COMPARISON AND PERFORMANCE ANALYSIS OF CACHE COHERENCE PROTOCOL USING GEM5 SIMULATOR

Team #6 :

Manish Rudra Vijayakumar,

Madhan Vibeeshanan,

Keerthana Sharon Kannan

Abstract

Cache coherence is important as two or more cores sharing the same data must maintain the recent updated value to avoid reading old value. This maintains consistency between multi-cores in a shared memory. We have made a study of existing cache coherence methods, such as Snoopy coherence technique and Directory coherence technique. Snoopy coherence technique is studied with the help of MOESI coherence protocol and Directory coherence technique is observed with the help of MI, MESI TWO LEVEL, MESI THREE LEVEL, MOESI, and MOESI TOKEN coherence protocol. We have used GEM5 simulator and Spec 2017 benchmark to compare their performance. For simulation a precompiled program called MemTest, Ruby random tester, and Spec 2017 suite is used. We observed the performance through varying parameters like, cache size, block size and associativity.

Introduction

Computers are becoming a part of our lives. Everybody requires a fast computer. This was achieved by increasing the processor's frequency. Recent processors have high performance and high power. In terms of high performance, Increasing the number of cores plays an important role as well. Thus most of the computers are using multiprocessors with shared memory. Consistency and coherence are two essential features required for Shared memory multiprocessors. Employing this shared memory creates a problem of Cache coherence. Cache coherence is defined as an issue when two different cores can have two different values for the same location. This is possible when there is a separate cache memory for each processor. This happens when other processors do not get any update from a writing processor. This problem might lead to another category of cache miss called coherence miss along with compulsory, capacity and conflict misses.

Cache coherence is a discipline, which ensures that the changes in the values of shared data are propagated throughout the system in a timely fashion. When a processor is said to be cache coherent processor, it means it has the most recent value for an address. There are two requirements for cache coherence. 1) Write propagation – Changes to the data in any cache must be broadcasted to other copies of that cache line in the peer caches. 2) Transaction Serialization – Reads/writes to a single memory location must be seen by all processors in the same order.

The solutions for this problem is either software based or hardware based. The software-based solution reduces the hardware complexity of cache coherence using software-programming models by using either Flush or Invalidate cache blocks shared among processors. As this is complicated, the preferable method is a hardware-based solution. The basic idea is that processors should broadcast its operation to all other processors and other processors should change the state of shared block accordingly.

The hardware-based solution is in turn categorized into a directory based, snoopy based or hybrid methods. The directory-based method makes use of a directory to keep track of the status of each block. Request generated by processors for a block directs the request to the directory containing block. Then the directory either forwards the request or serves the request depending on the state of the block. In general, it is a point-to-point communication. Thus, it has a low broadcast requirement. One drawback of this method is that it experiences storage overhead and are prone to design bugs.

