CS 572: Computer Architecture **(Due: June 10th, 2021)**

Final Project: Multi-level Cache

Instructor: Prof. Justin Goins *Name:* Rashmi Jadhav, *Student ID:* 934-069-574

INTRODUCTION

This report describes at length the implementation and experimentations of a multi-level cache simulator. The cache simulator handles caches with varying capacities, block sizes, levels of associativity (direct-mapped or n-way set associative), replacement policies (random or LRU), and write policies (write-through or write-back). It operates on two input files, a cache configuration file and a tracefile (generated using Valgrind) that indicates memory access instructions. The simulator makes the following assumptions:

1. The data is assumed to be read from the memory as the memory access cycles are utilized.
2. Accessing a sub-portion of a cache block takes the exact same time as it would require to access the entire block.
3. The main memory (RAM) is always accessed in units of 8 bytes at a time. It is expensive to access the first unit from the main memory and successive memory (in 8 byte chunks) takes 1 additional clock cycle per contiguous unit.
4. All caches utilize a "fetch-on-write" scheme if a miss occurs on a Store operation.
5. Instruction read requests are assumed to be handled by a separate instruction cache and hence not processed.
6. The address is a 64 bit hexadecimal number representing the address of the first byte that is being requested.
7. The caches follow pyramid style with L1 being the smallest in size followed by, L2, L3 and eventually RAM.

The output of the cache simulator is a text file which denotes any hits/misses/evictions the occurred on any of the cache levels for each instruction. It mentions the total number of cache hits, misses, and evictions for each cache, the total number of clock cycles utilized, and total CPU reads and CPU writes.

Note: The implementation uses stub code provided by the instructor.

THE CACHE DESIGN

The memory hierarchy is in the form of a pyramid with L1 cache being closest to the CPU and fastest in terms of access time. This is followed by a bit larger and slower in access L2 cache, then a bit larger and slower in access L3 cache and eventually the main memory (RAM) which is the costliest to access. The pyramid is shown in Fig.1.

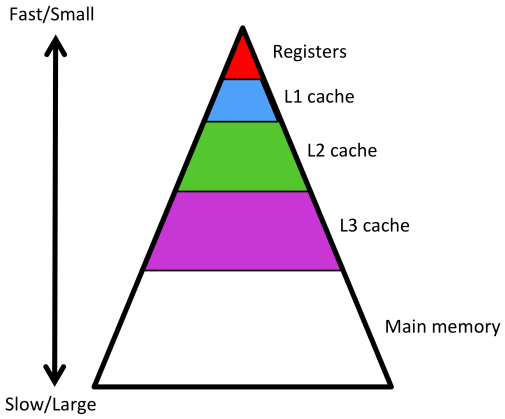

Figure 1: Memory Hierarchy Pyramid

To support this structure, the cache design uses inheritance object oriented paradigm such that we treat all 4 of these to be a type of `Memory`. Then to be specific, we have `Cache` and `RAM` classes that inherit the properties of `Memory` and have their own features on top. We also make sure that these memories follow the order in the pyramid. The `CacheSimulator` is the where the driver `main()` is written and it uses the `struct CacheInfo` to fill in the config parameters read from input. It then calls `CacheContoller` which creates and accesses `Memories` of given configs. The cache is implemented to contain `CacheSets` for set associative purposes. Each CacheSet then stores a doubly linked list of `CacheBlocks`. This high level design is shown in Fig.2.

