPS assembly language simulator,” *Proc. Thirty-Seventh SIGCSE Tech. Symp. Comput. Sci. Educ.*, pp. 239–243, 2007.

[26] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC benchmark suite: Characterization and architectural implications,” *Parallel Archit. Compil. Tech. - Conf. Proceedings, PACT*, no. May 2014, pp. 72–81, 2008.

[27] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The splash-2 programs: Characterization and methodological considerations,” *ACM SIGARCH computer architecture news*, vol. 23, no. 2, pp. 24–36, 1995.

[28] A. Jaleel, R. S. Cohn, C.Luk, and B. Jacob. CMP$im: A Binary Instrumentation Approach to Modeling Memory Behavior of Workloads on CMPs", Technical Report - *UMDSCA*-2006-01

[29] J. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach*, Waltham, MA, USA: Morgan Kaufmann, Elsevier, 2012.

[30] J. Gómez-Luna, E. Herruzo, and J. I. Benavides, “MESI Cache Coherence Simulator for Teaching Purposes,” *CLEI Electron. J.*, vol. 12, no. 1, 2009.

[31] N. B. Mallya, G. Patil, and B. Raveendran, “Simulation based Performance Study of Cache Coherence Protocols,” *Proc. - 2015 IEEE Int. Symp. Nanoelectron. Inf. Syst. iNIS 2015*, no. November 2019, pp. 125–130, 2016.

[32] W. Lin, X.C Ye, F.L. Song et al. “Using write mask to support hybrid write-back and write-through cache policy on Many-core architectures”, Chinese Journal of Computers, Vol 32, No.11 , 1918-1927, Nov. 2008.

[33] Hashemi, B., "Simulation and Evaluation Snoopy Cache Coherence Protocols with Update Strategy in Shared Memory Multiprocessor Systems," *Parallel and Distributed Processing with Applications Workshops (ISPAW)*, 2011 Ninth IEEE International Symposium on , pp.256,259, 26-28 May 2011.

[34] R. Mal, "A Simple Multi-Core Functional Cache Design Simulator," M.S. thesis, Dept. Elect. Eng., UT Rio Grande Valley, Edinburg, TX, 2017.

[35] M. Elver and V. Nagarajan, TSO-CC: Consistency directed cache coherence for TSO, *Proceedings of 2014 IEEE 20th International Symposium on high Performance Computer Architecture (HPCA)*, Orlando, Florida, U.S.A., 2014.

[36] "Memory traces," arco.unex.es http://arco.unex.es/smpcache/

[37] P. Stenstrom, “A Survey of Cache Coherence Schemes for Multiprocessors”, *Journal of Computer*, Vol. 23, Issue 6, June 1990, pp. 12-24.

APPENDIX A

APPENDIX A

SIMNCORE Manual

In this documentation, we have shown how to use SIMNCORE, and how to prepare Trace files with Pintool.

Section 1: How to Install SIMNCORE

Section 2: How to Perform Simulation with SIMNCORE

Section 3: Trace File Preparation with Pintool for Single-Core and Many-Core Systems

Section 4: Function Blocks in SIMNCORE

Section 5: How to Implement a New Replacement Policy or New Cache Design Scheme in SIMNCORE

Section 1: How to Install SIMNCORE

Step 1: Download the source files for SIMNCORE. Locate all the source files in the folder. The source files are as follows (Figure 1): 1) simncore.cpp (contains functions for executing different protocols, selected by users), 2) simncorefunc.h (contains all the function blocks that read parameters, create virtual cache memory, check cache hit/miss, etc.), 3) variable.h (contains all the global variables for the source code), 4) header.h (contains all the necessary libraries), and 5) build.h (contains all the cache parameters, structure for the L1 and L2 cache, etc.)

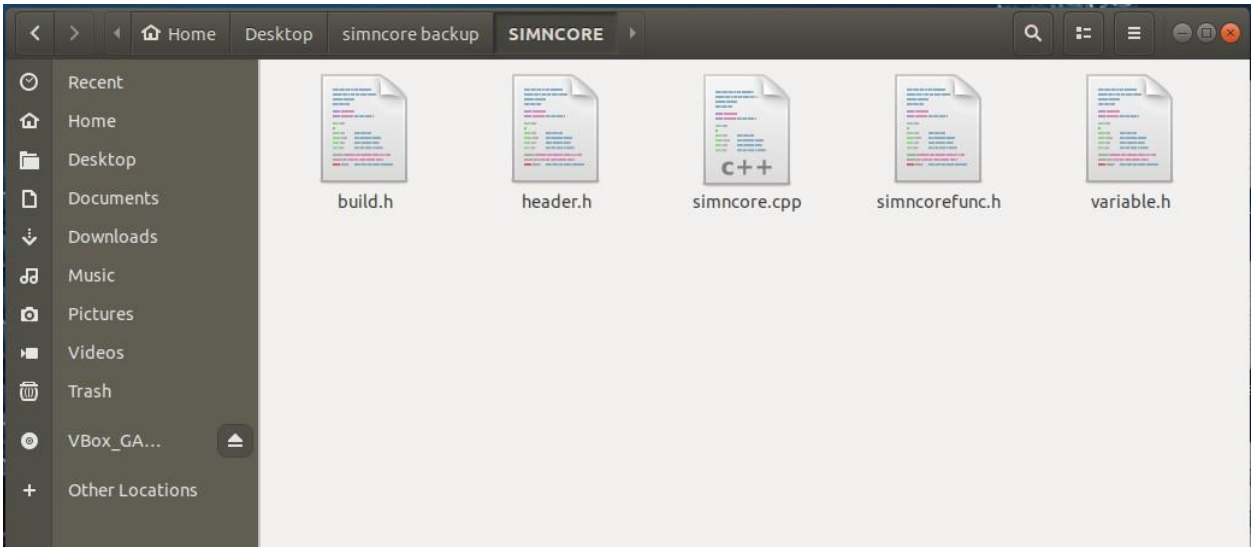

Figure A1: Source Files for SIMNCORE

Step 2: Download trace files from the SIMNCORE online resource or prepare trace files for SIMNCORE (see Section 3). Once trace files are collected, place them in the same folder with the SIMNCORE source files. We can save trace files with any name. But to identify them properly, it is better to put the core information in the file name. For example, in Figure 2, we have collected two trace files. The first trace file is FFT_SC, which is prepared for Single-core (SC) simulation, and the second trace file is Radix_MC16, which is prepared for Manycore (MC, 16 indicates the number of cores) simulation.

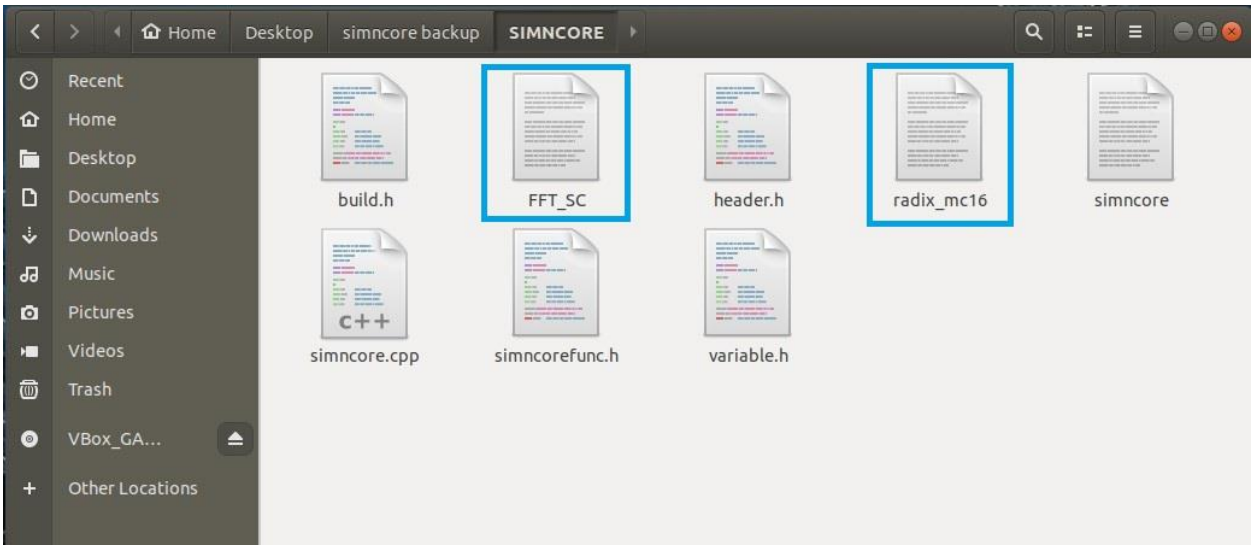

Figure A2: Trace Files for Simulation with SIMNCORE

Section 2: How to Perform Simulation with SIMNCORE

Step 1: In the SIMNCORE folder, execute the command 'c++ simncore.cpp –o simncore' in the terminal (Figure 3). This should create the application named 'simncore'.

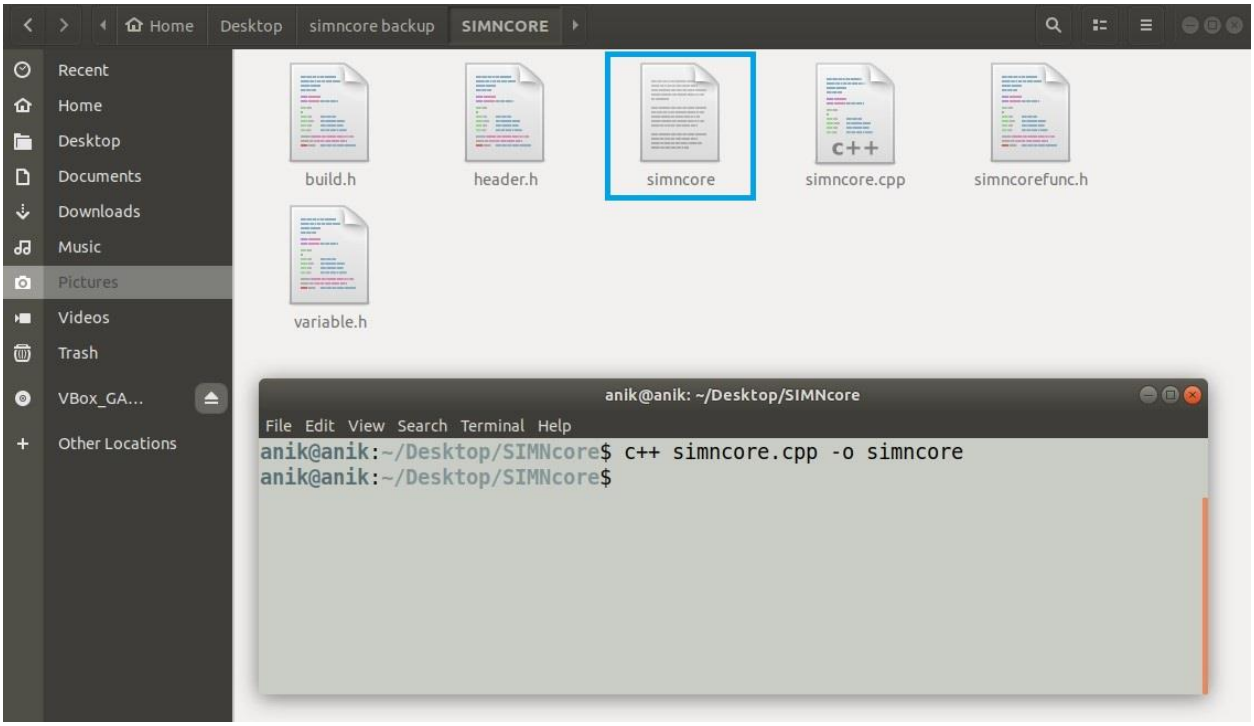

Figure A3: Installation Command

Step 2: Type command line for simulation in the terminal. The command line format is shown here,

$ ./simncore -l1 e:f:g:h -l2 r:s:u:v -CX -PY -WZ <trace file>

Here,

e = cache size of L1 cache (in bits); f = block size of L1 cache (in bits); g = associativity of L1 cache (in bits); and h = replacement policy of L1 cache.

Similarly,

r = cache size of L2 cache (in bits); s = block size of L2 cache (in bits); u = associativity of L2 cache (in bits); and v = replacement policy of L2 cache.

In CX, X denotes the number of cores in processor. (For example, X=1, for single-core; X=16 for 16 cores).

In PY, Y denotes the coherence protocols. (For example, Y = 0 represents a single-core protocol or no protocol; Y = 1 represents the MESI protocol; Y = 2 represents the SNOOPY protocol; Y = 3 represents the Firefly protocol; and Y = 4 represents any newly added protocol.)

WZ stands for writing policies. (For example, Z=1 is Write-back, and Z=2 is Write-through)

In this example, a command line is shown in the terminal (Figure 8).

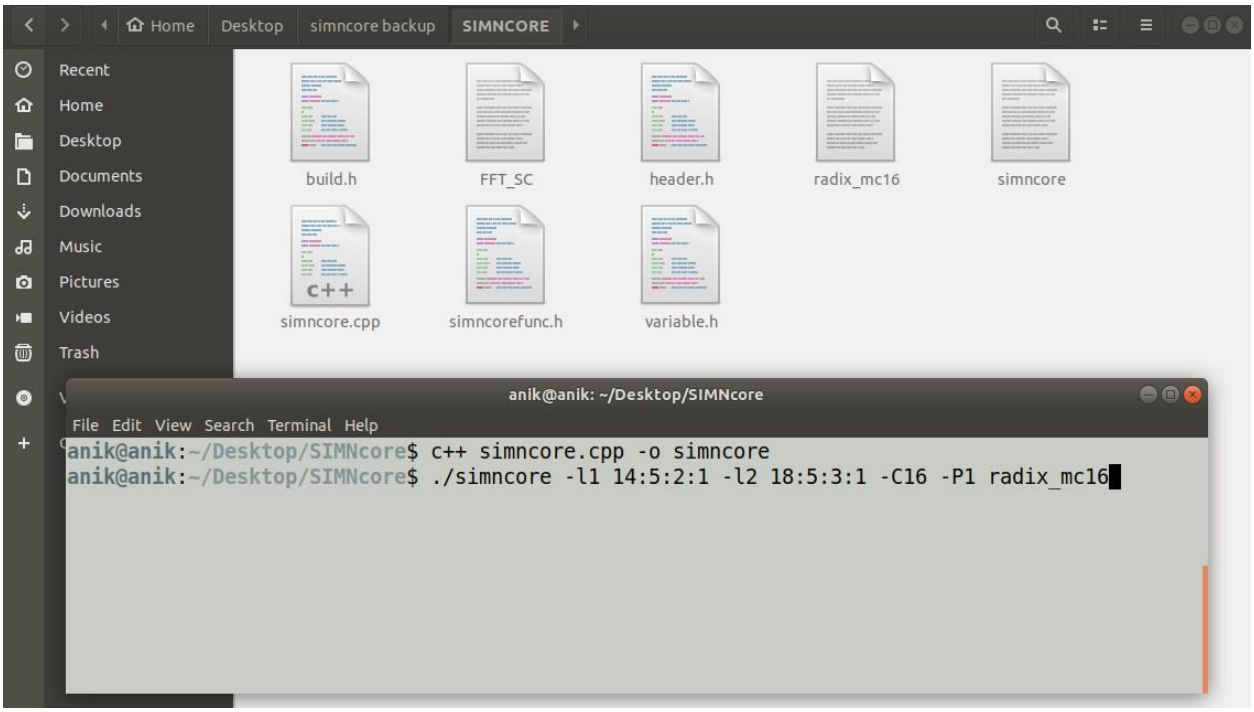

Figure A4: Command Line for Simulation with L1 and L2 Cache Parameters

In Figure 4, L1 cache parameters are,

Cache size: 2^14 =16 KB

Block size: 2^5 = 32 bytes

Set-associativity: 2^2= 4 way

Replacement policy: 1 = LRU

Similarly, L2 cache parameters are,

Cache size: 2^18 =256 KB

Block size: 2^5 = 32 bytes

Set-associativity: 2^3= 8 way

Replacement policy: 1 = LRU

C16 = 16-core processor

P1 = MESI coherence protocol

'radix_mc16' is the trace file for the simulation.

If the processor has only one level of cache (L1 cache), then the command line should contain the parameters for L1 cache only (Figure 5). Put '0' in the L2 cache parameter section.

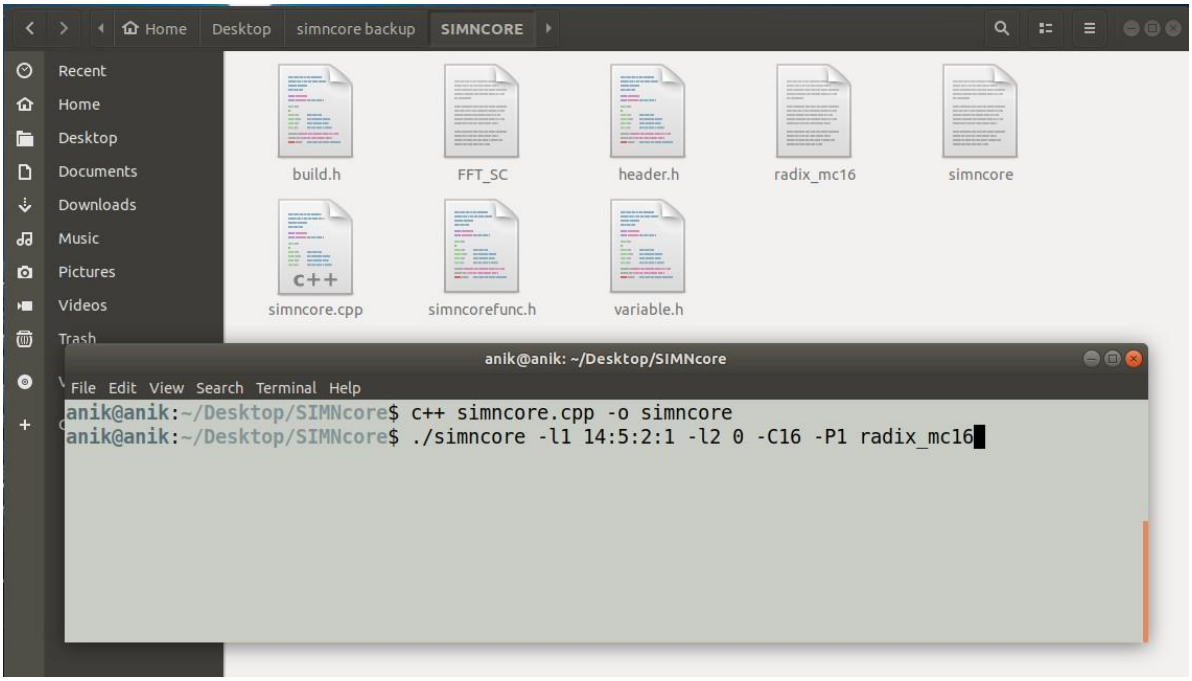

Figure A5: Command Line for Simulation with L1 Cache Parameters

In Figure 6 and Figure 7, we have shown the simulation output of two different input trace files.