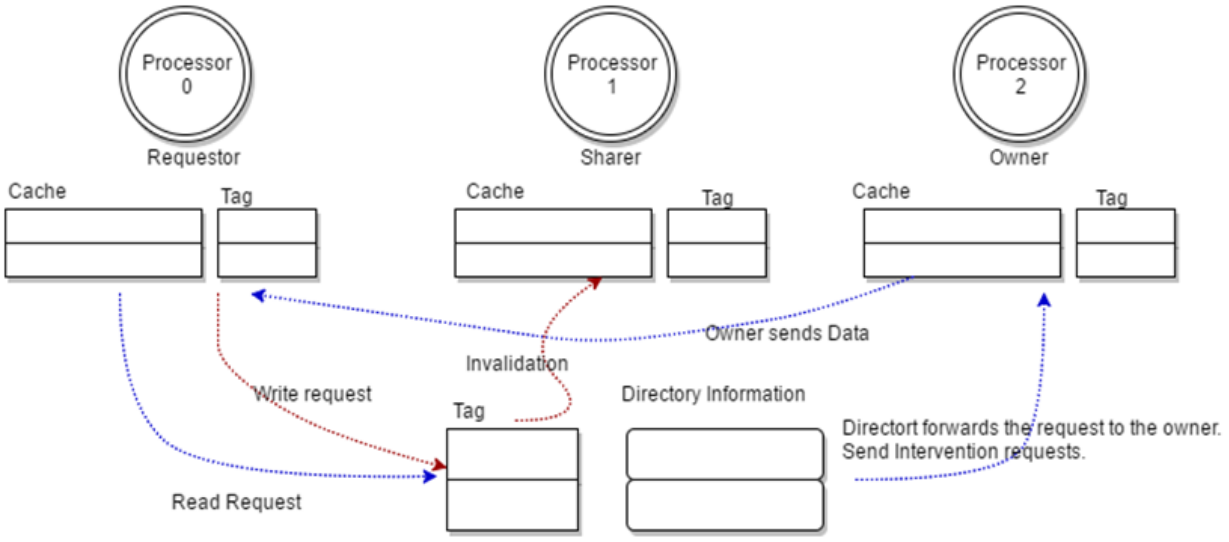


Fig 1: Directory based Coherence

The second method is the Snoopy based method. This depends on all the caches which observe the bus that connects processors to memory. Whenever a processor requests its cache for a certain value, the corresponding cache controller analyses the cache memory and makes decision accordingly. In this, bus-based communication is used to broadcast the messages to other connected processors. A cache controller that is snooping the bus will be concerned only if the

observed transaction involves a block of memory that the controller holds in its cache. Through bus activity, the request/response sent by other processors is observed. The drawback of this scheme is that it is not scalable and also it creates unnecessary traffic and consumes bandwidth.

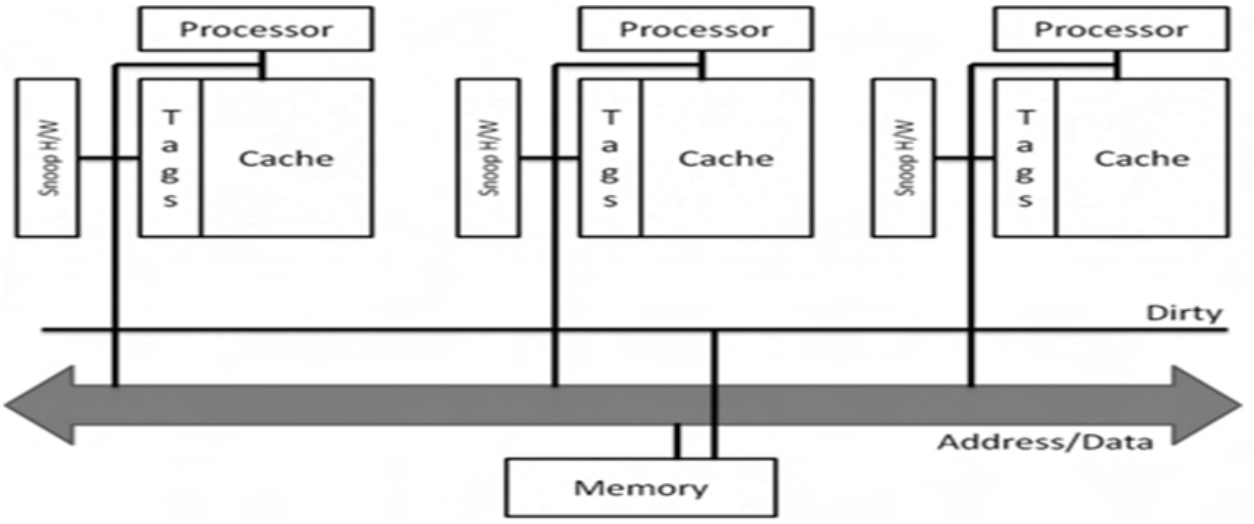


Fig 2:Snoopy based coherence

The third method is the hybrid approach, which dynamically adapts to snoopy or directory scheme based on interconnect network utilization. Either they broadcast (snoopy method) or unicast (which is the directory method) based on the availability of bandwidth.