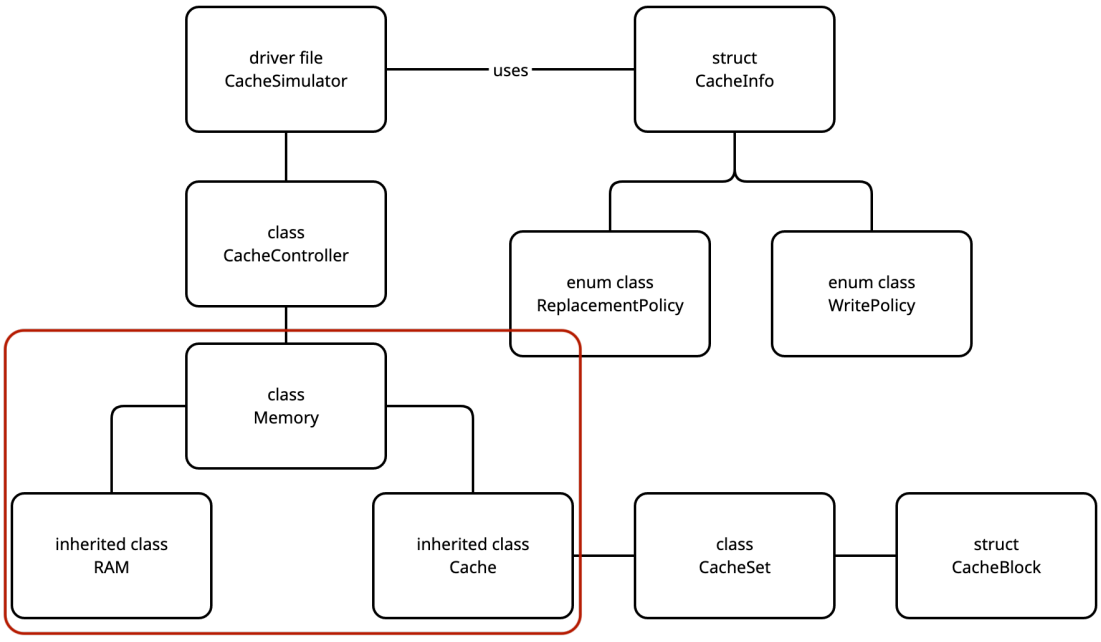

Figure 2: High Level Blocks in Cache Design

Inside the Cache:

An n-way set associative cache supporting i indexes should contain n CacheBlocks per set or index. We store these CacheBlocks per index in a structure called CacheSet. Thus, a CacheSet is a **linked list** of CacheBlocks. Linked list data structure was chosen here because in a least recently used scheme, we need a linked structure and an access to the head and the tail of the list for effectively maintaining the list as per the access order. Besides, insertions and removals are more efficient in a linked list than in a vector. When we try to access a block from a CacheSet, we first need to check if the block was already present in the set. If it was already present, we spot that block, remove it from that location and insert it at the head pointer. However, if this block wasn't present, we need to check if we have an empty spot (a block with false value of isValid) in the set to add the newly accessed block. If there was no empty spot, we track the replacement policy to take eviction decision. This structure overall makes sure that the most recently used block is always kept at the head/front and the least recently used at the tail/back. For a random replacement policy however, we generate a random index at the time of eviction and replace the block at that index with the new block.

Thus, for a cache, we have a **vector** of CacheSets which stores i CacheSets, one corresponding to each index. Vector is a good choice here since it has contiguous memory which makes random access easy, it has a default size (i indexes), and we do not need to perform any deletion operations on CacheSet objects.

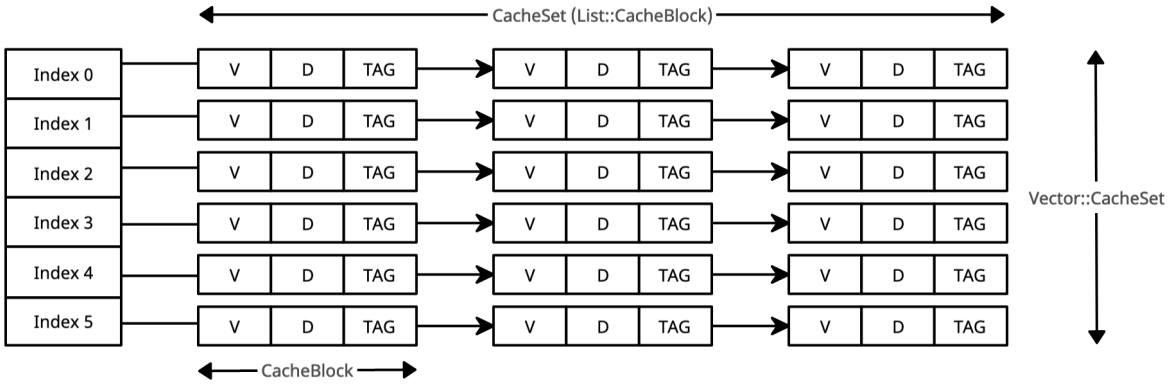

Figure 3: Example of a 3-way set associative cache

Now that we have the cache structure established, the instruction(Load, Store, or Modify) executions and the tracking of hits, misses, evictions, and cycles utilized can be discussed.