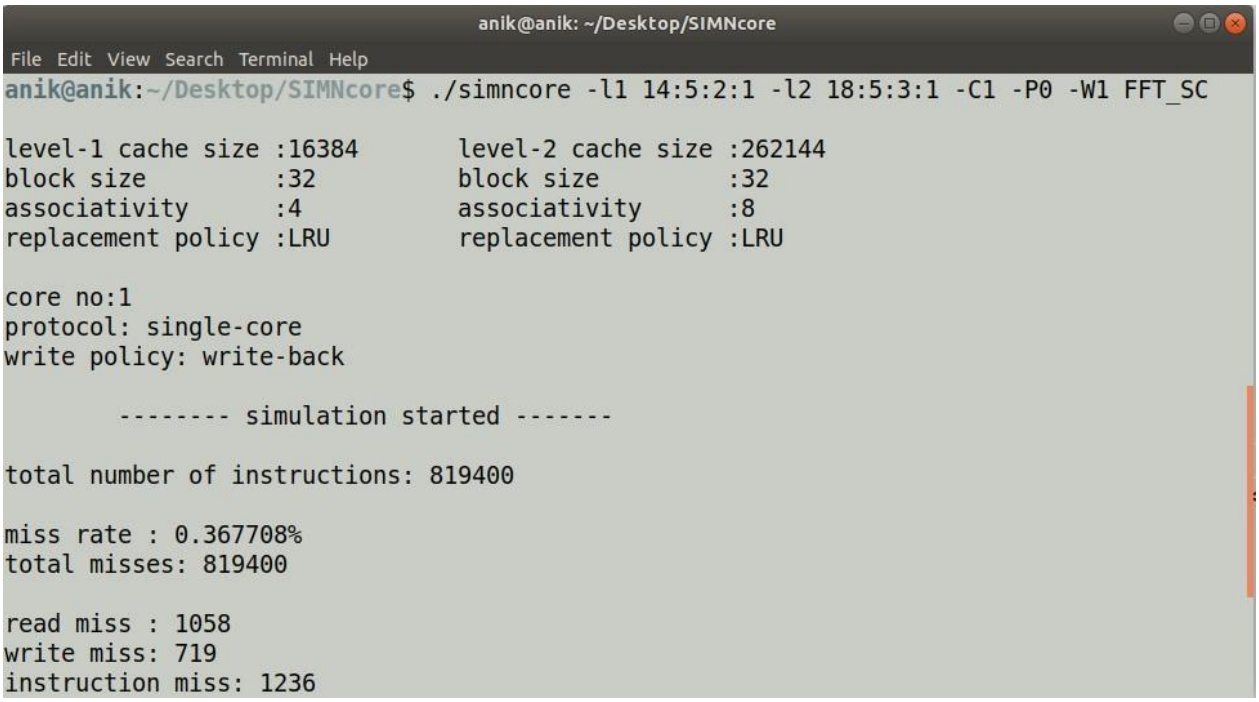

Figure A6: Single-core cache memory simulation (program: FFT)

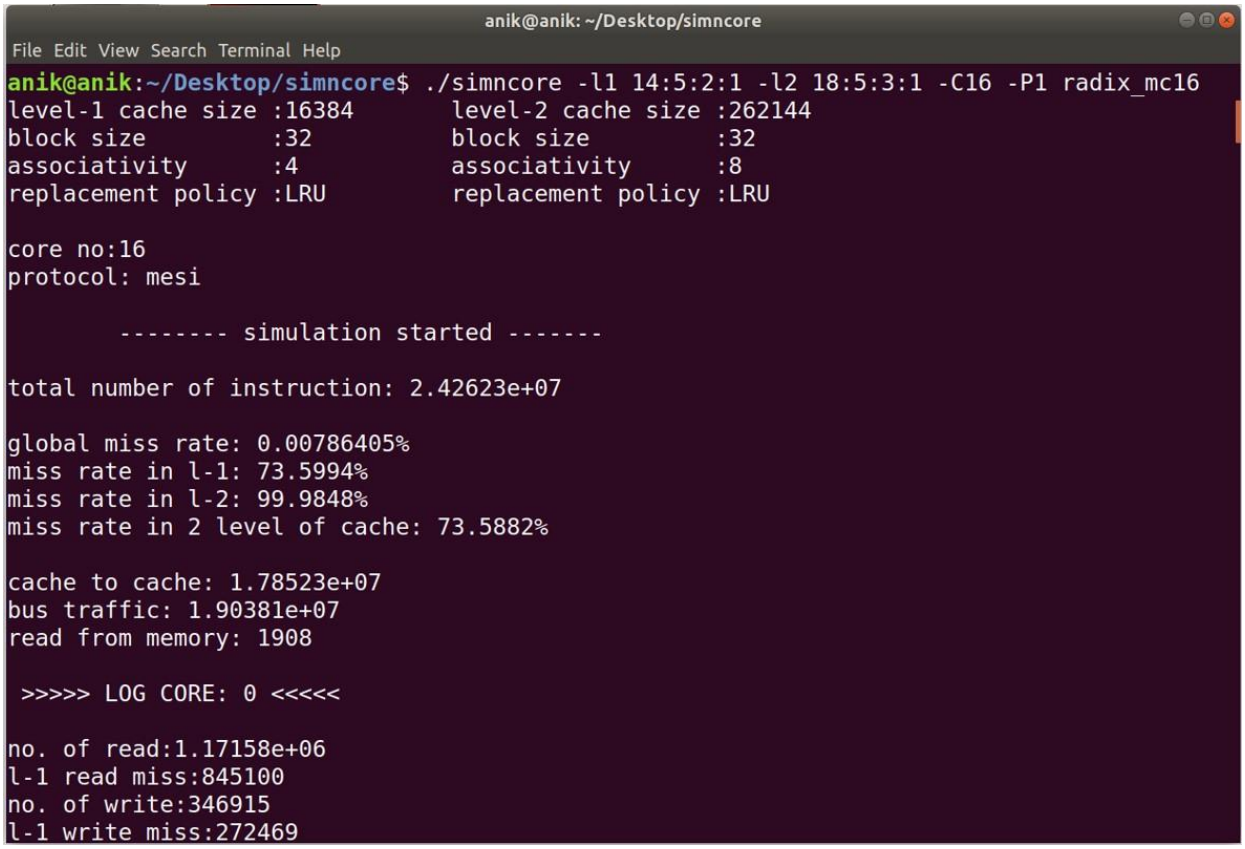

Figure A7: Manycore cache memory simulation (program: Radix, 16-core processor, protocol: MESI)

Section 3: Trace File Preparation with Pintool for Single-Core and Manycore Systems

Step 1: Download Intel PIN:

The installation procedure is provided on the webpage. The download link is given below:

https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-binary-instrumentation-tool-downloads.html

Step 2: Collect Benchmark Program:

We have used benchmark programs from Parsec 3.0 and Splash2 benchmark suits to collect trace files. The download link with installation procedure is provided in the link below:

https://parsec.cs.princeton.edu/parsec3-doc.htm

Step 3: Pintool Configuration to Collect Trace Files:

Once benchmark programs are created, place them in the same folder as Pin 'Manual Example'. Pintool (3.16) is used to generate the trace file necessary for SIMNCORE. We have used pintool 'pinatrace.cpp' to collect memory references for the programs. To get proper formatting, we modified 'pinatrace.cpp'. Our desired format is as follows:

<Core ID> <Instruction type> <Memory Address>

The 'instruction type' and 'memory address' are provided by the pinatrace.cpp output. The 'core id' we use here is considered an approximate call by any core of a single-core or manycore system. For generating 'core id' for each memory call, we simply inserted a function for a random number, keeping in mind how many processors are to be used for the simulation (see Figure 8).

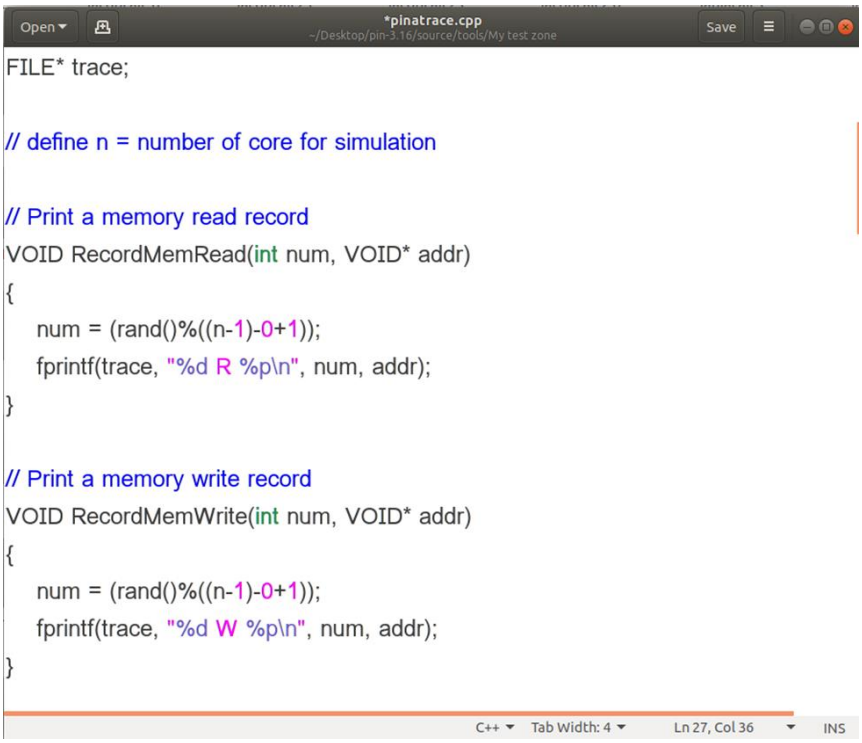

Figure A8: Modification of 'pinatrace.cpp'

In the 'rand ()' function, 'n' denotes the number of cores in the processor. (e.g., n = 1, 2, 3, 4…, etc.)

Next, we instrument benchmark or any other program to generate the trace file with our modified 'pinatrace.cpp' executables. The steps to get the executables/shared library file, and benchmark instrumentation are as follows:

1. *$ make pinatrace.test*
2. *$ pin -t obj-ia32/pinatrace.so -- [desired benchmark program]*

As benchmark programs often come with mandatory input parameters and environment setups, we need to provide those parameters during instrumentation (see Figures 9 and 10)

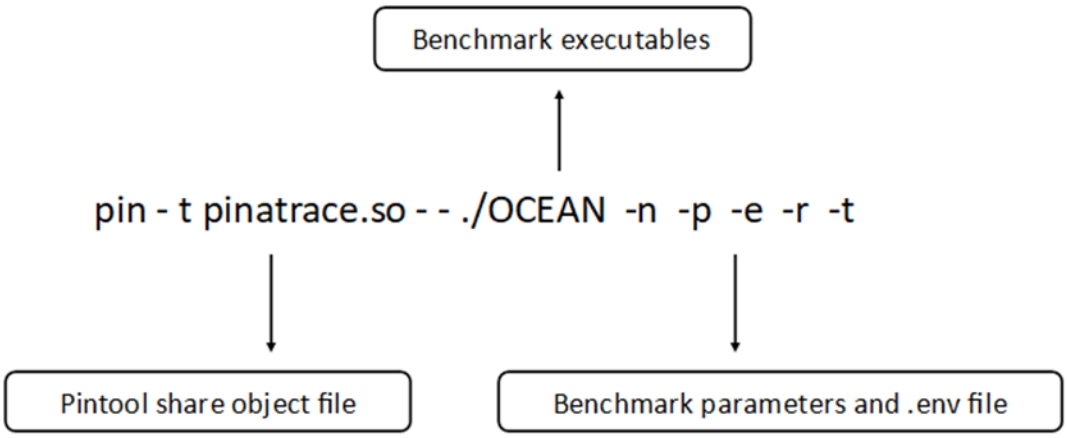

Figure 9: Pin instrumentation with associated parameters and environment

```
anik@ubuntu: ~/Desktop/pin-3.16/source/tools/testzone
anik@ubuntu:~/Desktop/pin-3.16/source/tools/testzone$ ../../../pin -t obj-ia32/
pinatrace.so -- ./ocean_cp -n514 -p1 -e1e-07 -r20000 -t28800

Ocean simulation with W-cycle multigrid solver
    Processors                          : 1
    Grid size                           : 514 x 514
    Grid resolution (meters)            : 20000.00
    Time between relaxations (seconds) : 28800
    Error tolerance                     : 1e-07


                    PROCESS STATISTICS
                Total          Multigrid         Multigrid
 Proc           Time             Time            Fraction
    0       -1254820572         679521779          -0.542

                    TIMING INFORMATION
Start time                        :       2125511766
Initialization finish time        :       3652488138
Overall finish time               :       2397675415
Total time with initialization    :        272163649
Total time without initialization :       3040154573
    (excludes first timestep)

anik@ubuntu:~/Desktop/pin-3.16/source/tools/testzone$
```

Figure A10: Command line instruction for Pin instrumentation (Benchmark: Ocean_cp)

The output of the program is stored in a ‘pinatrace.out’. The file holds all the traces of instructions and the memory references for the runtime operation. For instance (see Figure 6), our trace files for two benchmark programs have three columns each. The first column gives the approximate call from any core of a single-core or manycore system. The second column indicates the types of operations the processor is dealing with, such as ‘W’ for store operation, ‘R’ for read operation, and ‘Z’ for other instructions. In the third column, all the destination addresses for the associated

operation are collected. The left picture is a trace file of the FFT benchmark program for single-core simulation, and the right picture is a trace file of the Radix benchmark program for manycore (16-core) simulation. When we have an n-core processor in a system, the core id ranges from 0 to n-1. For example: in a 16-core processor, core id ranges from 0, 1, 2 …. to 15. For a single-core processor, the core id is 0.

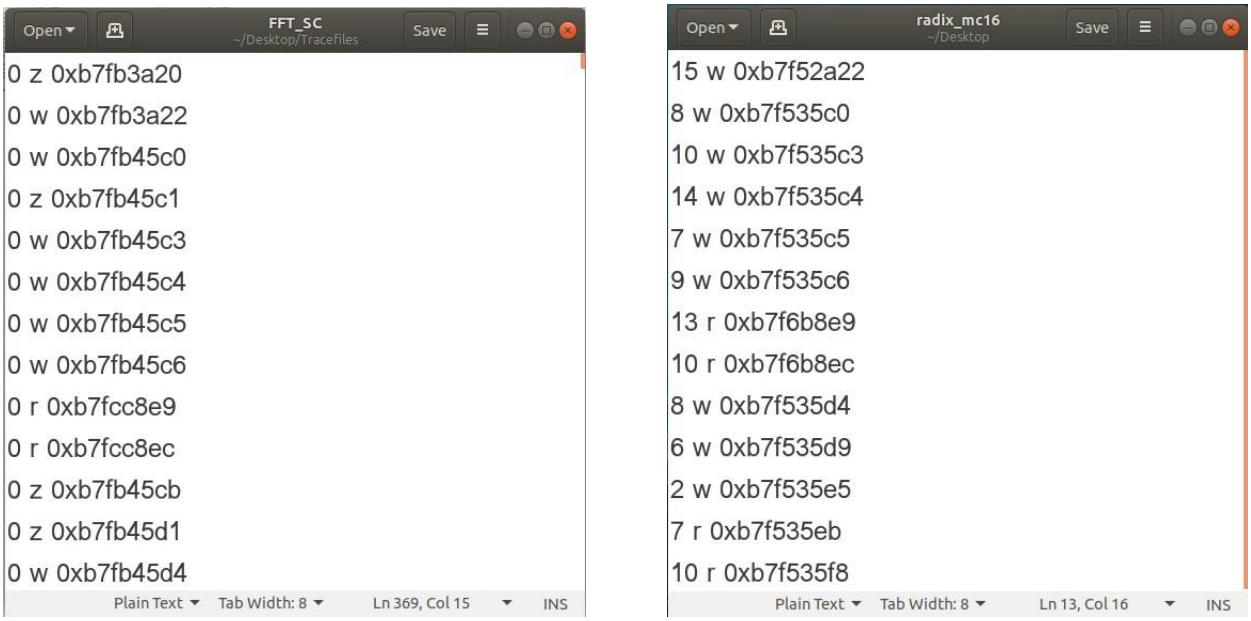

Figure A11: Trace file for FFT (single-core) and Radix (manycore) benchmark program

Section 4: Function Blocks in SIMNCORE

Function blocks in SIMNCORE is briefly discussed in this section. The following are some significant function blocks:

(a) **commandline (int argc, char **argv):** This function block is used to read the command line and segregate information (such as parameters, trace files, etc.).
(b) **parameter():** This function block is used to identify parameters related to L1 and L2 cache (such as, cache size, block size, etc.).
(c) **cache_design():** This function block is used to design caches based on their parameters.
(d) **primary_value():** This function block is used to set initial values for cache blocks (such as Dirty bit, LRU value, etc.).
(e) **cpp**: This code block is used to get information regarding the number of cores, coherence protocols, writing policies, etc. This code block also fetches memory addresses from the trace file.
(f) **adres_tag_index_block()**: This function block is used to deduce the tag bit, index bit, and block offset from memory addresses.
(g) **level1_hitmiss()**: This function block checks if the fetched address is present in the L1 cache.
(h) **level1()**: This function block performs two tasks.
    1) If a cache hit occurs in the L1 cache, this function block updates the L1 and L2 cache accordingly. Then the process moves on to update the replacement policy function. For manycore simulation, it also sends state updates and invalidation signals to the bus (when applicable).
    2) If a cache miss occurs in the L1 cache, the fetched address is searched in the L2 cache.
(i) **level2():** This function block performs two tasks similar to the previous one.
    1) If a cache hit occurs in L2 cache, this function block updates L1 and L2 cache accordingly. Then the process moves on to update the replacement policy function. For manycore simulation, it also sends state updates and invalidation signals to the bus (when applicable).
    2) For single-core simulation, if a cache miss occurs in L2 cache, requested data is collected from main memory. If there is a cache miss in the L2 cache during many-core cache simulation, the requested data is searched in the bus (the caches of other cores).If data is found on the bus, then caches are updated along with their states. Otherwise, data is read from main memory.