Cache coherence Requirements: There are three main requirements for cache coherence. 1) Low Latency Cache to cache transfer 2)Avoiding bus-based architecture 3) Efficient Bandwidth. Low Latency Cache to cache transfer - Transfer of data through the cache to cache is much preferred instead of transferring it through main memory. This requirement is satisfied by Snoopy based coherence. In directory-based coherence, the request is directed to destination cache and waits for an acknowledgment which takes an extra clock cycle. Avoiding bus-based architecture - This requirement is satisfied by Directory-based coherence as Snoopy based coherence makes use of Bus architecture. For a directory based, it is a point to point communication. The reason why bus architecture is not preferable is because it restricts the integration of more cores on the system. Efficient Bandwidth - Bandwidth has to be reduced to avoid interconnection contention, as it affects system performance.

Write update - All shared caches are updated via bus snooping. Broadcasts writes throughout bus. The main drawback is if data is present in only one cache, broadcasting the message is useless. It creates large bus traffic. Write Invalidate - Node invalidates all other cached copies of shared data and can then update its own copy without further bus operation. This is much preferred as it uses less bandwidth.

Cache Coherence Protocols

Cache coherence protocols coordinate distributed caches, to provide a consistent view of memory to processors. The cache coherent processor has the most current value. The different protocols are:

MESI Cache Coherence Protocol

The MESI protocol works based on invalid-based cache coherence protocol mechanism.It supports write-back caches. Write back caches can save a lot on the bandwidth that is generally wasted on a write-through cache. There is always a dirty state present in write-back caches which indicates that the data in the cache is different from that in main memory. MESI is based on MSI protocol with the addition of one more state called Exclusive state. MESI Protocol requires the cache to cache transfer on a miss if the block resides in another cache. This protocol reduces the number of Main memory transactions with respect to the Modified Shared Invalid protocol. A significant performance improvement is noted. The protocol has four stable states, **M**, **E**, **S** and **I**. A block in **M** state means the blocks are writable (i.e. has exclusive permission) and has been dirtied (i.e. its the only valid copy on-chip). **E** state represents a cache block with exclusive permission (i.e. writable) but is not written yet. **S** state means the cache block is only readable and possible multiple copies of it exist in multiple private caches and as well as in the shared cache. **I** means that the cache block is invalid. One of the primary optimizations in this protocol is that if L1 Cache requests a data block even for read permission, if the L2 cache controller finds that no other core has the block,then it returns the cache block with exclusive permission. This was created in anticipation that cache blocks read would be written by the same core. This is exactly why **E** state exits. The cache blocks which have not been written to and has permission to be read only can drop the cache block from the private L1 cache without informing the L2 cache. This optimization helps to reduce write-back traffic to the L2 cache controller.

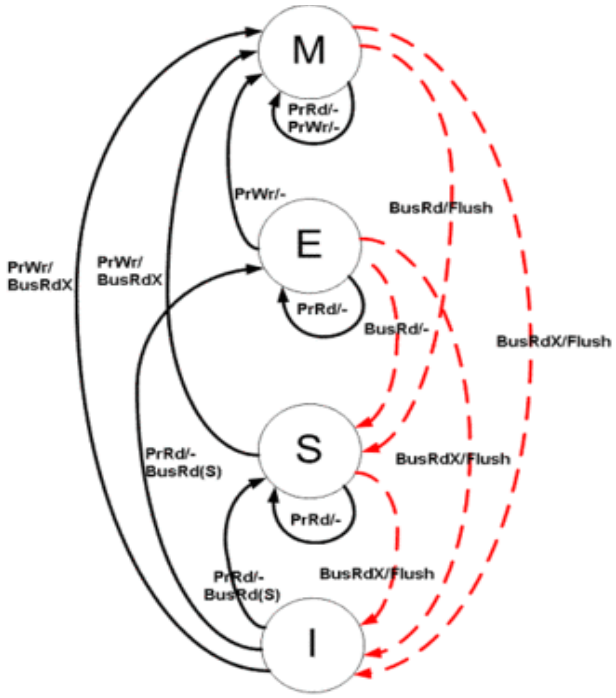
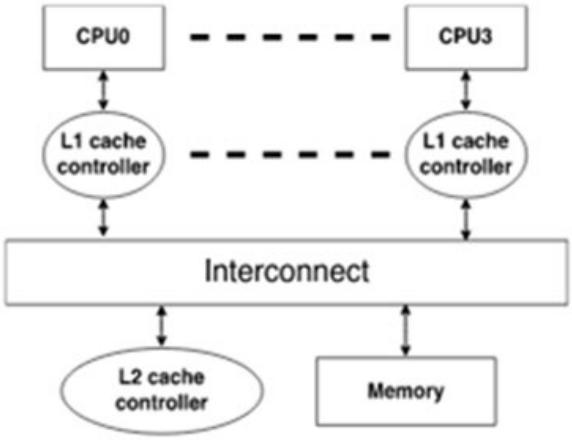