In addition to the data structures for cache, we also maintain variables like number of hits, number of misses, number of evictions, number of cycles utilized per operation and per execution to keep track of the results.

An important thing in the entire cache structure is that when we have a multi-layered cache, we need to not only create these L1, L2, or L3 caches but link them properly one after the other and the last cache then points to the main memory or RAM. This is where inheritance comes in pictures. When we execute a Load, Store, or Modify command, we call the access method on L1 cache in the beginning. Depending on whether or not the block at address was found in that cache, we delegate the access to the next layer. Thus our access to address bytes penetrate all the way down to the RAM in the order $L1 \rightarrow L2 \rightarrow L3 \rightarrow RAM$.

An instruction won't always try to access number of bytes equal to block size and it can access more number of bytes. In any case, the cache's access method makes sure to divide these bytes into chunks of block size for that cache and the look for that block in the CacheSets. If the block was found, we increment the number of hits and if it wasn't found, we increment the number of misses. In case of miss however, we go ahead and call the access method on the next memory layer to find the required block. Whenever a cache performs an eviction, we increment the number of evictions for that cache.

In the similar manner, we also track the number of cycles utilized in the access to each block of memory by each memory layer which was hit whilst trying to fetch that block. For a load operation, `isWrite = false`, for a store operation, `isWrite = true`, and Modify is Load followed by Store. We keep variable called `currentCycles` on each memory object to keep track of cycles it took for the entire operation. For each block accessed from a cache, we factor in the `cacheAccessCycles` of that cache. For a write miss, we factor in an extra penalty by adding `cacheAccessCycles` again. For RAM access however, we factor in `memoryAccessCycles` for the first 8 bytes and

then for each contiguous unit of 8-bytes accessed after that, we add 1 additional clock cycle to the `currentCycles` for RAM. In the end, we just loop through all caches to collect the cumulative number of cycles utilized by all the memory accesses during the entire operation.

Fig. 4 shows the overall sequence diagram of the cache simulation.