(j) **update_lru()**: This function block is used for replacement policies in different levels of caches. The LRU replacement policy is what SIMNCORE currently has, although additional replacement policies can be introduced.

(k) **log ()**: This function block is used to record cache events like cache misses, cache hits, bus traffic, etc.

In Figure 12, we have shown a flowchart that represents how the function blocks carry out the simulation task for single-core and manycore cache memory.

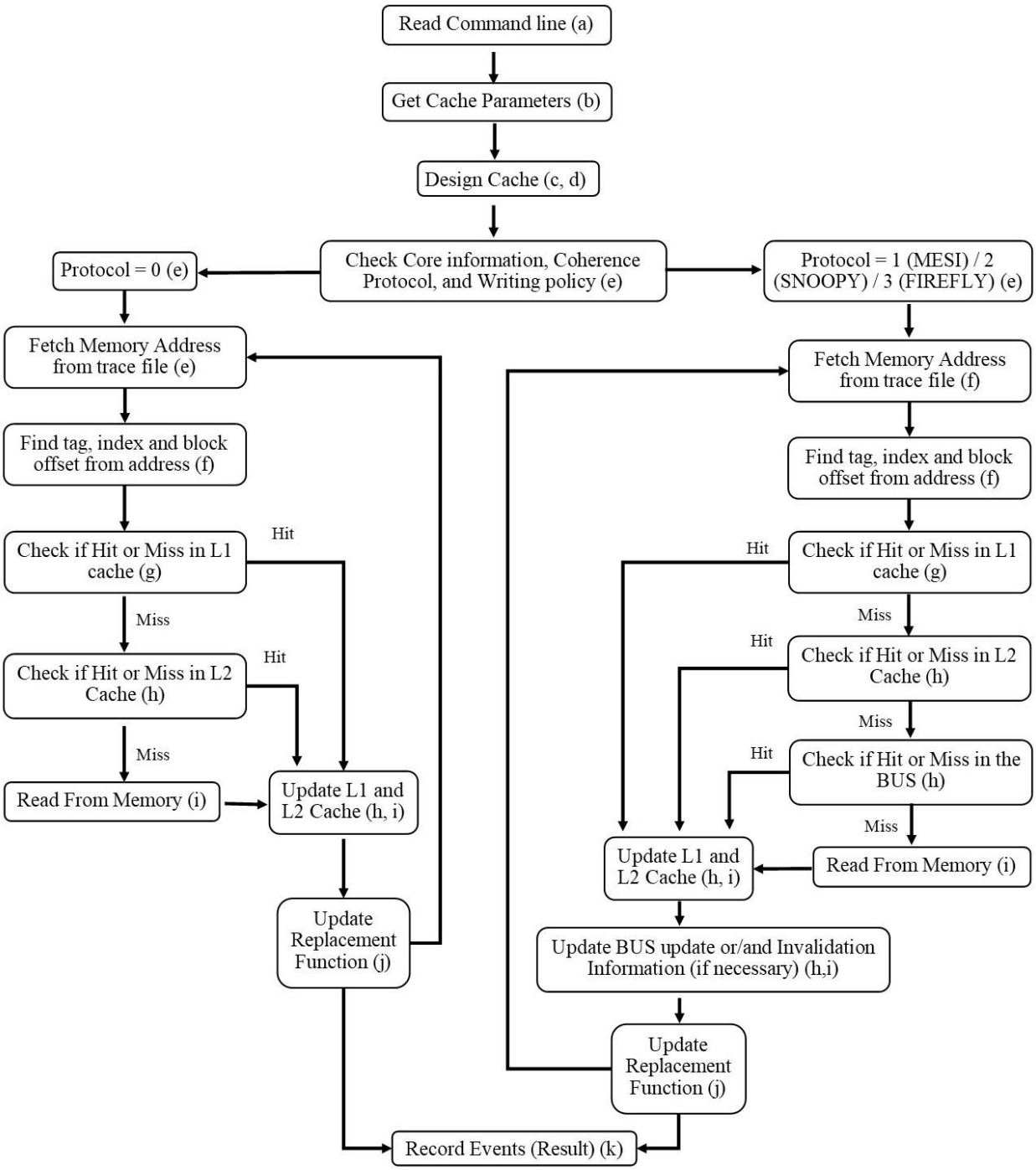

Figure A12: Function Blocks in SIMNCORE

Section 5: How to Implement a New Replacement Policy or New Cache Design Scheme in SIMNCORE

In this section, we have shown the procedures regarding porting new replacement policy or/and new coherence protocol in SIMNCORE.

For example, here, we have explained how to add '2-way skewed associative cache' in SIMNCORE.

The procedure for introducing new cache design in SIMNCORE can be categorized as follows: 1) selecting replacement policy for proposed cache design, 2) implementing mapping function for selected cache design; and 3) initializing the function through command line in SIMNCORE.

1) Replacement policy for 2-way skewed-associative Cache:

The conventional LRU policy for 2-way set associative cache memory has a control bit that is either 0 or 1. If any cache block's LRU value is '1', it means it has been updated recently, and if the LRU value is '0', it means it is the least recently used.

2-way skewed associative cache memory doesn't use conventional LRU replacement policy. Rather, it uses Pseudo-LRU replacement policy [1]. Similar to 2-way set associative cache memory, it has two banks: Bank 0 and Bank 1. With 'Pseudo-LRU' replacement policy, replacement control bit is associated with each cache block in Bank 0.

In case of a cache miss in Bank 0, cache block reads the data from memory, and its control bit is set to '1'. When control bit of cache block is '1' and it encounters a cache mis, new data is updated in Bank 1, and control bit is changed to '0'.

---

[1] A. Seznec, A case for 2-way skewed-associativity cache, *the 20th International Symposium on Computer Architecture (ISCA)*, San Diego, USA, May 1993.

Here is a sample function for Pseudo-LRU replacement policy,

```
// pseudo lru function
void pseudo_lru (int core)
{
   //when cache miss in bank 0, check lru value for bank 0
   if(cache1[core][bank0][index].lru==0)
   {
      f(read_memory);                    //read from memory to bank 0
      cache1[core][bank0][index].lru=1;  //bank 0 lru value is set to 1
   }

   if(cache1[core][bank0][index].lru==1)
   {
      f(read_memory);                    //read from memory to bank 1
      cache1[core][bank0][index].lru=0;  // bank 0 lru value is set to 0
   }
}
// end of function
```

2) XOR mapping function in 2-way skewed-associative cache:

Due to the same index value, multiple data may often face conflicts over the same cache blocks (due to temporal and spatial locality). In 2-way skewed cache memory, a different mapping function is used. To illustrate the situation closely, let's consider Bank 0 uses the conventional indexed mapping method (Figure 13). For bank 1, an XOR mapping function is used for indexing.

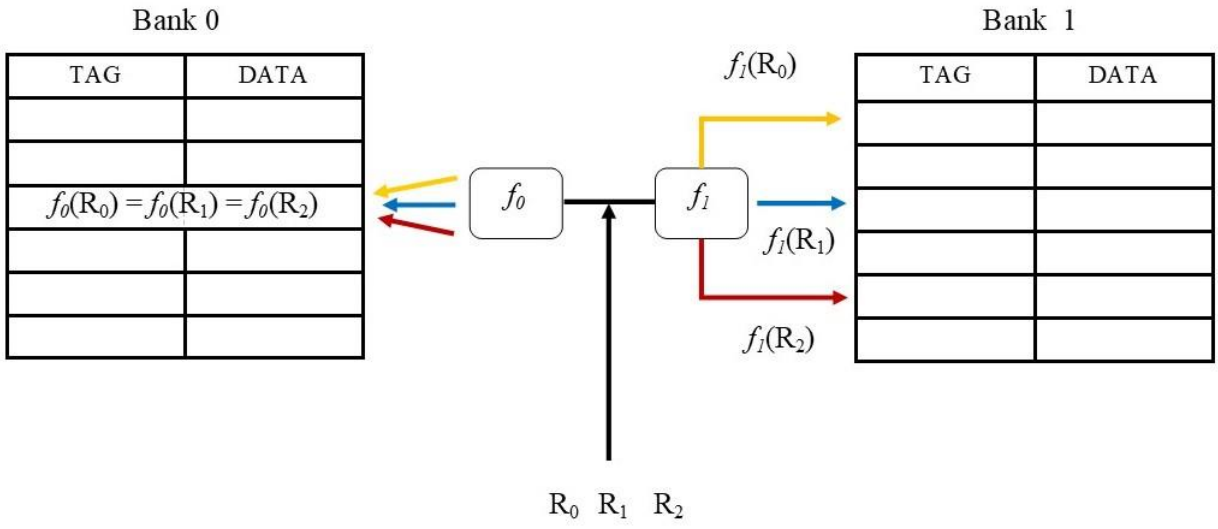

Figure A13: Conventional and XOR mapping function for Bank 0 and Bank 1 respectively

Figure 13 shows, in Bank 0, we can see three data (R0, R1, and R2) with same index value, access same location. Thus, the conflict miss occurs. On the other hand, three data even with the same index value, are dispersed throughout Bank 1, and thus conflict misses are reduced.

The process of XOR indexing in Bank 1 is discussed here:

Let's consider a cache memory with a 2^b block size and 2^n number of cache lines.

If the memory address is 32 bit long (Figure 14), then the tag will be 32 - n – b bit long.

We assume the index bit (set) is 6 bit long, denoted as A1 in Figure 14. In that case, we are splitting tag bits into two parts; we will consider the least significant 6 bit as A2, and rest of the bits as A3. A0 denotes the block offset.

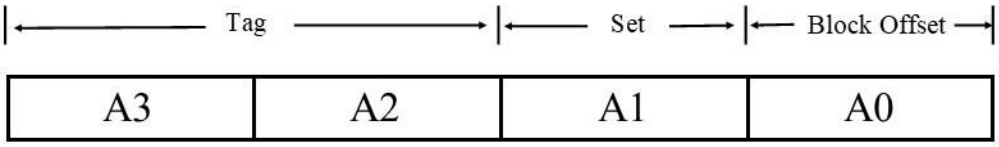

Figure A14: Tag, Index and Block offset of a memory address

For the XOR mapping function, we can consider a sample function as:

$f_1 (R0) = A2 \oplus A1$

This function performs a bitwise XOR operation with A1 and A2, finds the new indexed cache line in Bank 1, and then puts the data in that cache line.

There are also other versions of the XOR mapping function, like shuffling, circular shifting, etc. It is possible to have different mapping functions for each bank. But it is imperative to have the same bit length for A2 and A1 for XOR indexing. Figure 15 summarizes the XOR mapping function for 2-way set associative cache memory.

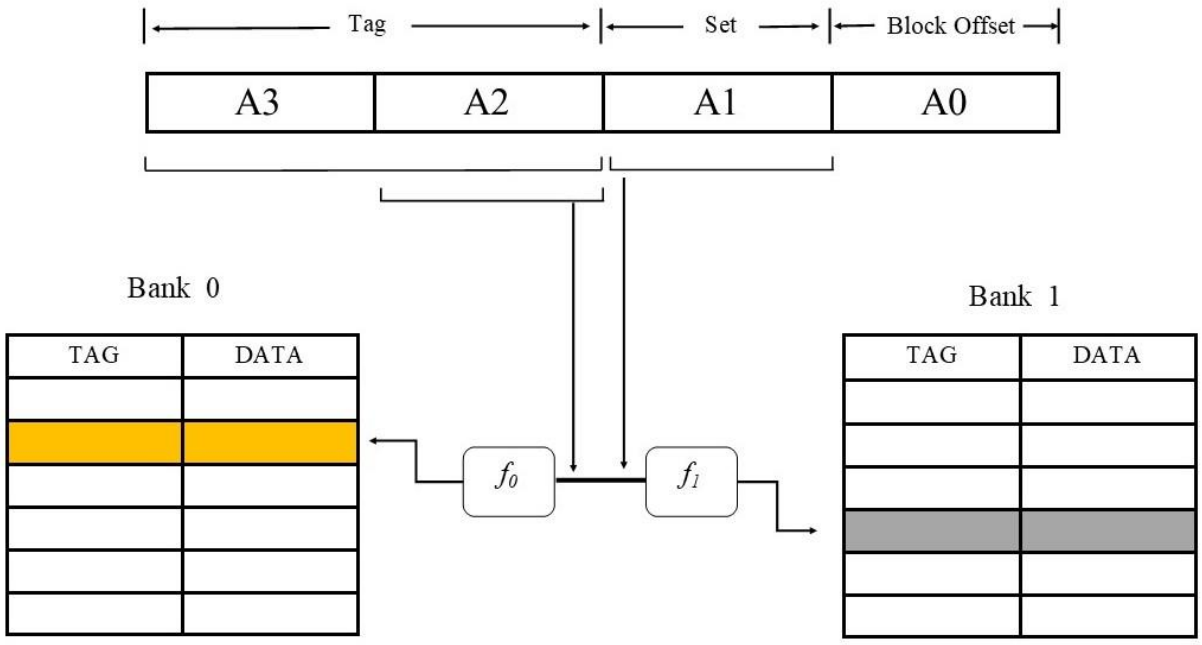

Figure A15: A 2-way Skewed Associative Cache

Here, some sample function for 2-way skewed-associated cache is shown:

```
// xor mapping function
void l1_skewed_cache()
{
   A2 = tag & ((1 << index_bit) - 1);
   A1 = index;
   new_index= A2^A1;                                    //bitwise xor
   cache1[core][bank1][new_index].address_tag=tag;      // put data in new indexed location
}
// end of function
```

3) Initializing the function through command line in SIMNCORE:

To implement a new cache scheme, such as a 2-way skewed associative cache, the replacement policy and mapping algorithm must be specified in SIMNCORE. In Simncore, the command line for a newly designed 2-way skewed-associative L1 may appear like this:

$ ./simncore -l1 12:5:S:2 -C1 -P0 -W1 <trace file>

Here, colored 'S' and '2' denote new associativity (XOR mapping function) and a new replacement policy (Pseudo-LRU) respectively, for the newly designed '2-way skewed associative cache'.

Comparing the flowchart from Figure 12, here, we have established how a 2-way skewed-associative cache can be designed by introducing Pseudo-LRU replacement policy and XOR mapping function with the function blocks (Figure 16).