Fig 3:State transition diagram for MESI protocol

MESI Two Level

Multiple levels of caches are included in cache hierarchies to reduce the cache miss rate and increases the cache hit rate. In MESI Two Level cache hierarchies, the Level 1 cache L1 is designed to be the private cache of the processor and Level 2 cache is designed to be shared with all the processors. Each cache has its own controllers like L1 cache controller or L2 cache controller which connect to the same network. Two levels of cache hierarchies in the MESI Two Level are inclusive which means L0 and L1 are a superset.



(a) Memory hierarchy

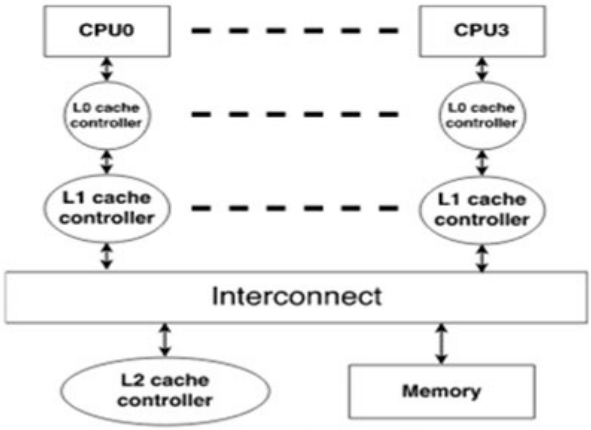
Parameter	Value
Number of directories	1
L1 instruction cache size	64kB
L1 instruction cache associativity	2
L1 data cache size	64kB
L1 data cache associativity	2
L2 data cache size	2MB
L2 data cache associativity	8
Cache line size	64B

(b) Default parameters

Fig 4:MESI Two-Level

MESI Three Level

MESI thee level cache hierarchies extends the previous two-level hierarchy with the third level of cache which consists of three caches where smaller caches L0, L1 are attached to the private caches and L2 cache is shared among all the caches. Each level of cache have their own cache controllers.



(a) Memory hierarchy

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

(b) Default parameters

Fig 5: MESI Three Level

MOESI Cache Coherence Protocol

This is an extension of the MESI protocol, which avoids the need to write a dirty cache line back to main memory when another processor tries to read it. MOESI is a full cache coherence protocol. It has states like Modified, Shared, Exclusive, Invalid which are commonly used in other protocols. Along with that there is a fifth state called "Owned" state which represents the data that is both modified and shared. The Owned state allows a processor to supply the modified data directly to the other processor. This is beneficial when the communication latency and bandwidth between two CPUs are significantly better than to main memory. The MOESI protocol can quickly share dirty cache lines from the cache, it cannot quickly share clean lines from the cache. If a cache line is clean with respect to memory and in the shared state, then any snoop request to that cache line will be filled from memory, rather than a cache. If a processor wishes to write to an Owned cache line, it must notify the other processors that are sharing that cache line. Depending on the implementation it may simply tell them to invalidate their copies or it may tell them to update their copies with the new contents.