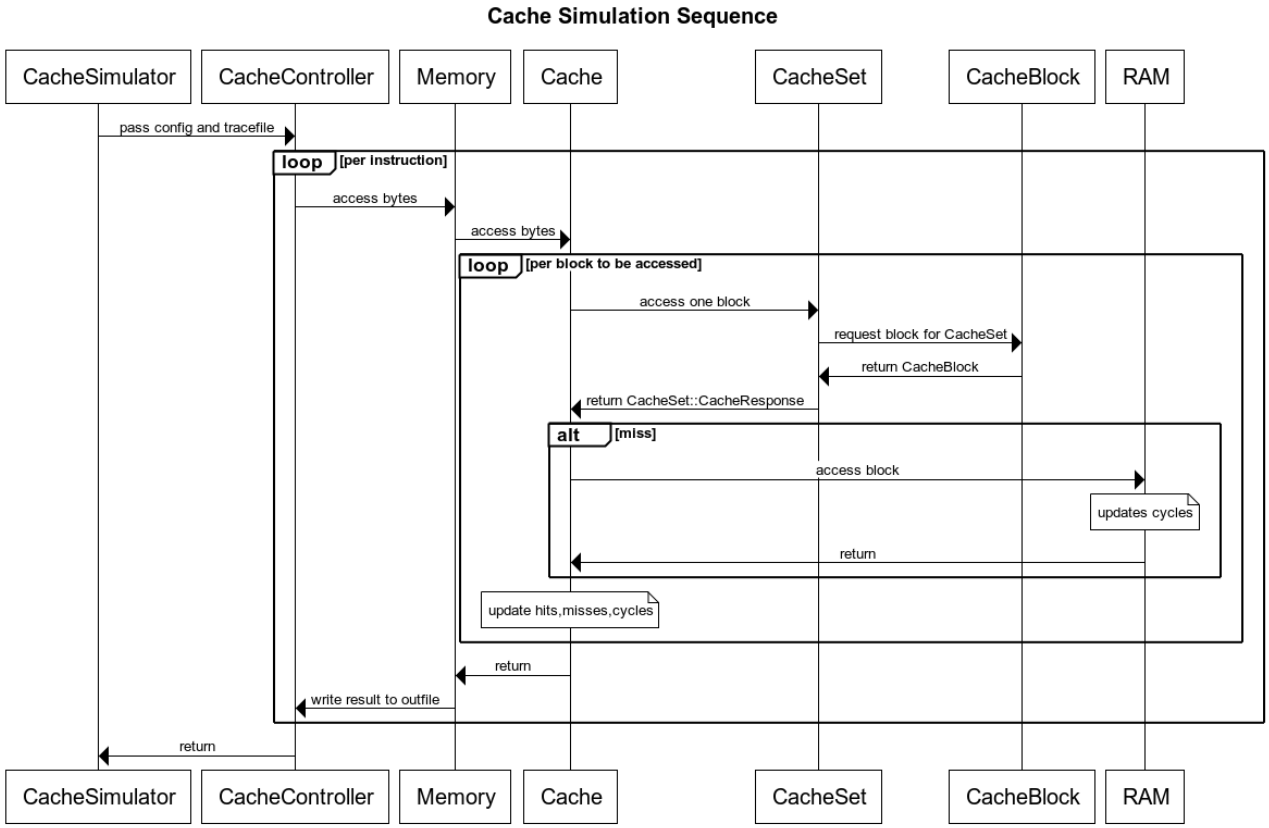

Figure 4: Sequence Diagram

All in all, this project was a wonderful experience that gave me thorough insights to the way things are implemented at the hardware level. It took me days to read and understand the project prompt; I had to re-read many parts multiple times. After understanding, I wanted to code in C (if not Java) as I had never worked with C++ before. My initial idea was to just write if else statements to serve the blocks from a series of caches. However, there were too many parameters and cases to handle. As instructed by Prof. Goins, it was indeed a wise decision to follow object oriented design. With this implementation, I can even scale the solution to even more number of caches and each cache handles itself just right. I have spent one whole week in this project and the segmentation fault was a nightmare. I would have taken even longer to finish had I not taken help from Prof. Goins' office hours and Ryan's guidance in understanding C++ syntax better (consts, pointers, virtual methods, built-in list are all weird). If I were to redo the project, I would like the start from scratch and without any stub code and once I have a working cache setup, I would try to optimize my O(n) searches and try to use more efficient data structures like HashMaps. Once I have that implemented, I'd like to support more Replacement Policies and then go on to the implementation of Parallel Caches maybe.

Fig. 5 shows the class diagram after implementation of the entire code.