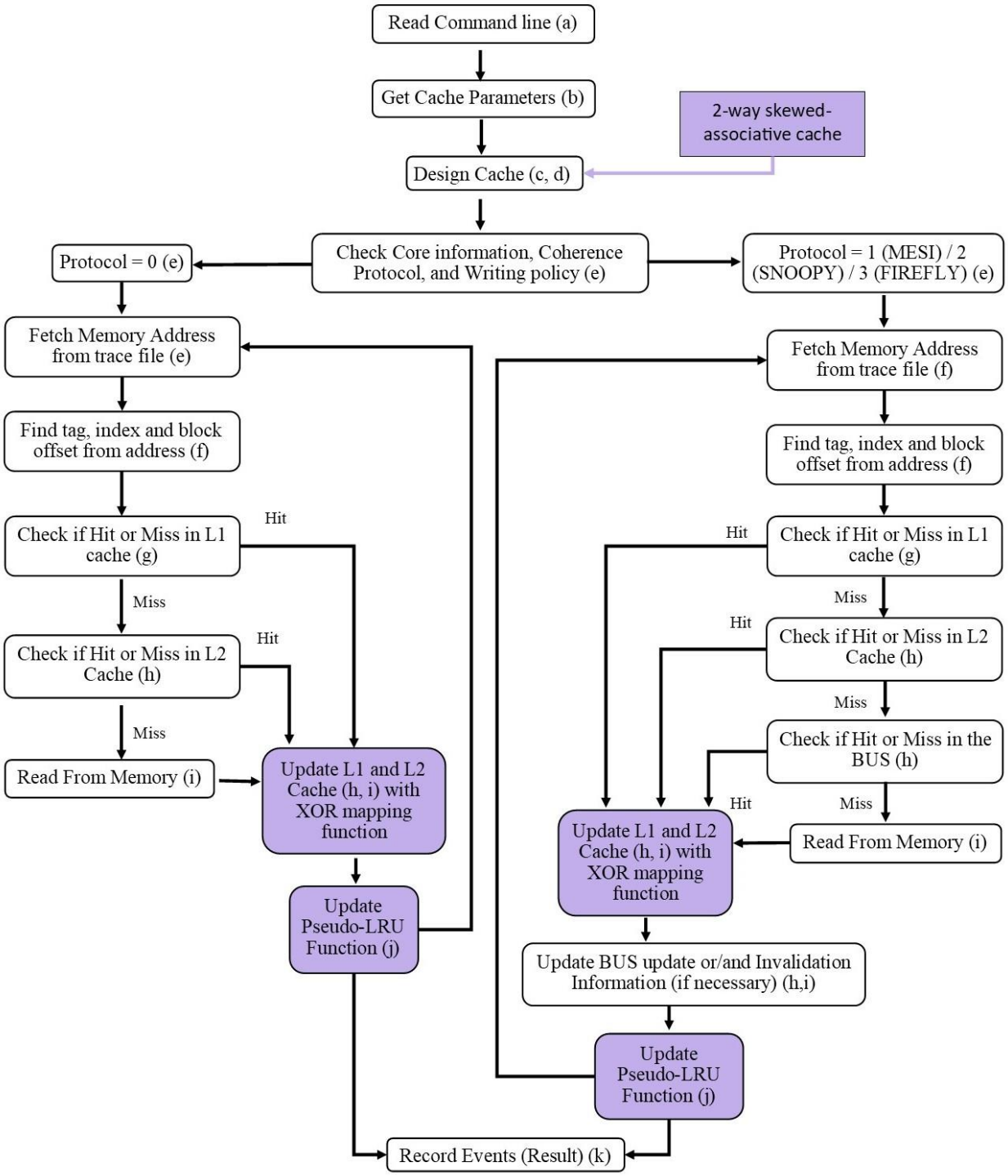

Figure A16: A schematic for designing 2-way skewed-associative cache memory in SIMNCORE

APPENDIX B

APPENDIX B

SOURCE CODE

Header.h

```
#include <stdio.h>
#include <stdlib.h>
#include <iomanip>
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <bitset>
#include <bits/stdc++.h>
#include "variable.h"
#include "build.h"
#include "simncorefunc.h"
```

Variable.h

```
int coreno1, coreno2, coreno, protocol, wp;
int para1, para2;
int core;

int ind1, blk1, tbit1, ca1, asc1, rp1;
int ind2, blk2, tbit2, ca2, asc2, rp2;

std::string line, cmdl1, cmdl2, adrs;
unsigned int dec_adrs;
unsigned int  block1, index1, tag1, block2, index2, tag2;
char type;

int a, b, m, n, out1, out2;

int ht1, tt1, ht2, tt2, th2;
int matchl2;
```

Build.h

```
// t=tag, v=validity, db= dirty bit, stat = state of the data
 struct level1
 {
    int t1, v1, lru1, db1;
    int data1;
    char stat1;
 }; struct level1** cash1[4096];

 struct level2
 {
    int t2, v2, lru2, db2;
    char stat2;
 }; struct level2** cash2[4096];

 struct lookup1
 {
    int mcore1, mthread1;
    char mstat1;
 }; struct lookup1 matchcore1[4096];

 struct lookup2
 {
    int mcore2, mthread2;
    char mstat2;
 }; struct lookup2 matchcore2[4096];

 double bustraffic=0, totalctoc=0, totalins=0, totalmissl1=0, missratel1=0;
 double missratel2=0, hitratel2=0, totalrml1=0, totalwml1=0, totaliml1=0;
 double totalread1=0, totalcalll2=0, totalmissl2=0;
 double totalrhpl1=0, totalwhpl1=0, totalrhbl1=0, totalwhbl1=0;
 double totalrml2=0, totalwml2=0, totaliml2=0;

 struct archive0
 {
    int rmem=0, wmem=0;
    int totalread = 0, totalwrite=0, totalins=0;
 }  cash0log;

 struct archive1
 {
    double rhit1, whit1, rmiss1, wmiss1, ihit1, imiss1;
 } cash1log;

 struct archive2
```

```
    {
      double rhit2, whit2, rmiss2, wmiss2, ihit2, imiss2;
    } cash2log;

    struct archive3
    {
      double rhl1, whl1, rml1, wml1, ihl1, iml1;
      double rhl2, whl2, rml2, wml2, ihl2, iml2;
      double  mrate, emrate;
      int rhp1=0, whp1=0, ihp1=0, rhp2=0, whp2=0,ihp2=0;
      int rhb=0, whb=0,rmb=0, wmb=0, ihb=0, imb=0;
      int mih1=0, eih1=0, sih1=0, mih2=0, eih2=0, sih2=0;
      int mrh1=0, erh1=0, srh1=0, mrh2=0, erh2=0, srh2=0;
      int mwh1=0, ewh1=0, swh1=0, mwh2=0, ewh2=0, swh2=0;
      int vrh1=0, drh1=0, vwh1=0, dwh1=0, vih1=0, dih1=0;
      int vrh2=0, drh2=0, vwh2=0, dwh2=0, vih2=0, dih2=0;
      int bu=0, pw=0, ctoc=0, exchange=0;
    }; struct archive3 corelog[4096];
```

Simncore.cpp

```
#include "header.h"

int main(int argc, char ** argv)
{
   commandline(argc, argv);
   parameter();
   cache_design();
   primary_value();

   if (ca2!=0)
   {
      cout <<"\nlevel-1 cache size :"<<(1<<ca1)<<'\t';
      cout <<"level-2 cache size :"<<(1<<ca2)<<endl;
      cout << "block size\t  :"<<(1<<blk1)<<'\t'<<'\t';
      cout << "block size\t  :"<<(1<<blk2)<<endl;
      cout << "associativity\t  :"<<(1<<asc1)<<'\t'<<'\t';
      cout << "associativity\t  :"<<(1<<asc2)<<endl;

      if (rp1==1){cout << "replacement policy :"<<"LRU"<<'\t'<<'\t';}
      if (rp2==1){cout << "replacement policy :"<<"LRU"<<endl;}

   }

   if (ca2==0)
```

```
{
   cout <<"\nlevel-1 cache size :"<<(1<<ca1);
   cout << "\nblock size\t  :"<<(1<<blk1);
   cout << "\nassociativity\t  :"<<(1<<asc1);

   if (rp1==1){cout << "\nreplacement policy :"<<"LRU"<<endl;}

}

cout << "\ncore no:"<< coreno<< endl;

if(protocol==0){cout << "protocol: single-core"<< endl;}
if(protocol==1){cout << "protocol: mesi"<< endl;}
if(protocol==2){cout << "protocol: snoopy"<< endl;}
if(protocol==3){cout << "protocol: firefly"<< endl;}

if (wp==1){cout<<"write policy: write-back"<<endl;}
if (wp==2){cout<<"write policy: write-through"<<endl;}

ifstream filename (argv[argc-1]);

if(protocol==0)//Singal core
{
   cout<<"\n\t-------- simulation started -------\n";

   while (filename >> core >> type >> adrs)
   {
      if (ca2==0)
      {
         adres_tag_index_block();
         level1_hitmiss_sc();
         only_l1_sc();
      }

      if(ca2!=0)
      {
         out2 = 2;
         adres_tag_index_block();
         level1_hitmiss_sc();
         level1_sc();
         level2_sc();
      }
   }
   if(ca2==0){log_sc_only_l1();}
   if(ca2!=0){log_sc();}
}
```

```
else if (protocol==1) //protocol: mesi
{
   cout<<"\n\t-------- simulation started -------\n";

   while (filename >> core >> type >> adrs)
   {
      if (ca2==0)
      {
         adres_tag_index_block();
         level1_hitmiss_mc();
         only_l1_mesi();
      }

      if(ca2!=0)
      {
         out2 = 2;
         adres_tag_index_block();
         level1_hitmiss_mc();
         level1_mesi();
         level2_mesi();
      }
   }
   if(ca2==0){log_mesi_only_l1();}
   if(ca2!=0){log_mesi();}
}

else if (protocol==2) //protocol: snoopy
{
   cout<<"\n\t-------- simulation started -------\n";

   while (filename >> core >> type >> adrs)
   {
      if (ca2==0)
      {
         adres_tag_index_block();
         level1_hitmiss_mc();
         only_l1_snoopy();
      }

      if(ca2!=0)
      {
         out2 = 2;
         adres_tag_index_block();
         level1_hitmiss_mc();
         level1_snoopy();
```

```
            level2_snoopy();
          }
        }

        if(ca2==0){log_mesi_only_l1();}
        if(ca2!=0){log_mesi();}
      }

      else if (protocol==3) //protocol: firefly
      {
        cout<<"\n\t-------- simulation started -------\n";

        while (filename >> core >> type >> adrs)
        {
          if (ca2==0)
          {
            adres_tag_index_block();
            level1_hitmiss_mc();
            only_l1_firefly();
          }

          if(ca2!=0)
          {
            out2 = 2;
            adres_tag_index_block();
            level1_hitmiss_mc();
            level1_firefly();
            level2_firefly();
          }
        }
        if(ca2==0){log_firefly_only_l1();}
        if(ca2!=0){log_firefly();}
      }
}
```