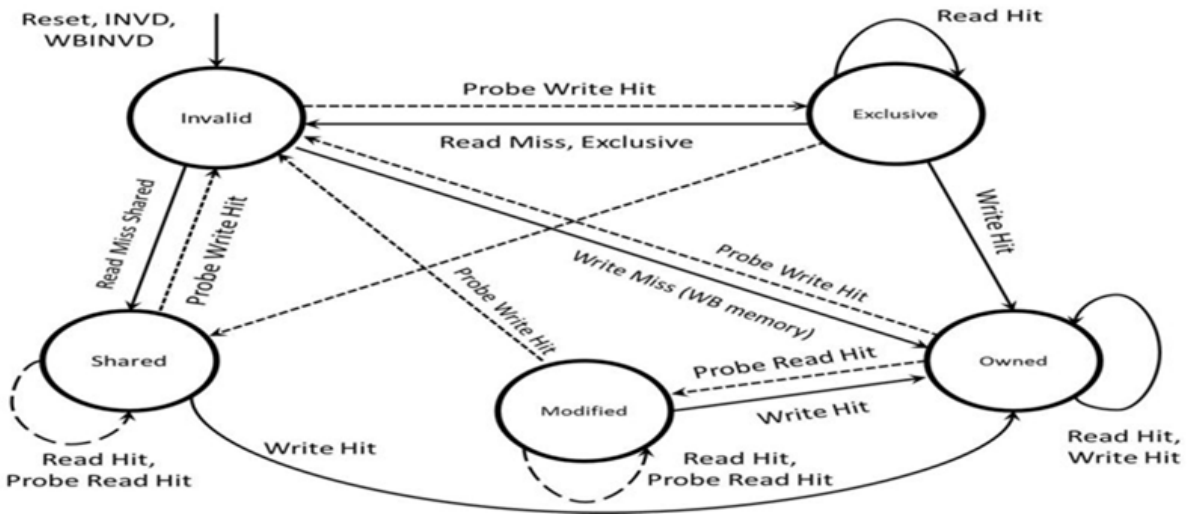


Fig 6:MOESI

MOESI_CMP_Directory

MOESI CMP Directory is the extended version of the MOESI cache coherence protocol. MOESI CMP Directory protocol has two levels of private caches which are non-inclusive. MOESI CMP Directory MOESI is a full cache coherence protocol that encloses all of the possible states like Modified, Shared, Exclusive, Invalid and Owned which are commonly used in other MOESI protocols. In addition to the fifth common MOESI protocol states, there are additional three states knows as MM, MM_W, and M_W states. MM state is similar to the conventional modified state,

MM_W is also similar to conventional modified state but replacements and DMA access are not allowed. This block automatically transitions to MM state after a timeout. M_W is similar to conventional E state which only allows loads and store operations. This blocks automatically transitions to the M state after a timeout. The state transition of the MOESI CMP Directory is given below.