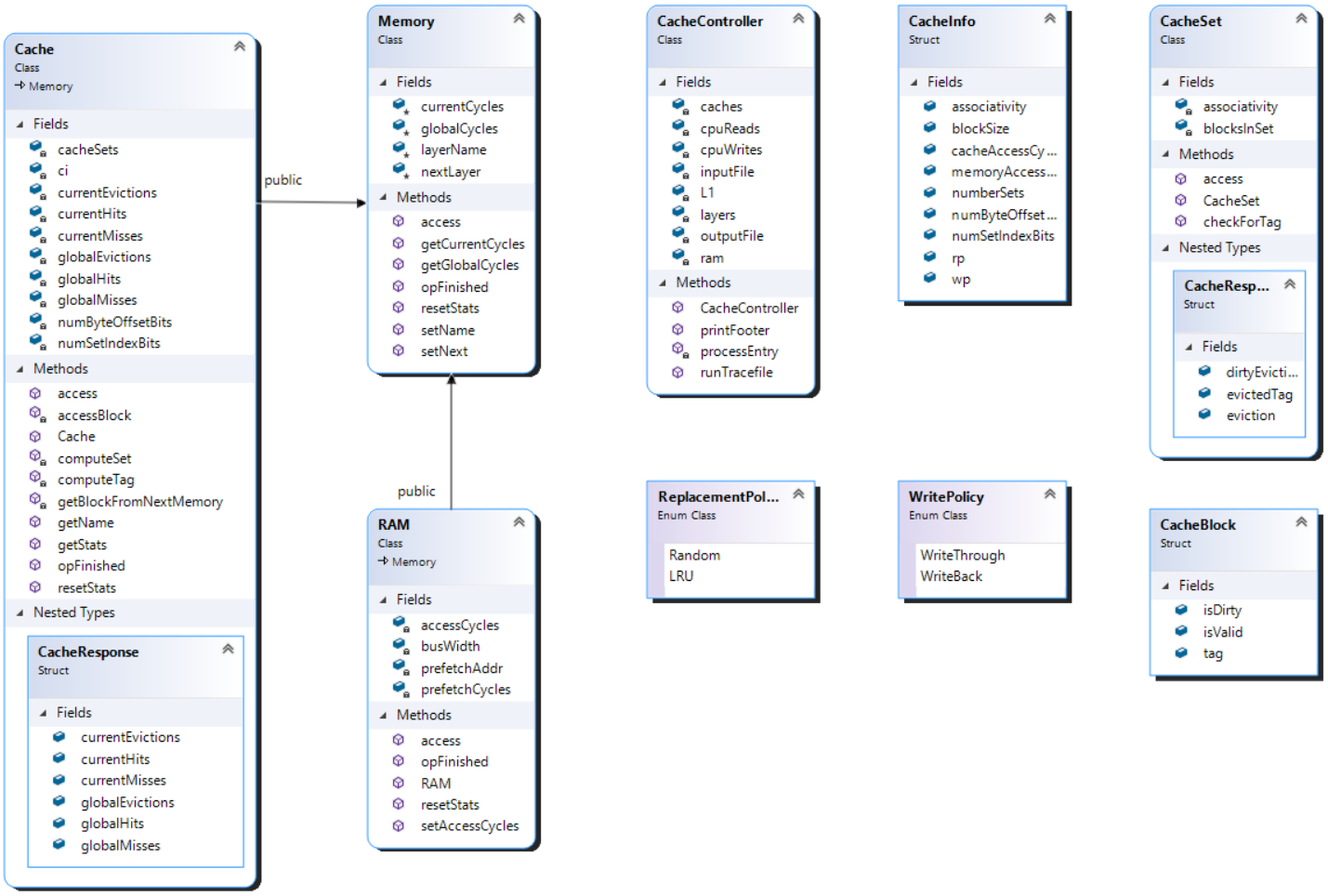

Figure 5: Class Diagram

EXPERIMENTATION SETUP

All the experimentation was done by simulating the CPU configurations of VM-Maple (an engineering research server at Oregon State). Fig. 6 shows the L1, L2, L3 capacities of this Intel(R) Xeon(R) CPU.

```
vm-maple ~ 1002$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                30
On-line CPU(s) list:   0-29
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):             30
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Model name:            Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz
Stepping:              2
CPU MHz:               2299.998
BogoMIPS:              4599.99
Hypervisor vendor:     VMware
Virtualization type:   full
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              30720K
NUMA node0 CPU(s):     0-14
NUMA node1 CPU(s):     15-29
```