Simncorefunc.h

```
using namespace std;

void commandline(int argc, char **argv);
void parameter();
void cache_design();
void primary_value();
void adres_tag_index_block();
```

```
void level1_hitmiss_sc();
void only_l1_sc();
void level1_sc();
void level2_sc();

void level1_hitmiss_mc();
void only_l1_mesi();
void level1_mesi();
void level2_mesi();
void only_l1_snoopy();
void level1_snoopy();
void level2_snoopy();
void only_l1_firefly();
void level1_firefly();
void level2_firefly();

int find_l2(int coreid2);

int highest_lru1();
int highest_lru2();
int searchcores1(int coreid1);
int searchcores2(int coreid2);
void update_lru1(int coreid1, int threadid1);
void check_lru1(int coreid1, int threadid1);
void check_lru2(int corid2, int threadid2);
void update_lru2(int coreid2, int threadid2);

void log_sc_only_l1();
void log_sc();
void log_mesi_only_l1();
void log_mesi();
void log_firefly_only_l1();
void log_firefly();

void update_lru1(int coreid1, int threadid1)
{
 int up1, n = 1 << asc1;
 up1 = cash1[coreid1][threadid1][index1].lru1;

 for (int i = 0; i < n; i++)
 {
  if (cash1[coreid1][i][index1].lru1 < up1)
  {cash1[coreid1][i][index1].lru1++;}
 }
 cash1[coreid1][threadid1][index1].lru1 = 0;
```

```
}

void check_lru1(int coreid1, int threadid1)
{
 int check1, n= 1 << asc1;
 check1 = cash1[coreid1][threadid1][index1].lru1;

 for (int i = 0; i < n; i++)
 {
  if (cash1[coreid1][i][index1].lru1 > check1)
  {cash1[coreid1][i][index1].lru1--;}
 }
 cash1[coreid1][threadid1][index1].lru1 = (n-1);
}

void check_lru2(int coreid2,int threadid2)
{
 int check2, m = 1 << asc2;
 check2 = cash2[coreid2][threadid2][index2].lru2;

 for (int i = 0; i < m; i++)
 {
  if (cash2[coreid2][i][index2].lru2 > check2)
  {cash2[coreid2][i][index2].lru2--;}
 }
 cash2[coreid2][threadid2][index2].lru2 = (m-1);
}

void update_lru2(int coreid2, int threadid2)
{
 int up2, m = 1 << asc2;
 up2 = cash2[coreid2][threadid2][index2].lru2;

 for (int i = 0; i < m; i++)
 {
  if (cash2[coreid2][i][index2].lru2 < up2)
  {cash2[coreid2][i][index2].lru2++;}
 }

 cash2[coreid2][threadid2][index2].lru2 = 0;
}

int highest_lru2()
{
 int cc, hlthread2, m = 1 << asc2;
 cc = cash2[core][0][index2].lru2;
```

```
 for (int i = 0; i < m; i++)
 {
  if (cc <= cash2[core][i][index2].lru2)
  {
   hlthread2 = i;
   cc = cash2[core][i][index2].lru2;
  }
 }
 return hlthread2;
}

int highest_lru1()
{
 int aa, hlthread1, n = 1 << asc1;
 aa = cash1[core][0][index1].lru1;

 for (int i = 0; i < n; i++)
 {
  if (aa <= cash1[core][i][index1].lru1)
  {
   hlthread1 = i;
   aa = cash1[core][i][index1].lru1;
  }
 }
 return hlthread1;
}

int searchcores1(int coreid1)
{
 int match1=0, n=1<<asc1;

 for (int j=0; j<coreno;j++)
 {
  if (j!=coreid1)
  {
   for (int i=0; i<n;i++)
   {
    if (cash1[j][i][index1].v1==1)
    {
     if (cash1[j][i][index1].t1==tag1)
     {
      matchcore1[match1].mcore1=j;
      matchcore1[match1].mthread1=i;
      matchcore1[match1].mstat1=cash1[j][i][index1].stat1;
      match1++;
```

```
        }
       }
      }
     }
    }

  return match1;
}

int searchcores2(int coreid2)
{
  int match2=0, m=1<<asc2;

  for (int y=0; y<coreno;y++)
  {
    if (y!=coreid2)
    {
      for (int x=0; x<m;x++)
      {
        if (cash2[y][x][index2].v2==1)
        {
          if (cash2[y][x][index2].t2==tag2)
          {
            matchcore2[match2].mcore2=y;
            matchcore2[match2].mthread2=x;
            matchcore2[match2].mstat2=cash2[y][x][index2].stat2;
            match2++;
          }
        }
      }
    }
  }
  return match2;
}

int find_l2(int coreid2)
{
  int th2 = 0, m = 1 << asc2;

  for (int j = 0; j < m; j++)
  {
    if (cash2[coreid2][j][index2].v2 == 1)
    {
      if (cash2[coreid2][j][index2].t2 == tag2)
      {
        th2 = j;
```

```
      break;
    }
   }
  }
  return th2;
}

void commandline(int argc, char **argv)
{
  for (int i = 1; i < argc; i++)
  {
   line = argv[i];
   if (line[0] == 45) // 45 is ASCII value of '-'
   {
    if (line[1] == 108) // 108 is ASCII value of 'l'
    {
     while (i == 1)
     {
      if (line[2] == 49) // 49 is ASCII value of '1'
      {
       i++;
       cmdl1 = argv[i];
      }

      else
      {
       std::cout << "error l-1 command" << std::endl;
       break;
      }
     }

     while (i == 3)
     {
      if (line[2] == 50) //// 50 is ASCII value of '2'
      {
       i++;
       cmdl2 = argv[i];
      }

      else
      {
       std::cout << "error l-2 command" << std::endl;
       break;
      }
     }
    }
```

```
    else if (line[1] == 67) // 67 is ASCII value of 'C'
    {
      stringstream ss;
      ss << line[2];
      ss >> coreno1;
      coreno=coreno1;

      stringstream rr;
      rr<<line[3];
      rr>>coreno2;

      if(coreno2!=0)
      {coreno=coreno*10+coreno2;}
    }

    else if (line[1] == 80) // 80 is ASCII value of 'P'
    {
      stringstream pp;
      pp << line[2];
      pp >> protocol;
    }

    // for write policy
    else if(line[1]==87) // 87 is ascii value of "W"
    {
      stringstream qq;
      qq << line[2];
      qq >> wp;
    }
   }
  }
}

void parameter()
{
  // parameter for level-1
  int j = 0; int para1[5] = {0, 0, 0, 0, 0};

  for (int i = 0; i <= 4; i++)
  {
    if (cmdl1[0] != 58)
    {
      char m = cmdl1[j];
      para1[i] = (int)m - 48;
      j = j + 1;
```

```
   if (j < cmdl1.length())
   {
    m = cmdl1[j];
    while (m != 58)
    {
     para1[i] = para1[i] * 10;
     para1[i] = para1[i] + ((int)m - 48);
     j = j + 1;
     m = cmdl1[j];
    }
    j = j + 1;
   }
   else
   {
    break;
   }
  }
  else
  {
   cout << "command line error" << endl;
  }
}

ca1 = para1[0];
blk1 = para1[1];
asc1 = para1[2];
rp1 = para1[3];
ind1 = ca1 - blk1 - asc1;
tbit1 = 32 - blk1 - ind1;

// parameter for level-2
int k = 0; int para2[5] = {0, 0, 0, 0, 0};

for (int i = 0; i <= 4; i++)
{
 if (cmdl2[0] != 58)
 {
  char m = cmdl2[k];
  para2[i] = (int)m - 48;
  k = k + 1;

  if (k < cmdl2.length())
  {
   m = cmdl2[k];
   while (m != 58)
```

```
      {
       para2[i] = para2[i] * 10;
       para2[i] = para2[i] + ((int)m - 48);
       k = k + 1;
       m = cmdl2[k];
      }
      k = k + 1;
    }
    else
    {
     break;
    }
   }
   else
   {
    cout << "command line error" << endl;
   }
 }

 ca2 = para2[0];
 blk2 = para2[1];
 asc2 = para2[2];
 rp2 = para2[3];
 ind2 = ca2 - blk2 - asc2;
 tbit2 = 32 - blk2 - ind2;
}

void cache_design()
{
 // level-1 cache
 for (int i = 0; i < coreno; i++)
 {
  cash1[i] = new struct level1 *[1 << asc1];

  for (int n = 0; n < (1 << asc1); n++)
  {
   cash1[i][n] = new struct level1[1 << ind1];
  }
 }

 // level-2 cache
 for (int x = 0; x < (1 << coreno); x++)
 {
  cash2[x] = new struct level2 *[1 << asc2];

  for (int m = 0; m < (1 << asc2); m++)
```

```
   {
    cash2[x][m] = new struct level2[1 << ind2];
   }
  }
}

void primary_value()
{
 // for level-1
 for (int i = 0; i < coreno; i++)
 {
  for (int j = 0; j < (1 << asc1); j++)
  {
   for (int k = 0; k < (1 << ind1); k++)
   {
    cash1[i][j][k].t1 = -1;
    cash1[i][j][k].v1 = 0;
    cash1[i][j][k].lru1 = j;
    cash1[i][j][k].db1 = 1;
    cash1[i][j][k].stat1 = 'i';
    cash1[i][j][k].data1 = -1;
   }
  }
 }

 // for level-2
 for (int x = 0; x < (1 << coreno); x++)
 {
  for (int y = 0; y < (1 << asc2); y++)
  {
   for (int z = 0; z < (1 << ind2); z++)
   {
    cash2[x][y][z].t2 = -1;
    cash2[x][y][z].v2 = 0;
    cash2[x][y][z].lru2 = y;
    cash2[x][y][z].db2 = 1;
    cash2[x][y][z].stat2 = 'i';
   }
  }
 }

 // assign value
 for (int i = 0; i < coreno; i++)
 {
  corelog[i].rhl1 = 0;
  corelog[i].whl1 = 0;
```

```
   corelog[i].rml1 = 0;
   corelog[i].wml1 = 0;
   corelog[i].ihl1 = 0;
   corelog[i].iml1 = 0;

   corelog[i].rhl2 = 0;
   corelog[i].whl2 = 0;
   corelog[i].rml2 = 0;
   corelog[i].wml2 = 0;
   corelog[i].ihl2 = 0;
   corelog[i].iml2 = 0;
  }
}

void adres_tag_index_block()
{
  // get address in decimal
  dec_adrs = std::stoul(adrs, nullptr, 16);

  // get tag, index, block address in decimal for level-1
  block1 = dec_adrs & ((1 << blk1) - 1);
  index1 = (dec_adrs >> blk1) & ((1 << ind1) - 1);
  tag1 = (dec_adrs >> (ind1 + blk1)) & ((1 << tbit1) - 1);

  // get tag, index, block address in decimal for level-2
  block2 = dec_adrs & ((1 << blk2) - 1);
  index2 = (dec_adrs >> blk2) & ((1 << ind2) - 1);
  tag2 = (dec_adrs >> (ind2 + blk2)) & ((1 << tbit2) - 1);
}



void level1_hitmiss_sc()
{
  int a = 2, n = 1 << asc1;

  for (int i = 0; i < n; i++)
  {
    if (cash1[core][i][index1].v1 == 1)
    {
      if (cash1[core][i][index1].t1 == tag1) // hit in level-1
      {
        ht1 = i;
        a = 1;
        out1 = 1;
```

```
      if (type == 'z'){corelog[core].ihl1++;}
      if (type == 'r'){corelog[core].rhl1++;}
      if (type == 'w'){corelog[core].whl1++;}

      break;
    }
   }
  }

  if (a == 2) //miss in level-1
  {
   out1 = 0;

   if (type == 'z'){corelog[core].iml1++;}
   if (type == 'r'){corelog[core].rml1++;}
   if (type == 'w'){corelog[core].wml1++;}
  }
}

void only_l1_sc()
{
  if (out1 == 1) // when hit in level-1
  {
   if (type == 'z'&& wp==1){update_lru1(core, ht1);}
   if (type == 'z'&& wp==2){update_lru1(core, ht1);}
   if (type == 'r' && wp==1){update_lru1(core, ht1);}
   if (type == 'r' && wp==2){update_lru1(core, ht1);}

   if (type == 'w' && wp==1)
   {
    cash1[core][ht1][index1].db1 = 1; // dirtybit =1
    update_lru1(core, ht1);
   }

   if (type == 'w' && wp==2)
   {
    cash0log.wmem++;
    update_lru1(core, ht1);
   }
  }

  if (out1 == 0) //miss in level-1 check in level-2
  {
   // find highest lru in level-1
   tt1 = highest_lru1();
```

```
if (type == 'z' && wp==1)
{
 cash0log.rmem++;

 if (cash1[core][tt1][index1].db1==0)
 {
  //copy data from memory to l1
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1 = 1;
  cash1[core][tt1][index1].db1=0;
 }

 if (cash1[core][tt1][index1].db1==1)
 {
  cash0log.wmem++;

  //copy data from memory to l1
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1 = 1;
  cash1[core][tt1][index1].db1=0;
 }
 update_lru1(core, tt1);
}

if (type == 'z' && wp==2) // write-through / no allocate
{
 cash0log.rmem++;

 // copy from memory to l1
 cash1[core][tt1][index1].t1=tag1;
 cash1[core][tt1][index1].v1 = 1;
 update_lru1(core, tt1);
}

if (type == 'r' && wp==1)
{
 cash0log.rmem++;

 if (cash1[core][tt1][index1].db1==0)
 {
  //copy data from memory to tt1
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1 = 1;
  cash1[core][tt1][index1].db1=0;
 }
```

```
 if (cash1[core][tt1][index1].db1==1)
 {
  cash0log.wmem++;

  //copy data from memory to tt1
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1 = 1;
  cash1[core][tt1][index1].db1=0;
 }
 update_lru1(core, tt1);
}

if (type == 'r' && wp==2) // write-through / no allocate
{
 cash0log.rmem++;

 // copy from memory to tt2
 cash1[core][tt1][index1].t1=tag1;
 cash1[core][tt1][index1].v1 = 1;
 update_lru1(core, tt1);
}

if (type == 'w' && wp==1)
{
 if (cash1[core][tt1][index1].db1==0)
 {
  cash0log.wmem++;

  // copy data from processor to l1
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1 = 1;
  cash1[core][tt1][index1].db1=1;
 }

 if (cash1[core][tt1][index1].db1==1)
 {
  cash0log.wmem++;

  // copy data from processor to l1
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1 = 1;
 }
 update_lru1(core, tt1);
}

if (type == 'w' && wp==2)
```

```
   {
    cash0log.wmem++;
   }
  }
}

void level1_sc()
{
 if (out1 == 1) // hit in level-1
 {
  //find copy in l2
  th2=find_l2(core);

  if (type=='z'&& wp==1){update_lru1(core, ht1);}
  if (type == 'z'&& wp==2){update_lru1(core, ht1);}
  if (type == 'r' && wp==1){update_lru1(core, ht1);}
  if (type == 'r' && wp==2){update_lru1(core, ht1);}

  if (type == 'w' && wp==1)
  {
   cash1[core][ht1][index1].db1 = 1;
   cash2[core][th2][index2].db2 = 1;
   update_lru1(core, ht1);
   update_lru2(core, th2);
  }

  if (type == 'w' && wp==2)
  {
   cash0log.wmem++;
   update_lru1(core, ht1);
   update_lru2(core, th2);
  }
 }

 if (out1 == 0) //miss in level-1 check in level-2
 {
  tt1 = highest_lru1(); //find highest lru in level-1
  tt2 = highest_lru2(); //find highest lru in level-2

  b = 0, m = 1 << asc2;
  for (int i = 0; i < m; i++)
  {
   if (cash2[core][i][index2].v2 == 1)
   {
    if (cash2[core][i][index2].t2 == tag2)
    {
```

```
        ht2 = i;
        b = 1;
        out2 = 1; // in case of hit in level-2

        break;
       }
     }

     if (b == 0) // in case of miss in level-2
     {
       out2 = 0;
     }
   }
  }
}

void level2_sc()
{
  if (out2==1) //hit in level-2
  {
    if (type == 'z'){corelog[core].ihl2++;}
    if (type == 'r'){corelog[core].rhl2++;}
    if (type == 'w'){corelog[core].whl2++;}

    if (type == 'z' && wp==1)
    {
      //copy from l2 hit thread to l1 target thread
      cash1[core][tt1][index1].t1=tag1;
      cash1[core][tt1][index1].v1=1;
      cash1[core][tt1][index1].db1=0;
      update_lru1(core, tt1);
      update_lru2(core, ht2);
    }

    if (type == 'z' && wp==2)
    {
      //copy from l2 hit thread to l1 target thread
      cash1[core][tt1][index1].t1=tag1;
      cash1[core][tt1][index1].v1=1;
      update_lru1(core, tt1);
      update_lru2(core, ht2);
    }

    if (type == 'r' && wp==1)
    {
//copy from l2 hit thread to l1 target thread
```

```
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1=1;
  cash1[core][tt1][index1].db1=0;
  update_lru1(core, tt1);
  update_lru2(core, ht2);
 }

 if (type == 'r' && wp==2) // write-through
 {
  //copy from l2 hit thread to l1 target thread
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1=1;
  update_lru1(core, tt1);
  update_lru2(core, ht2);
 }

 if (type == 'w' && wp==1)
 {
  // write to l1
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1=1;
  cash1[core][tt1][index1].db1=1;

  //update l2
  cash2[core][ht2][index2].t2=tag2;
  cash2[core][ht2][index2].db2=1;

  update_lru1(core, tt1);
  update_lru2(core, ht2);
 }

 if (type == 'w' && wp==2)
 {
  //write to l1
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1=1;

  //update l2
  cash2[core][ht2][index2].t2=tag2;

  cash0log.wmem++;
  update_lru1(core, tt1);
  update_lru2(core, ht2);
 }
}
```

```
if (out2==0) // miss in level-2
{
 if (type == 'z'){corelog[core].iml2++;}
 if (type == 'r'){corelog[core].rml2++;}
 if (type == 'w'){corelog[core].wml2++;}

 if (type == 'z' && wp==1)
 {
  if (cash1[core][tt1][index1].db1==1){cash0log.wmem++;}

  //copy data from mmemory to l1 & l2
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1 = 1;
  cash1[core][tt1][index1].db1=0;

  cash2[core][tt2][index2].t2=tag2;
  cash2[core][tt2][index2].v2=1;
  cash2[core][tt2][index2].db2=0;

  cash0log.rmem++;
  update_lru1(core, tt1);
  update_lru2(core, tt2);
 }

 if (type == 'z' && wp==2)
 {
  // copy from memory to l1 and l2
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1 = 1;

  cash2[core][tt2][index2].t2=tag2;
  cash2[core][tt2][index2].v2=1;

  cash0log.rmem++;
  update_lru1(core, tt1);
  update_lru2(core, tt2);
 }

 if (type == 'r' && wp==1)
 {
  if (cash1[core][tt1][index1].db1==1){cash0log.wmem++;}

  //copy data from memory to l1 and l2
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1 = 1;
  cash1[core][tt1][index1].db1=0;
```

```
  cash2[core][tt2][index2].t2=tag2;
  cash2[core][tt2][index2].v2=1;
  cash2[core][tt2][index2].db2=0;

  cash0log.rmem++;
  update_lru1(core, tt1);
  update_lru2(core, tt2);
 }

 if (type == 'r' && wp==2)
 {
  // copy from memory to l1 and l2
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1 = 1;

  cash2[core][tt2][index2].t2=tag2;
  cash2[core][tt2][index2].v2=1;

  cash0log.rmem++;
  update_lru1(core, tt1);
  update_lru2(core, tt2);
 }

 if (type == 'w' && wp==1)
 {
  if (cash1[core][tt1][index1].db1==1) {cash0log.wmem++;}

  // write to l1 and l2
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1 = 1;
  cash1[core][tt1][index1].db1=1;

  cash2[core][tt2][index2].t2=tag2;
  cash2[core][tt2][index2].v2=1;
  cash2[core][tt2][index2].db2=1;

  update_lru1(core, tt1);
  update_lru2(core, tt2);
 }

 if (type == 'w' && wp==2)
 {
  cash0log.wmem++;
 }
}
```

```
}

void level1_hitmiss_mc()
{
 int a=2, n=1<<asc1;

 for (int i=0;i<n;i++)
 {
  if (cash1[core][i][index1].v1==1)
  {
   if (cash1[core][i][index1].t1==tag1)
   {
    a=1;
    out1=1;
    ht1=i;

    if (type=='z'){corelog[core].ihl1++;}
    if (type=='r'){corelog[core].rhl1++;}
    if (type=='w'){corelog[core].whl1++;}

    break;
   }
  }
 }

 if (a==2)
 {
  out1=0;

  if (type=='z'){corelog[core].iml1++;}
  if (type=='r'){corelog[core].rml1++;}
  if (type=='w'){corelog[core].wml1++;}
 }
}

void only_l1_mesi()
{
 if (out1==1) //in case of hit in level-1 private core
 {

  if (type =='z')
  {
   corelog[core].ihp1++;

   if (cash1[core][ht1][index1].stat1=='m'){corelog[core].mih1++;}
   if (cash1[core][ht1][index1].stat1=='e'){corelog[core].eih1++;}
```

```
  if (cash1[core][ht1][index1].stat1=='s'){corelog[core].sih1++;}
 }

 if (type =='r')
 {
  corelog[core].rhp1++;

  if (cash1[core][ht1][index1].stat1=='m'){corelog[core].mrh1++;}
  if (cash1[core][ht1][index1].stat1=='e'){corelog[core].erh1++;}
  if (cash1[core][ht1][index1].stat1=='s'){corelog[core].srh1++;}
 }

 if (type == 'w')
 {
  corelog[core].whp1++;

  if (cash1[core][ht1][index1].stat1=='m'){corelog[core].mwh1++;}
  if (cash1[core][ht1][index1].stat1=='e'){corelog[core].ewh1++;}
  if (cash1[core][ht1][index1].stat1=='s'){corelog[core].swh1++;}

  // invalidate other copies
  if (cash1[core][ht1][index1].stat1=='s')
  {
   int totalmatch1 = 0; totalmatch1=searchcores1(core);

   for(int i=0; i<totalmatch1; i++)
   {
    cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='i';
    cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].v1=0;
    check_lru1(matchcore1[i].mcore1,matchcore1[i].mthread1);
   }

   if(totalmatch1!=0){corelog[core].bu++;} //update bus
  }

  //write to l1 cache
  cash1[core][ht1][index1].stat1='m';
 }

 update_lru1(core, ht1);
}

if (out1==0) //miss in l1
{
 int tt1 = 0; tt1 = highest_lru1(); //highest lru line to replace
```

```
// find hit or miss in bus
int totalmatch1 = 0; totalmatch1=searchcores1(core);

if(totalmatch1!=0) //hit in bus
{
 corelog[core].ctoc++;

 if (type=='z')
 {
  corelog[core].ihb++;

  //copy from hit thread to target l1
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1=1;
  cash1[core][tt1][index1].stat1='s';

  //change state to shared of copies in bus
  int totalmatch1=0; totalmatch1=searchcores1(core);
  for (int i=0; i<totalmatch1; i++)
  {
   if (matchcore1[i].mstat1!='s')
   {
    cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='s';
   }
  }
  update_lru1(core, tt1);
 }

 if (type=='r')
 {
  corelog[core].rhb++;

  //copy from hit thread to target l1
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1=1;
  cash1[core][tt1][index1].stat1='s';

  // change state to shared of copies in bus
  int totalmatch1=0; totalmatch1=searchcores1(core);
  for (int i=0; i<totalmatch1; i++)
  {
   if (matchcore1[i].mstat1!='s')
   {
    cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='s';
   }
  }
```

```
  update_lru1(core, tt1);
 }

 if (type=='w')
 {
  corelog[core].whb++;

  // write to l1
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1=1;
  cash1[core][tt1][index1].stat1='m';

  //invalidate copies in bus
  int totalmatch1=0; totalmatch1=searchcores1(core);
  for(int i=0; i<totalmatch1;i++)
  {
   cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='i';
   cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].v1=0;
   check_lru1(matchcore1[i].mcore1,matchcore1[i].mthread1);
  }
  update_lru1(core, tt1);
 }
}

if(totalmatch1==0) //miss in bus
{
 if (type == 'z')
 {
  corelog[core].imb++;

  //copy from memory to l1
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1=1;
  cash1[core][tt1][index1].stat1='s';

  update_lru1(core, tt1);
 }

 if (type == 'r')
 {
  corelog[core].rmb++;

  //copy from memory to l1
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1=1;
  cash1[core][tt1][index1].stat1='s';
```

```
      update_lru1(core, tt1);
     }

     if (type == 'w')
     {
      corelog[core].wmb++;

      // write operation in tt1
      cash1[core][tt1][index1].t1=tag1;
      cash1[core][tt1][index1].v1=1;
      cash1[core][tt1][index1].stat1='m';

      update_lru1(core, tt1);
     }
    }
   }
} //end_only_l1_mesi

void level1_mesi()
{
  if (out1==1) //in case of hit in level-1 private core
  {
    th2=find_l2(core); //find copy in l2

    if (type =='z')
    {
     corelog[core].ihp1++;

     if (cash1[core][ht1][index1].stat1=='m'){corelog[core].mih1++;}
     if (cash1[core][ht1][index1].stat1=='e'){corelog[core].eih1++;}
     if (cash1[core][ht1][index1].stat1=='s'){corelog[core].sih1++;}
    }

    if (type =='r')
    {
     corelog[core].rhp1++;

     if (cash1[core][ht1][index1].stat1=='m'){corelog[core].mrh1++;}
     if (cash1[core][ht1][index1].stat1=='e'){corelog[core].erh1++;}
     if (cash1[core][ht1][index1].stat1=='s'){corelog[core].srh1++;}
    }

    if (type == 'w')
    {
     corelog[core].whp1++;
```

```
   if (cash1[core][ht1][index1].stat1=='m'){corelog[core].mwh1++;}
   if (cash1[core][ht1][index1].stat1=='e'){corelog[core].ewh1++;}
   if (cash1[core][ht1][index1].stat1=='s'){corelog[core].swh1++;}

   // invalidate other copies in bus
   if (cash1[core][ht1][index1].stat1=='s')
   {
    int totalmatch1 = 0; totalmatch1=searchcores1(core);

    for(int i=0; i<totalmatch1; i++)
    {
     cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='i';
     cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].v1=0;
     check_lru1(matchcore1[i].mcore1,matchcore1[i].mthread1);
    }

    int totalmatch2 = 0; totalmatch2=searchcores2(core);

    for(int i=0; i<totalmatch2; i++)
    {
     cash2[matchcore2[i].mcore2][matchcore2[i].mthread2][index2].stat2='i';
     cash2[matchcore2[i].mcore2][matchcore2[i].mthread2][index2].v2=0;
     check_lru2(matchcore2[i].mcore2,matchcore2[i].mthread2);
    }

    if(totalmatch2!=0){corelog[core].bu++;}  //update bus
   }

   //write to private l1 and l2
   cash1[core][ht1][index1].stat1='m';
   cash2[core][th2][index2].stat2='m';
  }
  update_lru1(core, ht1);
  update_lru2(core, th2);
}

if (out1==0) //miss in l1
{
  int tt1 = 0; tt1 = highest_lru1(); //tag highest lru for replacement

  // check hit or miss in level-2
  b = 2, m = 1 << asc2;
  for (int i = 0; i < m; i++)
  {
   if (cash2[core][i][index2].v2 == 1)
```

```
    {
     if (cash2[core][i][index2].t2 == tag2)  // in case of hit in level-2
     {
      ht2 = i;
      b = 1;
      out2 = 1;

      if (type=='z'){corelog[core].ihl2++;}
      if (type=='r'){corelog[core].rhl2++;}
      if (type=='w'){corelog[core].whl2++;}

      break;
     }
    }
   }

   if (b == 2) // in case of miss in level-2
   {
    out2 = 0;

    if (type=='z'){corelog[core].iml2++;}
    if (type=='r'){corelog[core].rml2++;}
    if (type=='w'){corelog[core].wml2++;}
   }
  }
}

void level2_mesi()
{
 if (out2 == 1) // hit in level-2
 {
  if (type=='z')
  {
   corelog[core].ihp2++;

   if (cash2[core][ht2][index2].stat2=='m'){corelog[core].mih2++;}
   if (cash2[core][ht2][index2].stat2=='e'){corelog[core].eih2++;}
   if (cash2[core][ht2][index2].stat2=='s'){corelog[core].sih2++;}

   // copy data from l2 to l1
   cash1[core][tt1][index1].t1=tag1;
   cash1[core][tt1][index1].v1=1;
   cash1[core][tt1][index1].stat1 = cash2[core][ht2][index2].stat2;

   update_lru1(core, tt1);
   update_lru2(core, ht2);
```

```
}

if (type=='r')
{
 corelog[core].rhp2++;

 if (cash2[core][ht2][index2].stat2=='m'){corelog[core].mrh2++;}
 if (cash2[core][ht2][index2].stat2=='e'){corelog[core].erh2++;}
 if (cash2[core][ht2][index2].stat2=='s'){corelog[core].srh2++;}

 // copy data from l2 to l1
 cash1[core][tt1][index1].t1=tag1;
 cash1[core][tt1][index1].v1=1;
 cash1[core][tt1][index1].stat1 = cash2[core][ht2][index2].stat2;

 update_lru1(core, tt1);
 update_lru2(core, ht2);
}

if (type=='w')
{
 corelog[core].whp2++;

 if (cash2[core][ht2][index2].stat2=='m'){corelog[core].mwh2++;}
 if (cash2[core][ht2][index2].stat2=='e') {corelog[core].ewh2++;}
 if (cash2[core][ht2][index2].stat2=='s'){corelog[core].swh2++;}

 if( cash2[core][ht2][index2].stat2!='s')
 {
  //invalidate other copies in bus
  int totalmatch1 = 0; totalmatch1=searchcores1(core);
  for(int i=0; i<totalmatch1;i++)
  {
   cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='i';
   cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].v1=0;
   check_lru1(matchcore1[i].mcore1,matchcore1[i].mthread1);
  }

  int totalmatch2 = 0; totalmatch2=searchcores2(core);
  for(int i=0; i<totalmatch2;i++)
  {
   cash2[matchcore2[i].mcore2][matchcore2[i].mthread2][index2].stat2='i';
   cash2[matchcore2[i].mcore2][matchcore2[i].mthread2][index2].v2=0;
   check_lru2(matchcore2[i].mcore2,matchcore2[i].mthread2);
  }
```

```
    if(totalmatch1!=0){corelog[core].bu++;} //update bus
   }

   //write operation to l1
   cash1[core][tt1][index1].t1=tag1;
   cash1[core][tt1][index1].v1=1;
   cash1[core][tt1][index1].stat1='m';

   // update l2
   cash2[core][ht2][index2].stat2='m';

   update_lru1(core, tt1);
   update_lru2(core, ht2);
  }
}

if (out2 == 0) // miss in level-2
{
 int tt2=0; tt2=highest_lru2(); // find highest lru in L2

 // find hit or miss in bus
 int totalmatch2 = 0; totalmatch2=searchcores2(core);

 if(totalmatch2!=0) //hit in bus
 {
  corelog[core].ctoc++;

  if (type=='z')
  {
   corelog[core].ihb++;

   if (cash2[core][tt2][index2].stat2=='m'){cash0log.wmem++;} // write to memory

   //copy from bus to target l1 and l2
   cash1[core][tt1][index1].t1=tag1;
   cash1[core][tt1][index1].v1=1;
   cash1[core][tt1][index1].stat1='s';
   cash2[core][tt2][index2].t2=tag2;
   cash2[core][tt2][index2].v2=1;
   cash2[core][tt2][index2].stat2='s';

   //change state to 'shared' of copies in bus
   int totalmatch1=0; totalmatch1=searchcores1(core);
   for (int i=0; i<totalmatch1; i++)
   {
    if (matchcore1[i].mstat1!='s')
```

```
  {
    cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='s';
  }
 }

 // change state to shared of copies in bus
 int totalmatch2=0; totalmatch2=searchcores2(core);
 for (int i=0; i<totalmatch2; i++)
 {
  if (matchcore2[i].mstat2!='s')
  {
    cash2[matchcore2[i].mcore2][matchcore2[i].mthread2][index2].stat2='s';
  }
 }

 update_lru1(core, tt1);
 update_lru2(core, tt2);
}

if (type=='r')
{
 //log
 corelog[core].rhb++;

 if (cash2[core][tt2][index2].stat2=='m'){cash0log.wmem++;} // write to memory

 //copy from bus hit thread to target l1 and l2
 cash1[core][tt1][index1].t1=tag1;
 cash1[core][tt1][index1].v1=1;
 cash1[core][tt1][index1].stat1='s';
 cash2[core][tt2][index2].t2=tag2;
 cash2[core][tt2][index2].v2=1;
 cash2[core][tt2][index2].stat2='s';

 // change state to shared of copies in bus
 int totalmatch1=0; totalmatch1=searchcores1(core);
 for (int i=0; i<totalmatch1; i++)
 {
  if (matchcore1[i].mstat1!='s')
  {
    cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='s';
  }
 }

 // state change in other cores l2
 int totalmatch2=0; totalmatch2=searchcores2(core);
```

```
 for (int i=0; i<totalmatch2; i++)
 {
  if (matchcore2[i].mstat2!='s')
  {
   cash2[matchcore2[i].mcore2][matchcore2[i].mthread2][index2].stat2='s';
  }
 }

 update_lru1(core, tt1);
 update_lru2(core, tt2);
}

if (type=='w')
{
 corelog[core].whb++;

 if (cash2[core][tt2][index2].stat2=='m'){cash0log.wmem++;} // write to memory

 //write operation to target l1 and l2 cache
 cash1[core][tt1][index1].t1=tag1;
 cash1[core][tt1][index1].v1=1;
 cash1[core][tt1][index1].stat1='m';
 cash2[core][tt2][index2].t2=tag2;
 cash2[core][tt2][index2].v2=1;
 cash2[core][tt2][index2].stat2='m';

 //invalid copies in bus
 int totalmatch1=0; totalmatch1=searchcores1(core);

 for(int i=0; i<totalmatch1;i++)
 {
  cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='i';
  cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].v1=0;
  check_lru1(matchcore1[i].mcore1,matchcore1[i].mthread1);
 }

 int totalmatch2=0; totalmatch2=searchcores2(core);
 for(int i=0; i<totalmatch2;i++)
 {
  cash2[matchcore2[i].mcore2][matchcore2[i].mthread2][index2].stat2='i';
  cash2[matchcore2[i].mcore2][matchcore2[i].mthread2][index2].v2=0;
  check_lru2(matchcore2[i].mcore2,matchcore2[i].mthread2);
 }
 update_lru1(core, tt1);
 update_lru2(core, tt2);
}
```

```
}

if(totalmatch2==0) //miss in bus
{
 if (type == 'z')
 {
  corelog[core].imb++;

  if (cash2[core][tt2][index2].stat2=='m'){cash0log.wmem++;}  // write to memory

  // copy from memory to l1 and l2
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1=1;
  cash1[core][tt1][index1].stat1='e';
  cash2[core][tt2][index2].t2=tag2;
  cash2[core][tt2][index2].v2=1;
  cash2[core][tt2][index2].stat2='e';

  update_lru1(core, tt1);
  update_lru2(core, tt2);
 }

 if (type == 'r')
 {
  corelog[core].rmb++;

  if (cash2[core][tt2][index2].stat2=='m'){cash0log.wmem++;} // write to memory

  // copy from memory to l1 and l2 cache
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1=1;
  cash1[core][tt1][index1].stat1='e';
  cash2[core][tt2][index2].t2=tag2;
  cash2[core][tt2][index2].v2=1;
  cash2[core][tt2][index2].stat2='e';

  update_lru1(core, tt1);
  update_lru2(core, tt2);
 }

 if (type == 'w')
 {
  corelog[core].wmb++;

  if (cash2[core][tt2][index2].stat2=='m'){cash0log.wmem++;} // write to memory
```

```
      // write to l1 and l2
      cash2[core][tt2][index2].t2=tag2;
      cash2[core][tt2][index2].v2=1;
      cash2[core][tt2][index2].stat2='m';
      cash1[core][tt1][index1].t1=tag1;
      cash1[core][tt1][index1].v1=1;
      cash1[core][tt1][index1].stat1='m';

      update_lru1(core, tt1);
      update_lru2(core, tt2);
    }
   }
  }
}

// snoopy protocol
void only_l1_snoopy()
{
 if (out1==1) //in case of hit in level-1 private core
 {

   if (type =='z')
   {
    corelog[core].ihp1++;

    if (cash1[core][ht1][index1].stat1=='m') {corelog[core].mih1++;}
    if (cash1[core][ht1][index1].stat1=='s'){corelog[core].sih1++;}
   }

   if (type =='r')
   {
    corelog[core].rhp1++;

    if (cash1[core][ht1][index1].stat1=='m'){corelog[core].mrh1++;}
    if (cash1[core][ht1][index1].stat1=='s'){corelog[core].srh1++;}
   }

   if (type == 'w')
   {
    corelog[core].whp1++;

    if (cash1[core][ht1][index1].stat1=='m'){corelog[core].mwh1++;}
    if (cash1[core][ht1][index1].stat1=='s'){corelog[core].swh1++;}

    // invalidate other copies in bus
    if (cash1[core][ht1][index1].stat1=='s')
```

```
   {
    int totalmatch1 = 0; totalmatch1=searchcores1(core);

    for(int i=0; i<totalmatch1; i++)
    {
     cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='i';
     cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].v1=0;
     check_lru1(matchcore1[i].mcore1,matchcore1[i].mthread1);
    }

    if(totalmatch1!=0){corelog[core].bu++;} //update bus
   }

   cash1[core][ht1][index1].stat1='m'; //write to l1
  }

  update_lru1(core, ht1);
}

if (out1==0) //miss in l1
{
 int tt1 = 0; tt1 = highest_lru1(); //tag highest lru for replacement
 int totalmatch1 = 0; totalmatch1=searchcores1(core);

 if(totalmatch1!=0) //hit in bus
 {
  corelog[core].ctoc++;

  if (type=='z')
  {
   corelog[core].ihb++;

   //copy from hit thread to target l1
   cash1[core][tt1][index1].t1=tag1;
   cash1[core][tt1][index1].v1=1;
   cash1[core][tt1][index1].stat1='s';

   //change state to shared of copies in bus
   int totalmatch1=0; totalmatch1=searchcores1(core);
   for (int i=0; i<totalmatch1; i++)
   {
    if (matchcore1[i].mstat1!='s')
    {
     cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='s';
    }
   }
```

```
  update_lru1(core, tt1);
}

if (type=='r')
{
  corelog[core].rhb++;

  //copy from hit thread to target l1
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1=1;
  cash1[core][tt1][index1].stat1='s';

  // change state to shared of copies in bus
  int totalmatch1=0; totalmatch1=searchcores1(core);
  for (int i=0; i<totalmatch1; i++)
  {
    if (matchcore1[i].mstat1!='s')
    {
      cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='s';
    }
  }

  update_lru1(core, tt1);
}

if (type=='w')
{
  corelog[core].whb++;

  // write to l1
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1=1;
  cash1[core][tt1][index1].stat1='m';