Figure 6: CPU configurations

The cache associativities and block sizes for the CPU are as follows (Fig. 7):

```
vm-maple ~ 1005$ getconf -a | grep CACHE
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                8
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 32768
LEVEL1_DCACHE_ASSOC                8
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  262144
LEVEL2_CACHE_ASSOC                 8
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  31457280
LEVEL3_CACHE_ASSOC                 20
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

Figure 7: L1, L2, L3 Cache Specifications

Thus, the configuration parameters used are:

memoryAccessCycles	240
L1 numberSets	64
L1 blockSize	64
L1 associativity	8
L1 ReplacementPolicy	1 (LRU)
L1 WritePolicy	0 (WT)
L1 cacheAccessCycles	15

L2 numberSets	512
L2 blockSize	64
L2 associativity	8
L2 ReplacementPolicy	1 (LRU)
L2 WritePolicy	0 (WT)
L2 cacheAccessCycles	25

L3 numberSets	24576
L3 blockSize	64
L3 associativity	20
L3 ReplacementPolicy	1 (LRU)
L3 WritePolicy	0 (WT)
L3 cacheAccessCycles	120

PERFORMANCE ANALYSIS

The equations used to calculate the metrics miss rate and AMAT:

$$Miss\ Rate = TotalMisses\ /\ (Total\ Hits + Total\ Misses) * 100\% \tag{0.1}$$

$$AMAT = number\ of\ cycles\ /\ (CPU\ Reads + CPU\ Writes)\ cycles \tag{0.2}$$

Following (Fig. 8) is the table that denotes the miss rates (in percentage) and average memory access times AMAT (in cycles) for the provided three tracefiles by running experiments with 1 cache, 2 cache, and 3 cache configurations.

gcctracefile_1_core0	L1 Miss Rate	L2 Miss Rate	L3 Miss Rate	AMAT
Using L1 Cache	2.19%	NA	NA	81.0689
Using L1 and L2 Caches	2.19%	5.27%	NA	86.5776
Using L1, L2, and L3 Caches	2.19%	5.27%	5.38%	119.8686

netpathtracefile_1_core0	L1 Miss Rate	L2 Miss Rate	L3 Miss Rate	AMAT
Using L1 Cache	2.62%	NA	NA	103.5027
Using L1 and L2 Caches	2.62%	3.19%	NA	109.8263
Using L1, L2, and L3 Caches	2.62%	3.19%	3.14%	154.7178

openssltracefile_1_core0	L1 Miss Rate	L2 Miss Rate	L3 Miss Rate	AMAT
Using L1 Cache	0.0553%	NA	NA	43.095
Using L1 and L2 Caches	0.0553%	0.2522%	NA	45.9893
Using L1, L2, and L3 Caches	0.0553%	0.2522%	0.21%	5.4601

Figure 8: Miss Rates and AMATs for three files