  //invalidate copies in bus
  int totalmatch1=0; totalmatch1=searchcores1(core);
  for(int i=0; i<totalmatch1;i++)
  {
    cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='i';
    cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].v1=0;
    check_lru1(matchcore1[i].mcore1,matchcore1[i].mthread1);
  }

  update_lru1(core, tt1);
}
```

```
   }

   if(totalmatch1==0) //miss in bus
   {
    if (type == 'z')
    {
     corelog[core].imb++;

     //copy from memory to target l1
     cash1[core][tt1][index1].t1=tag1;
     cash1[core][tt1][index1].v1=1;
     cash1[core][tt1][index1].stat1='s';

     update_lru1(core, tt1);

    }

    if (type == 'r')
    {
     corelog[core].rmb++;

     //copy from memory to target l1
     cash1[core][tt1][index1].t1=tag1;
     cash1[core][tt1][index1].v1=1;
     cash1[core][tt1][index1].stat1='s';

     update_lru1(core, tt1);
    }

    if (type == 'w')
    {
     corelog[core].wmb++;

     // write operation in tt1
     cash1[core][tt1][index1].t1=tag1;
     cash1[core][tt1][index1].v1=1;
     cash1[core][tt1][index1].stat1='m';

     update_lru1(core, tt1);
    }
   }
  }
}

void level1_snoopy()
{
```

```
if (out1==1) //hit in l1
{
 th2=find_l2(core); //find copy in l2

 if (type =='z')
 {
  corelog[core].ihp1++;

  if (cash1[core][ht1][index1].stat1=='m'){corelog[core].mih1++;}
  if (cash1[core][ht1][index1].stat1=='s'){corelog[core].sih1++;}
 }

 if (type =='r')
 {
  corelog[core].rhp1++;

  if (cash1[core][ht1][index1].stat1=='m'){corelog[core].mrh1++;}
  if (cash1[core][ht1][index1].stat1=='s'){corelog[core].srh1++;}
 }

 if (type == 'w')
 {
  corelog[core].whp1++;

  if (cash1[core][ht1][index1].stat1=='m'){corelog[core].mwh1++;}
  if (cash1[core][ht1][index1].stat1=='s'){corelog[core].swh1++;}

  // invalidate other copies in bus
  if (cash1[core][ht1][index1].stat1=='s')
  {
   int totalmatch1 = 0; totalmatch1=searchcores1(core);

   for(int i=0; i<totalmatch1; i++)
   {
    cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='i';
    cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].v1=0;
    check_lru1(matchcore1[i].mcore1,matchcore1[i].mthread1);
   }

   int totalmatch2 = 0; totalmatch2=searchcores2(core);
   for(int i=0; i<totalmatch2; i++)
   {
    cash2[matchcore2[i].mcore2][matchcore2[i].mthread2][index2].stat2='i';
    cash2[matchcore2[i].mcore2][matchcore2[i].mthread2][index2].v2=0;
    check_lru2(matchcore2[i].mcore2,matchcore2[i].mthread2);
   }
```

```
    if(totalmatch1!=0){corelog[core].bu++;} //update bus
   }

   //write to l1 and l2
   cash1[core][ht1][index1].stat1='m';
   cash2[core][th2][index2].stat2='m';
  }
  update_lru1(core, ht1);
  update_lru2(core, th2);
 }

 if (out1==0) //miss in l1
 {
  int tt1 = 0; tt1 = highest_lru1(); //tag highest lru for replacement

  // check hit or miss in level-2
  b = 2, m = 1 << asc2;
  for (int i = 0; i < m; i++)
  {
   if (cash2[core][i][index2].v2 == 1)
   {
    if (cash2[core][i][index2].t2 == tag2)
    {
     ht2 = i;
     b = 1;
     out2 = 1; // in case of hit in level-2

     // log
     if (type=='z'){corelog[core].ihl2++;}
     if (type=='r'){corelog[core].rhl2++;}
     if (type=='w'){corelog[core].whl2++;}

     break;
    }
   }
  }

  if (b == 2) // in case of miss in level-2
  {
   out2 = 0;

   if (type=='z'){corelog[core].iml2++;}
   if (type=='r'){corelog[core].rml2++;}
   if (type=='w'){corelog[core].wml2++;}
  }
```

```
  }
}

void level2_snoopy()
{
 if (out2 == 1) // hit in level-2
 {
  corelog[core].exchange++;

  if (type=='z')
  {
   corelog[core].ihp2++;

   if (cash2[core][ht2][index2].stat2=='m'){corelog[core].mih2++;}
   if (cash2[core][ht2][index2].stat2=='s'){corelog[core].sih2++;}

   // copy data from l2 to l1
   cash1[core][tt1][index1].t1=tag1;
   cash1[core][tt1][index1].v1=1;
   cash1[core][tt1][index1].stat1 = cash2[core][ht2][index2].stat2;

   update_lru1(core, tt1);
   update_lru2(core, ht2);
  }

  if (type=='r')
  {
   corelog[core].rhp2++;

   if (cash2[core][ht2][index2].stat2=='m'){corelog[core].mrh2++;}
   if (cash2[core][ht2][index2].stat2=='s'){corelog[core].srh2++;}

   // copy data from l2 to l1
   cash1[core][tt1][index1].t1=tag1;
   cash1[core][tt1][index1].v1=1;
   cash1[core][tt1][index1].stat1 = cash2[core][ht2][index2].stat2;

   update_lru1(core, tt1);
   update_lru2(core, ht2);
  }

  if (type=='w')
  {
   corelog[core].whp2++;

   if (cash2[core][ht2][index2].stat2=='m'){corelog[core].mwh2++;}
```

```
   if (cash2[core][ht2][index2].stat2=='s'){corelog[core].swh2++;}

   if( cash2[core][ht2][index2].stat2!='s') //invalidate other copies in bus
   {
    int totalmatch1 = 0; totalmatch1=searchcores1(core);
    for(int i=0; i<totalmatch1;i++)
    {
     cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='i';
     cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].v1=0;
     check_lru1(matchcore1[i].mcore1,matchcore1[i].mthread1);
    }

    int totalmatch2 = 0; totalmatch2=searchcores2(core);
    for(int i=0; i<totalmatch2;i++)
    {
     cash2[matchcore2[i].mcore2][matchcore2[i].mthread2][index2].stat2='i';
     cash2[matchcore2[i].mcore2][matchcore2[i].mthread2][index2].v2=0;
     check_lru2(matchcore2[i].mcore2,matchcore2[i].mthread2);
    }

    if(totalmatch2!=0){corelog[core].bu++;} //update bus
   }

   //write operation in target l1
   cash1[core][tt1][index1].t1=tag1;
   cash1[core][tt1][index1].v1=1;
   cash1[core][tt1][index1].stat1='m';

   // update l2
   cash2[core][ht2][index2].stat2='m';

   update_lru1(core, tt1);
   update_lru2(core, ht2);
  }
 }

if (out2 == 0) // miss in level-2
{
  int tt2=0; tt2=highest_lru2();    // find highest lru in L2

  // find hit or miss in bus
  int totalmatch2 = 0; totalmatch2=searchcores2(core);

  if(totalmatch2!=0) //hit in bus
  {
   corelog[core].ctoc++;
```

```
if (type=='z')
{
 corelog[core].ihb++;

 if (cash2[core][tt2][index2].stat2=='m'){cash0log.wmem++;}  // write to memory

 //copy from bus to target l and l2
 cash1[core][tt1][index1].t1=tag1;
 cash1[core][tt1][index1].v1=1;
 cash1[core][tt1][index1].stat1='s';
 cash2[core][tt2][index2].t2=tag2;
 cash2[core][tt2][index2].v2=1;
 cash2[core][tt2][index2].stat2='s';

 //change state to 'shared' for copies in bus
 int totalmatch1=0; totalmatch1=searchcores1(core);
 for (int i=0; i<totalmatch1; i++)
 {
  if (matchcore1[i].mstat1!='s')
  {
   cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='s';
  }
 }

 int totalmatch2=0;  totalmatch2=searchcores2(core);
 for (int i=0; i<totalmatch2; i++)
 {
  if (matchcore2[i].mstat2!='s')
  {
   cash2[matchcore2[i].mcore2][matchcore2[i].mthread2][index2].stat2='s';
  }
 }
 update_lru1(core, tt1);
 update_lru2(core, tt2);
}

if (type=='r')
{
 corelog[core].rhb++;

 if (cash2[core][tt2][index2].stat2=='m')
 {cash0log.wmem++;}  // write to memory

 //copy from bus to target l1 and l2
 cash1[core][tt1][index1].t1=tag1;
```

```
  cash1[core][tt1][index1].v1=1;
  cash1[core][tt1][index1].stat1='s';
  cash2[core][tt2][index2].t2=tag2;
  cash2[core][tt2][index2].v2=1;
  cash2[core][tt2][index2].stat2='s';

  // change state to 'shared' for copies in bus
  int totalmatch1=0; totalmatch1=searchcores1(core);
  for (int i=0; i<totalmatch1; i++)
  {
   if (matchcore1[i].mstat1!='s')
   {
    cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='s';
   }
  }

  int totalmatch2=0; totalmatch2=searchcores2(core);
  for (int i=0; i<totalmatch2; i++)
  {
   if (matchcore2[i].mstat2!='s')
   {
    cash2[matchcore2[i].mcore2][matchcore2[i].mthread2][index2].stat2='s';
   }
  }
  update_lru1(core, tt1);
  update_lru2(core, tt2);
 }

 if (type=='w')
 {
  corelog[core].whb++;

  if (cash2[core][tt2][index2].stat2=='m'){cash0log.wmem++;}  // write to memory

  // write to l1 and l2
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1=1;
  cash1[core][tt1][index1].stat1='m';

  //update l2
  cash2[core][tt2][index2].t2=tag2;
  cash2[core][tt2][index2].v2=1;
  cash2[core][tt2][index2].stat2='m';

  //invalid copies in bus
  int totalmatch1=0; totalmatch1=searchcores1(core);
```

```
   for(int i=0; i<totalmatch1;i++)
   {
    cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='i';
    cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].v1=0;
    check_lru1(matchcore1[i].mcore1,matchcore1[i].mthread1);
   }

   int totalmatch2=0; totalmatch2=searchcores2(core);
   for(int i=0; i<totalmatch2;i++)
   {
    cash2[matchcore2[i].mcore2][matchcore2[i].mthread2][index2].stat2='i';
    cash2[matchcore2[i].mcore2][matchcore2[i].mthread2][index2].v2=0;
    check_lru2(matchcore2[i].mcore2,matchcore2[i].mthread2);
   }
   update_lru1(core, tt1);
   update_lru2(core, tt2);
  }
}

if(totalmatch2==0) //miss in bus
{
 if (type == 'z')
 {
  corelog[core].imb++;

  if (cash2[core][tt2][index2].stat2=='m'){cash0log.wmem++; } // write to memory

  // copy from memory to l1 and l2
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1=1;
  cash1[core][tt1][index1].stat1='s';
  cash2[core][tt2][index2].t2=tag2;
  cash2[core][tt2][index2].v2=1;
  cash2[core][tt2][index2].stat2='s';

  update_lru1(core, tt1);
  update_lru2(core, tt2);
 }

 if (type == 'r')
 {
  corelog[core].rmb++;

  if (cash2[core][tt2][index2].stat2=='m'){cash0log.wmem++;} // write to memory

  // copy from memory to l1 and l2
```

```
      cash1[core][tt1][index1].t1=tag1;
      cash1[core][tt1][index1].v1=1;
      cash1[core][tt1][index1].stat1='s';
      cash2[core][tt2][index2].t2=tag2;
      cash2[core][tt2][index2].v2=1;
      cash2[core][tt2][index2].stat2='s';

      update_lru1(core, tt1);
      update_lru2(core, tt2);
    }

    if (type == 'w')
    {
      corelog[core].wmb++;

      if (cash2[core][tt2][index2].stat2=='m'){cash0log.wmem++;}  // write to memory

      // write operation in l1
      cash1[core][tt1][index1].t1=tag1;
      cash1[core][tt1][index1].v1=1;
      cash1[core][tt1][index1].stat1='m';

      //update l2
      cash2[core][tt2][index2].t2=tag2;
      cash2[core][tt2][index2].v2=1;
      cash2[core][tt2][index2].stat2='m';

      update_lru1(core, tt1);
      update_lru2(core, tt2);
    }
   }
  }
}

// firefly protocol
void only_l1_firefly()
{
  if (out1==1) //in case of hit in level-1 private core
  {
    if (type =='r')
    {
      corelog[core].rhp1++;

    if (cash1[core][ht1][index1].stat1=='v'){corelog[core].vrh1++;}
    if (cash1[core][ht1][index1].stat1=='d'){corelog[core].drh1++;}
    if (cash1[core][ht1][index1].stat1=='s'){corelog[core].srh1++;}
```

```
 }

 if (type == 'w')
 {
  corelog[core].whp1++;

  if (cash1[core][ht1][index1].stat1=='v'){corelog[core].vwh1++;}
  if (cash1[core][ht1][index1].stat1=='d'){corelog[core].dwh1++;}
  if (cash1[core][ht1][index1].stat1=='s'){corelog[core].swh1++;}

  if (cash1[core][ht1][index1].stat1=='v')
  {cash1[core][ht1][index1].stat1='d';}

  else if (cash1[core][ht1][index1].stat1=='s')
  {
   int totalmatch1=0; totalmatch1=searchcores1(core);

   if (totalmatch1=!0)
   {
    cash1[core][ht1][index1].stat1='s';
    corelog[core].bu++; // bus update
   }

   if (totalmatch1==0)
   {cash1[core][ht1][index1].stat1='d';}
  }
 }

 update_lru1(core,ht1);
}

if (out1==0) // when miss in level-1 private core
{
 int tt1, totalmatch1=0;
 tt1 = highest_lru1(); totalmatch1 = searchcores1(core);

 if (totalmatch1!=0) // hit in level-1 other cores
 {
  corelog[core].ctoc++;

  if (type =='z')
  {
   corelog[core].ihb++;

   // copy from bus to l1
   cash1[core][tt1][index1].t1=tag1;
```

```
 cash1[core][tt1][index1].v1=1;
 cash1[core][tt1][index1].stat1='s';

 // change state to 'shared' for other copies
 for (int i=0; i<totalmatch1; i++)
 {
  if (matchcore1[i].mstat1!='s')
  {
   cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='s';
  }
 }
 update_lru1(core, tt1);
}

if (type =='r')
{
 corelog[core].rhb++;

 // copy from bus to l1
 cash1[core][tt1][index1].t1=tag1;
 cash1[core][tt1][index1].v1=1;
 cash1[core][tt1][index1].stat1='s';

 // change state to 'shared' for other copies
 for (int i=0; i<totalmatch1; i++)
 {
  if (matchcore1[i].mstat1!='s')
  {
   cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='s';
  }
 }

 update_lru1(core, tt1);
}

if (type =='w')
{
 corelog[core].whb++;

 // copy from bus to l1
 cash1[core][tt1][index1].t1=tag1;
 cash1[core][tt1][index1].v1=1;
 cash1[core][tt1][index1].stat1='s';

 // change state to 'shared' for copies in bus
 for(int i=0; i<totalmatch1;i++)
```

```
   {
    cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='s';
    cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].v1=1;
    check_lru1(matchcore1[i].mcore1,matchcore1[i].mthread1);
   }

   update_lru1(core, tt1);
  }
 }

 if (totalmatch1==0) // miss in bus
 {
  if (type=='z')
  {
   corelog[core].imb++;

   //copy from memory to l1
   cash1[core][tt1][index1].t1=tag1;
   cash1[core][tt1][index1].v1=1;
   cash1[core][tt1][index1].stat1='v';

   update_lru1(core, tt1);
  }

  if (type=='r')
  {
   corelog[core].rmb++;

   //copy from memory to l1
   cash1[core][tt1][index1].t1=tag1;
   cash1[core][tt1][index1].v1=1;
   cash1[core][tt1][index1].stat1='v';

   update_lru1(core, tt1);
  }

  if (type=='w')
  {
   corelog[core].wmb++;

   //write to l1
   cash1[core][tt1][index1].t1=tag1;
   cash1[core][tt1][index1].v1=1;
   cash1[core][tt1][index1].stat1='d';
   update_lru1(core, tt1);
  }
```

```
   }
  }
}

void level1_firefly()
{
 if (out1==1) //in case of hit in level-1 private core
 {
  th2=find_l2(core); //find copy in l2

  if (type =='z')
  {
   corelog[core].ihp1++;

   if (cash1[core][ht1][index1].stat1=='v'){corelog[core].vrh1++;}
   if (cash1[core][ht1][index1].stat1=='d'){corelog[core].drh1++;}
   if (cash1[core][ht1][index1].stat1=='s'){corelog[core].srh1++;}
  }

  if (type =='r')
  {
   corelog[core].rhp1++;

   if (cash1[core][ht1][index1].stat1=='v'){corelog[core].vrh1++;}
   if (cash1[core][ht1][index1].stat1=='d'){corelog[core].drh1++;}
   if (cash1[core][ht1][index1].stat1=='s'){corelog[core].srh1++;}
  }

  if (type == 'w')
  {
   corelog[core].whp1++;

   if (cash1[core][ht1][index1].stat1=='v'){corelog[core].vwh1++;}
   if (cash1[core][ht1][index1].stat1=='d'){corelog[core].dwh1++;}
   if (cash1[core][ht1][index1].stat1=='s'){corelog[core].swh1++;}

   if (cash1[core][ht1][index1].stat1=='v')
   {
    cash1[core][ht1][index1].stat1='d';
    cash2[core][th2][index2].stat2='d';
   }

   else if (cash1[core][ht1][index1].stat1=='s')
   {
    int totalmatch1 = 0; totalmatch1=searchcores1(core);
    int totalmatch2 = 0; totalmatch2=searchcores2(core);
```

```
    if (totalmatch2=!0)
    {
      cash1[core][ht1][index1].stat1='s';
      cash2[core][th2][index2].stat2='s';

      corelog[core].bu++;
    }

    if (totalmatch2==0)
    {
      cash1[core][ht1][index1].stat1='d';
      cash2[core][th2][index2].stat2='d';
    }
   }
  }
  update_lru1(core,ht1);
  update_lru2(core,th2);
}

if (out1==0) // when miss in level-1 private core
{
  int tt1 = 0; tt1 = highest_lru1(); //tag highest lru for replacement

  // check hit or miss in level-2
  b = 2, m = 1 << asc2;
  for (int i = 0; i < m; i++)
  {
   if (cash2[core][i][index2].v2 == 1)
   {
    if (cash2[core][i][index2].t2 == tag2)
    {
      ht2 = i;
      b = 1;
      out2 = 1; // in case of hit in level-2

      if (type=='z'){corelog[core].ihl2++;}
      if (type=='r'){corelog[core].rhl2++;}
      if (type=='w'){corelog[core].whl2++;}

      break;
    }
   }
  }

  if (b == 2)
```

```
   {
    out2 = 0; // in case of miss in level-2

    if (type=='z'){corelog[core].iml2++;}
    if (type=='r'){corelog[core].rml2++;}
    if (type=='w'){corelog[core].wml2++;}
   }
  }
}



void level2_firefly()
{
  if (out2 == 1) // hit in level-2
  {
   if (type=='z')
   {
    corelog[core].ihp2++;

    if (cash2[core][ht2][index2].stat2=='d'){corelog[core].dih2++;}
    if (cash2[core][ht2][index2].stat2=='v'){corelog[core].vih2++;}
    if (cash2[core][ht2][index2].stat2=='s'){corelog[core].sih2++;}

    // copy data from l2 to l1
    cash1[core][tt1][index1].t1=tag1;
    cash1[core][tt1][index1].v1=1;
    cash1[core][tt1][index1].stat1 = cash2[core][ht2][index2].stat2;

    update_lru1(core, tt1);
    update_lru2(core, ht2);
   }

   if (type=='r')
   {
    corelog[core].rhp2++;

    if (cash2[core][ht2][index2].stat2=='d'){corelog[core].drh2++;}
    if (cash2[core][ht2][index2].stat2=='v'){corelog[core].vrh2++;}
    if (cash2[core][ht2][index2].stat2=='s'){corelog[core].srh2++;}

    // copy data from l2 to l1
    cash1[core][tt1][index1].t1=tag1;
    cash1[core][tt1][index1].v1=1;
    cash1[core][tt1][index1].stat1 = cash2[core][ht2][index2].stat2;
```

```
 update_lru1(core, tt1);
 update_lru2(core, ht2);
}

if (type=='w')
{
 corelog[core].whp2++;

 if (cash2[core][ht2][index2].stat2=='m'){corelog[core].dwh2++;}
 if (cash2[core][ht2][index2].stat2=='v'){corelog[core].vwh2++;}
 if (cash2[core][ht2][index2].stat2=='s'){corelog[core].swh2++;}

 cash1[core][tt1][index1].t1=tag1;
 cash1[core][tt1][index1].v1=1;

 // check for copies in bus
 int totalmatch1=0; totalmatch1=searchcores1(core);
 int totalmatch2=0; totalmatch2=searchcores2(core);

 if (totalmatch2!=0)
 {
  for(int i=0; i<totalmatch1;i++)
  {
   cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='s';
   update_lru1(matchcore1[i].mcore1,matchcore1[i].mthread1);
  }

  for(int i=0; i<totalmatch2;i++)
  {
   cash2[matchcore2[i].mcore2][matchcore2[i].mthread2][index2].stat2='s';
   update_lru2(matchcore2[i].mcore2,matchcore2[i].mthread2);
  }
  cash1[core][tt1][index1].stat1='s';
  cash2[core][tt2][index2].stat2='s';
 }

 if (totalmatch2==0)
 {
  cash1[core][tt1][index1].stat1='d';
  cash2[core][tt2][index2].stat2='d';
 }

 if(totalmatch2!=0) {corelog[core].bu++;}  //update bus
 update_lru1(core, tt1);
 update_lru2(core, ht2);
}
```

```
}

if (out2 == 0) // miss in level-2
{
 int tt2=0; tt2=highest_lru2(); // find highest lru in L2

 // find hit or miss in bus
 int totalmatch1 = 0; totalmatch1=searchcores1(core);
 int totalmatch2 = 0; totalmatch2=searchcores2(core);

 if(totalmatch2!=0) //hit in bus
 {
  corelog[core].ctoc++;

  if (cash2[core][tt2][index2].stat2=='d') {cash0log.wmem++;} // write to memory

  if (type=='z')
  {
   corelog[core].ihb++;

   if (cash2[core][tt2][index2].stat2=='d'){cash0log.wmem++;}  // write to memory

   // change state to shared of copies in bus
   for (int i=0; i<totalmatch1; i++)
   {
    if (matchcore1[i].mstat1!='s')
    {
     cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='s';
    }
   }

   for (int i=0; i<totalmatch2; i++)
   {
    if (matchcore2[i].mstat2!='s')
    {
     cash2[matchcore2[i].mcore2][matchcore2[i].mthread2][index2].stat2='s';
    }
   }
  }

  if (type=='r')
  {
   corelog[core].rhb++;

   if (cash2[core][tt2][index2].stat2=='d') {cash0log.wmem++;}  // write to memory
```

```
 // change state to shared for copies in bus
 for (int i=0; i<totalmatch1; i++)
 {
  if (matchcore1[i].mstat1!='s')
  {
   cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='s';
  }
 }

 for (int i=0; i<totalmatch2; i++)
 {
  if (matchcore2[i].mstat2!='s')
  {
   cash2[matchcore2[i].mcore2][matchcore2[i].mthread2][index2].stat2='s';
  }
 }
}

if (type=='w')
{
 corelog[core].whb++;

 if (cash2[core][tt2][index2].stat2=='d') {cash0log.wmem++;} // write to memory

 //change state to shared for copies in bus
 for(int i=0; i<totalmatch1;i++)
 {
  cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].stat1='s';
  cash1[matchcore1[i].mcore1][matchcore1[i].mthread1][index1].v1=1;
  check_lru1(matchcore1[i].mcore1,matchcore1[i].mthread1);
 }

 for(int i=0; i<totalmatch2;i++)
 {
  cash2[matchcore2[i].mcore2][matchcore2[i].mthread2][index2].stat2='s';
  cash2[matchcore2[i].mcore2][matchcore2[i].mthread2][index2].v2=1;
  check_lru2(matchcore2[i].mcore2,matchcore2[i].mthread2);
 }
}

//copy from bus to target l1 and l1
cash1[core][tt1][index1].t1=tag1;
cash1[core][tt1][index1].v1=1;
cash1[core][tt1][index1].stat1='s';
cash2[core][tt2][index2].t2=tag2;
cash2[core][tt2][index2].v2=1;
```

```
 cash2[core][tt2][index2].stat2='s';

 update_lru1(core, tt1);
 update_lru2(core, tt2);
}

if(totalmatch2==0) //miss in bus
{
 if (type == 'z')
 {
  corelog[core].imb++;

  if (cash2[core][tt2][index2].stat2=='d'){cash0log.wmem++;}  // write to memory

  // copy from memory to l1 and l2
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1=1;
  cash1[core][tt1][index1].stat1='v';
  cash2[core][tt2][index2].t2=tag2;
  cash2[core][tt2][index2].v2=1;
  cash2[core][tt2][index2].stat2='v';
 }

 if (type == 'r')
 {
  corelog[core].rmb++;

  if (cash2[core][tt2][index2].stat2=='d') {cash0log.wmem++;}  // write to memory

  // copy from memory to l1 and l2
  cash1[core][tt1][index1].t1=tag1;
  cash1[core][tt1][index1].v1=1;
  cash1[core][tt1][index1].stat1='v';
  cash2[core][tt2][index2].t2=tag2;
  cash2[core][tt2][index2].v2=1;
  cash2[core][tt2][index2].stat2='v';
 }

 if (type == 'w')
 {
  corelog[core].wmb++;

  if (cash2[core][tt2][index2].stat2=='d'){cash0log.wmem++;} // write to memory

  // copy from memory to l1 and l2
  cash2[core][tt2][index2].t2=tag2;
```

```
      cash2[core][tt2][index2].v2=1;
      cash2[core][tt2][index2].stat2='d';
      cash1[core][tt1][index1].t1=tag1;
      cash1[core][tt1][index1].v1=1;
      cash1[core][tt1][index1].stat1='d';
     }
     update_lru1(core, tt1);
     update_lru2(core, tt2);
   }
  }
}

void log_sc_only_l1()
{
  for (int i=0; i<coreno; i++)
  {

totalins=totalins+corelog[i].rhl1+corelog[i].rml1+corelog[i].whl1+corelog[i].wml1+corel
og[i].ihl1+corelog[i].iml1;
    totalrml1= totalrml1+corelog[i].rml1;
    totalwml1= totalwml1+corelog[i].wml1;
    totaliml1= totaliml1+corelog[i].iml1;
    totalmissl1=totalmissl1+corelog[i].rml1+corelog[i].wml1+corelog[i].iml1;
  }
  cout<<"\ntotal number of instruction: "<<totalins<<endl;
  cout << "\ntotal misses: "<<totalmissl1<<endl;
  cout<<"l1 cache miss rate : "<<((totalmissl1)*100)/totalins<<"%"<<endl;
  cout<<"read miss : "<< totalrml1<<endl;
  cout<<"write miss: "<< totalwml1<<endl;
  cout<<"instruction miss: "<< totaliml1<<endl;
  cout << "\nread from memory: "<< cash0log.rmem<< endl;
}

void log_sc()
{
  for (int i=0; i<coreno; i++)
  {

totalins=totalins+corelog[i].rhl1+corelog[i].rml1+corelog[i].whl1+corelog[i].wml1+corel
og[i].ihl1+corelog[i].iml1;
    totalrml1= totalrml1+corelog[i].rml1;
    totalwml1= totalwml1+corelog[i].wml1;
    totaliml1= totaliml1+corelog[i].iml1;
    totalmissl1=totalmissl1+corelog[i].rml1+corelog[i].wml1+corelog[i].iml1;
    totalrml2= totalrml2+corelog[i].rml2;
    totalwml2= totalwml2+corelog[i].wml2;
```

```
   totaliml2= totaliml2+corelog[i].iml2;
   totalmissl2=totalmissl2+corelog[i].rml2+corelog[i].wml2+corelog[i].iml2;
  }
  cout<<"\ntotal number of instruction: "<<totalins<<endl;
  cout << "\nmisses in l1: "<<totalmissl1<<endl;
  cout<<"l1 cache miss rate : "<<((totalmissl1)*100)/totalins<<"%"<<endl;
  cout<<"l1 read miss : "<< totalrml1<<endl;
  cout<<"l1 write miss: "<< totalwml1<<endl;
  cout<<"l1 instruction miss: "<< totaliml1<<endl;

  cout << "\nl2 cache call: "<< totalmissl1 << endl;
  cout << "misses in l2: "<<totalmissl2<< endl;
  cout << "l2 cache miss rate: "<<((totalmissl2/totalmissl1)*100)<<" %"<<endl;
  cout<<"l2 read miss : "<< totalrml2<<endl;
  cout<<"l2 write miss: "<< totalwml2<<endl;
  cout<<"l2 instruction miss: "<< totaliml2<<endl;
  cout << "\nmiss rate in two level of cache: " << ((totalmissl2/totalins)*100)<<endl;
  cout << "\nread from memory: "<< cash0log.rmem<< endl;
}

void log_mesi_only_l1()
{
  for (int i=0; i<coreno; i++)
  {

totalins=totalins+corelog[i].rhl1+corelog[i].rml1+corelog[i].whl1+corelog[i].wml1+corel
og[i].ihl1+corelog[i].iml1;
   totalrml1= totalrml1+corelog[i].rml1;
   totalwml1= totalwml1+corelog[i].wml1;
   totaliml1= totaliml1+corelog[i].iml1;
   totalmissl1=totalmissl1+corelog[i].rml1+corelog[i].wml1+corelog[i].iml1;
   totalctoc=totalctoc+corelog[i].ctoc;

bustraffic=bustraffic+corelog[i].imb+corelog[i].ihb+corelog[i].rmb+corelog[i].rhb+corel
og[i].wmb+corelog[i].whb+corelog[i].bu;
  }

  for ( int i=0; i<coreno;i++)
  {
   cout << "\n >>>>> LOG CORE: "<<i<<" <<<<<"<<endl;
   cout<<"\nnumber of instruction in core:
"<<(corelog[i].rhl1+corelog[i].rml1+corelog[i].whl1+corelog[i].wml1+corelog[i].ihl1+co
relog[i].iml1)<<endl;
   cout << "\nl1 cache miss rate: "<<
((corelog[i].rml1+corelog[i].wml1+corelog[i].iml1)/(corelog[i].rhl1+corelog[i].rml1+cor
elog[i].whl1+corelog[i].wml1+corelog[i].ihl1+corelog[i].iml1)*100)<<"%"<<endl;
```

```
   cout << "l1 read miss:"<<corelog[i].rml1<< endl;
   cout << "l1 write miss:"<<corelog[i].wml1<< endl;
   cout << "l1 other ins miss:"<<corelog[i].iml1<< endl;
   cout << "\n#data transfer amoung caches: "<< corelog[i].ctoc<<endl;
  }

  cout<<"\ntotal number of instruction: "<<totalins<<endl;
  cout<<"\nglobal miss rate: "<< (((totalmissl1-totalctoc)*100)/totalins)<<"%"<<endl;
  cout<<"average miss rate in l1: "<<(totalmissl1/totalins)*100<<"%"<<endl;
  cout <<"\ntotal #data transfer among caches: "<<totalctoc<<endl;
  cout <<"bus traffic: "<<bustraffic<<endl;
  cout <<"read from memory: "<<(totalmissl1-totalctoc)<<endl;
}

void log_mesi()
{
  for (int i=0; i<coreno; i++)
  {

totalins=totalins+corelog[i].rhl1+corelog[i].rml1+corelog[i].whl1+corelog[i].wml1+corel
og[i].ihl1+corelog[i].iml1;

    totalrml1= totalrml1+corelog[i].rml1;
    totalwml1= totalwml1+corelog[i].wml1;
    totaliml1= totaliml1+corelog[i].iml1;
    totalmissl1=totalmissl1+corelog[i].rml1+corelog[i].wml1+corelog[i].iml1;
    totalrml2= totalrml2+corelog[i].rml2;
    totalwml2= totalwml2+corelog[i].wml2;
    totaliml2= totaliml2+corelog[i].iml2;
    totalmissl2=totalmissl2+corelog[i].rml2+corelog[i].wml2+corelog[i].iml2;
    totalctoc=totalctoc+corelog[i].ctoc;

bustraffic=bustraffic+corelog[i].imb+corelog[i].ihb+corelog[i].rmb+corelog[i].rhb+corel
og[i].wmb+corelog[i].whb+corelog[i].bu;
  }

  for ( int i=0; i<coreno;i++)
  {
    cout << "\n >>>>> LOG CORE: "<<i<<" <<<<<"<<endl;

    cout<<"\nnumber of instruction in core:
"<<(corelog[i].rhl1+corelog[i].rml1+corelog[i].whl1+corelog[i].wml1+corelog[i].ihl1+co
relog[i].iml1)<<endl;
```

```
   cout << "\nl1 cache miss rate: "<<
((corelog[i].rml1+corelog[i].wml1+corelog[i].iml1)/(corelog[i].rhl1+corelog[i].rml1+cor
elog[i].whl1+corelog[i].wml1+corelog[i].ihl1+corelog[i].iml1)*100)<<"%"<<endl;
   cout << "l1 read miss:"<<corelog[i].rml1<< endl;
   cout << "l1 write miss:"<<corelog[i].wml1<< endl;
   cout << "l1 other ins miss:"<<corelog[i].iml1<< endl;
   cout << "l2 cache call: " <<(corelog[i].rml1+corelog[i].wml1+corelog[i].iml1)<<endl;
   cout << "l2 miss rate:
"<<((corelog[i].rml2+corelog[i].wml2+corelog[i].iml2)/(corelog[i].rml1+corelog[i].wml1
+corelog[i].iml1)*100)<<endl;
   cout << "l2 read miss:"<<corelog[i].rml2<< endl;
   cout << "l2 write miss:"<<corelog[i].wml2<< endl;
   cout << "l2 other ins miss:"<<corelog[i].iml2<< endl;
   cout << "\n#data transfer amoung caches: "<< corelog[i].ctoc<<endl;
  }
  cout<<"\ntotal number of instruction: "<<totalins<<endl;
  cout<<"\nglobal miss rate: "<< (((totalmissl2-totalctoc)*100)/totalins)<<"%"<<endl;
  cout<<"miss rate in l1: "<<(totalmissl1/totalins)*100<<"%"<<endl;
  cout << "miss rate in l2: "<<((totalmissl2/totalmissl1)*100)<<"%"<<endl;
  cout << "miss rate in 2 level of cache: "<<((totalmissl2/totalins)*100)<<"%"<<endl;
  cout <<"\ntotal #data transfer among caches: "<<totalctoc<<endl;
  cout <<"bus traffic: "<<bustraffic<<endl;
  cout <<"read from memory: "<<(totalmissl2-totalctoc)<<endl;
}

void log_firefly_only_l1()
{
  for (int i=0; i<coreno; i++)
  {

totalins=totalins+corelog[i].rhl1+corelog[i].rml1+corelog[i].whl1+corelog[i].wml1+corel
og[i].ihl1+corelog[i].iml1;

   totalrml1= totalrml1+corelog[i].rml1;
   totalwml1= totalwml1+corelog[i].wml1;
   totaliml1= totaliml1+corelog[i].iml1;
   totalmissl1=totalmissl1+corelog[i].rml1+corelog[i].wml1+corelog[i].iml1;
   totalctoc=totalctoc+corelog[i].ctoc;

bustraffic=bustraffic+corelog[i].imb+corelog[i].ihb+corelog[i].rmb+corelog[i].rhb+corel
og[i].wmb+corelog[i].whb+corelog[i].bu;
  }

  for ( int i=0; i<coreno;i++)
  {
   cout << "\n >>>>> LOG CORE: "<<i<<" <<<<<"<<endl;
```

```
   cout<<"\nnumber of instruction in core:
"<<(corelog[i].rhl1+corelog[i].rml1+corelog[i].whl1+corelog[i].wml1+corelog[i].ihl1+co
relog[i].iml1)<<endl;
   cout << "\nl1 cache miss rate: "<<
((corelog[i].rml1+corelog[i].wml1+corelog[i].iml1)/(corelog[i].rhl1+corelog[i].rml1+cor
elog[i].whl1+corelog[i].wml1+corelog[i].ihl1+corelog[i].iml1)*100)<<"%"<<endl;
   cout << "l1 read miss:"<<corelog[i].rml1<< endl;
   cout << "l1 write miss:"<<corelog[i].wml1<< endl;
   cout << "l1 other ins miss:"<<corelog[i].iml1<< endl;
   cout << "\n#data transfer amoung caches: "<< corelog[i].ctoc<<endl;
  }
  cout<<"\ntotal number of instruction: "<<totalins<<endl;
  cout<<"\nglobal miss rate: "<< (((totalmissl1-totalctoc)*100)/totalins)<<"%"<<endl;
  cout<<"average miss rate in l1: "<<(totalmissl1/totalins)*100<<"%"<<endl;
  cout <<"\ntotal #data transfer among caches: "<<totalctoc<<endl;
  cout <<"bus traffic: "<<bustraffic<<endl;
  cout <<"read from memory: "<<(totalmissl1-totalctoc)<<endl;
}

void log_firefly()
{
  for (int i=0; i<coreno; i++)
  {

totalins=totalins+corelog[i].rhl1+corelog[i].rml1+corelog[i].whl1+corelog[i].wml1+corel
og[i].ihl1+corelog[i].iml1;
   totalrml1= totalrml1+corelog[i].rml1;
   totalwml1= totalwml1+corelog[i].wml1;
   totaliml1= totaliml1+corelog[i].iml1;
   totalmissl1=totalmissl1+corelog[i].rml1+corelog[i].wml1+corelog[i].iml1;
   totalrml2= totalrml2+corelog[i].rml2;
   totalwml2= totalwml2+corelog[i].wml2;
   totaliml2= totaliml2+corelog[i].iml2;
   totalmissl2=totalmissl2+corelog[i].rml2+corelog[i].wml2+corelog[i].iml2;
   totalctoc=totalctoc+corelog[i].ctoc;

bustraffic=bustraffic+corelog[i].imb+corelog[i].ihb+corelog[i].rmb+corelog[i].rhb+corel
og[i].wmb+corelog[i].whb+corelog[i].bu;
  }

  for ( int i=0; i<coreno;i++)
  {
   cout << "\n >>>>> LOG CORE: "<<i<<" <<<<<"<<endl;
   cout<<"\nnumber of instruction in core:
"<<(corelog[i].rhl1+corelog[i].rml1+corelog[i].whl1+corelog[i].wml1+corelog[i].ihl1+co
relog[i].iml1)<<endl;
```

```
   cout << "\nl1 cache miss rate: "<<
((corelog[i].rml1+corelog[i].wml1+corelog[i].iml1)/(corelog[i].rhl1+corelog[i].rml1+cor
elog[i].whl1+corelog[i].wml1+corelog[i].ihl1+corelog[i].iml1)*100)<<"%"<<endl;
   cout << "l1 read miss:"<<corelog[i].rml1<< endl;
   cout << "l1 write miss:"<<corelog[i].wml1<< endl;
   cout << "l1 other ins miss:"<<corelog[i].iml1<< endl;
   cout << "l2 cache call: " <<(corelog[i].rml1+corelog[i].wml1+corelog[i].iml1)<<endl;
   cout << "l2 miss rate:
"<<((corelog[i].rml2+corelog[i].wml2+corelog[i].iml2)/(corelog[i].rml1+corelog[i].wml1
+corelog[i].iml1)*100)<<endl;
   cout << "l2 read miss:"<<corelog[i].rml2<< endl;
   cout << "l2 write miss:"<<corelog[i].wml2<< endl;
   cout << "l2 other ins miss:"<<corelog[i].iml2<< endl;
   cout << "\n#data transfer amoung caches: "<< corelog[i].ctoc<<endl;
  }
  cout<<"\ntotal number of instruction: "<<totalins<<endl;
  cout<<"\nglobal miss rate: "<< (((totalmissl2-totalctoc)*100)/totalins)<<"%"<<endl;
  cout<<"miss rate in l1: "<<(totalmissl1/totalins)*100<<"%"<<endl;
  cout << "miss rate in l2: "<<((totalmissl2/totalmissl1)*100)<<"%"<<endl;
  cout << "miss rate in 2 level of cache: "<<((totalmissl2/totalins)*100)<<"%"<<endl;
  cout <<"\ntotal #data transfer among caches: "<<totalctoc<<endl;
  cout <<"bus traffic: "<<bustraffic<<endl;
  cout <<"read from memory: "<<(totalmissl2-totalctoc)<<endl;
}
```

BIOGRAPHICAL SKETCH

Provashish Roy was born in Jessore, Bangladesh, to his father, Gour Roy, and mother, Karuna Roy. He attended the Bangladesh University of Engineering and Technology and completed his Bachelor of Science in Electrical and Electronics Engineering, in 2015. He joined the Electrical Engineering Department of the University of Texas Rio Grande Valley with a Presidential Graduate Research Assistantship. He got his master's in science in 2022. His thesis was about memory hierarchies and the design of cache memory. For the next step in his career, he is looking for research opportunities in microprocessor system design and computer architecture. You can contact him via email at royeeebuet@gmail.com.