The miss rates for *gcctracefile_1_core0* and *netpathtracefile_1_core0* follow a trend that L1 cache's miss rate is lower than L2 Cache's miss rate and L2 Cache's miss rate is lower than L3 Cache's miss rate. This might mean that these files have instructions that access addresses of a wide range such that the access mostly has to go all the way down to L3 Cache. However, *openssltracefile_1_core0* tends to exhibit a different behavior after using same configurations. For this file, L1 Cache's miss rate is lower than L2 Cache's miss rate but L2 Cache's miss rate is higher than L3 Cache. This odd behavior maybe due to the fact that the instructions in *openssltracefile_1_core0* have a good locality for most of the accesses.

Fig. 9 shows this miss rate trend in the form of a chart.

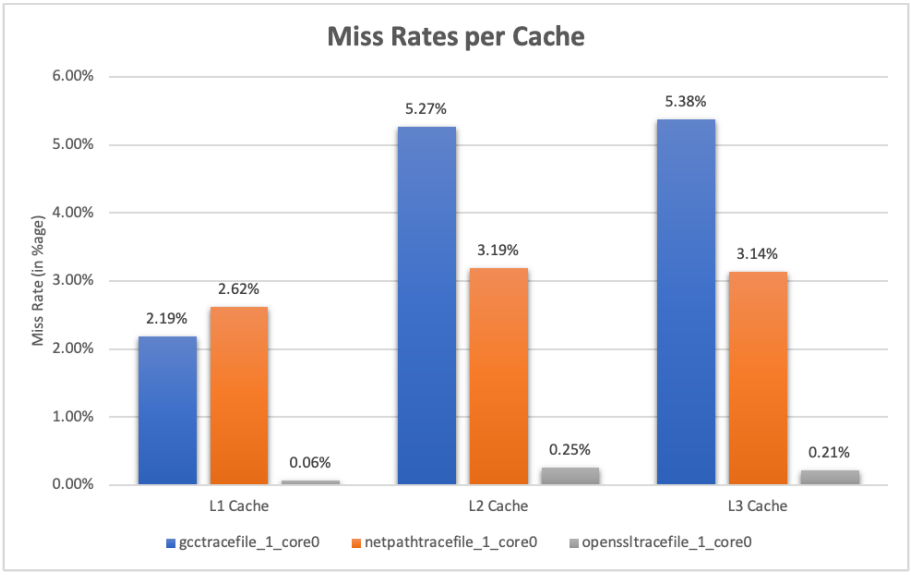

Figure 9: Miss Rate Analysis

The exact behavior is observed for AMATs as well and it can be seen in Fig. 10.

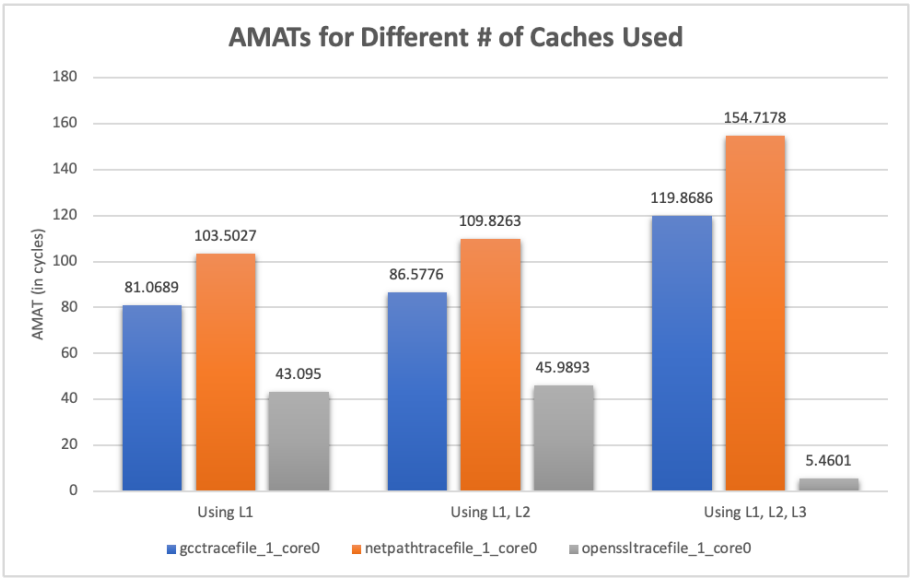

Figure 10: AMAT Analysis

SUPPORT TO WRITE-BACK SCHEME

Write-Back policy was implemented which doesn't write to the lower levels in the memory hierarchy until there's a need to evict the block. To support this scheme, we keep track of a dirty bit per CacheBlock. Whenever we write a new CacheBlock into the CacheSet of a layer, we write it with it's *isDirty* flag set to *true*. This copy of the block remains in the cache until the CacheSet gets full and there's a need to evict this block which had been set dirty. The access method in CacheSet keeps track of the dirty bit and notifies the cache of any dirty eviction that may have happened. If there was a dirty eviction, this block is written to the next memory layer and propagated up to the RAM.

gcctracefile_1_core0 was run using the same configurations as mentioned in the experimentation setup except that, now it's run in Write-Back mode. Fig. 11 shows the impact of using write-back mode in terms of total cycles:

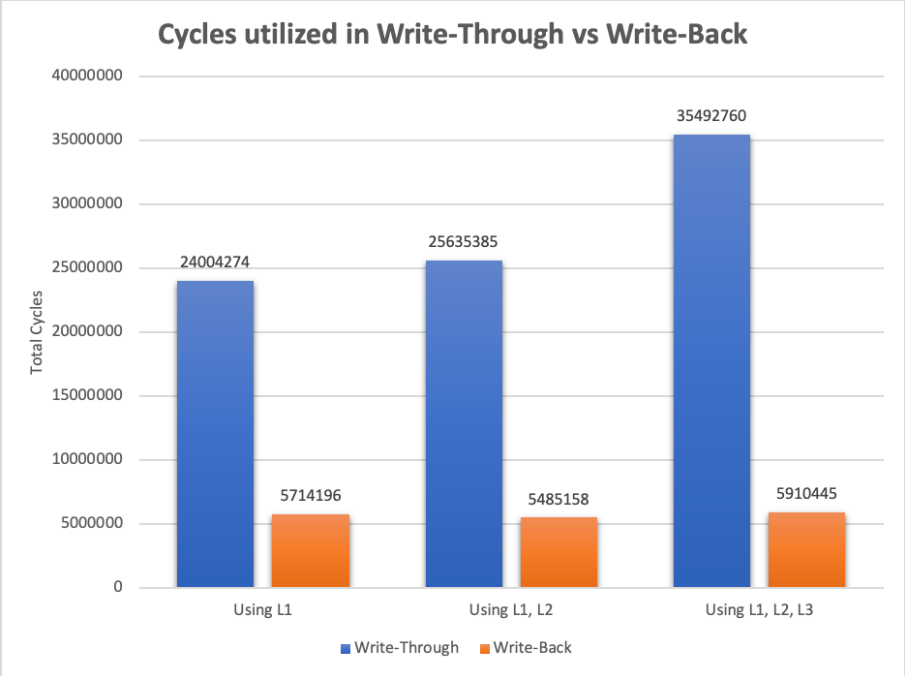

Figure 11: Write-Through vs Write-Back scheme

Write-back scheme is observed to utilize significantly less number of cycles. This is really good performance. In terms of hits, the L1 hits and misses are left unaffected because we always write the fresh block to the L1 cache. The L2 and L3 Cache hits reduce significantly whereas L2 and L3 misses may not reduce always. Overall, Write-back is a faster mode however, in case of a power failure, all the changes in the cache are lost.

All the code for this cache implementation can be found in the .tar file attached with the submission.