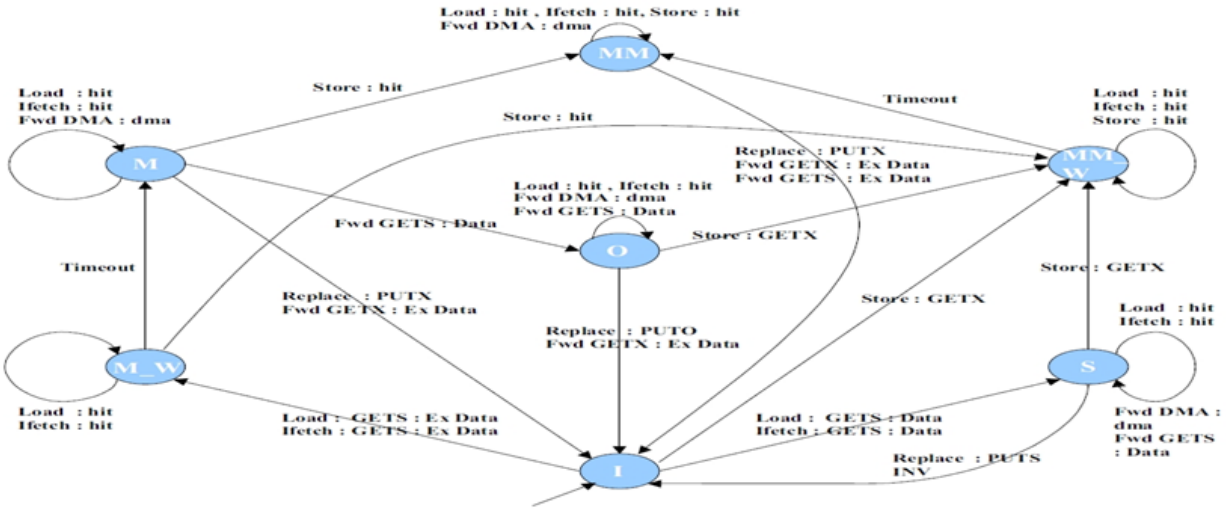


Fig 7:MOESI CMP Directory State Transition Diagram

MOESI_CMP_Token

MOESI_CMP_Token – 2 level caches hierarchy. This protocol maintains coherence permissions by explicitly exchanging and counting tokens. Each block has T token including one owner token. A processor can perform write operation on the block only, if it holds all the tokens and can perform read operation on the block only if it holds at least one token of the corresponding block and contains valid data. The coherent message will contain data only if hold owner token.

Benchmarks

For the Classic model, an existing program called MemTest was executed to do the analysis. For the Ruby model, SPEC 2017 was used. Under SPEC 2017, Leela, ImageMagick, LBM, Povray, Exchange2 were used.

Analysis and Results

Classic model: The following screenshot depicts the data through the bus for varying cache sizes and varying block sizes with the cache size being the same. As the cache size decreases,

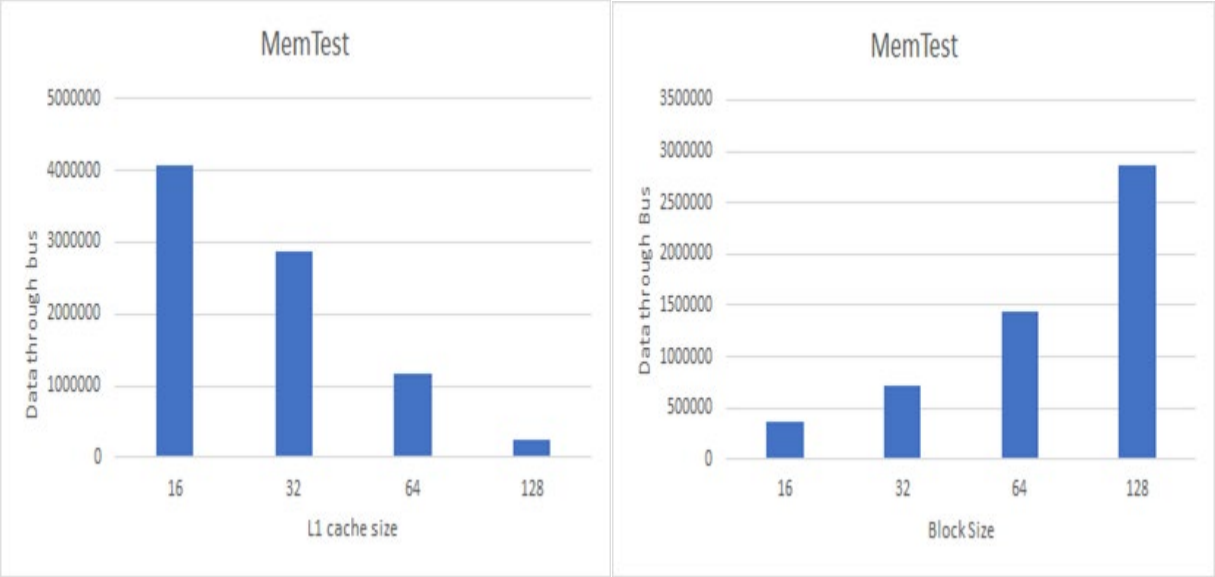


Fig 8: Classic Model with varying cache size and Block size

Ruby model:

The Default values that we used are as follows:

L1 Data cache Size	64kB
L1 Data Cache Associative	2
L1 Instruction cache size	32kB
L1 Instruction cache Associative	2
L2 Size	2MB
L2 Associative	8
Block Size	64MB
Memory Size	512MB
Maximum Workload	10000

Using these default values, the execution time is calculated with various benchmarks for the four protocols as shown in Fig 9. For Leela – MOESI_CMP_Token has a better execution time. For Imagick , Mesi-two level and MOESI_cmp_token has almost the same execution

time and is much better than others. For Povray and lbmcode – MOESI_CMP_token and for Exchange2 – MOESI_CMP_directory.

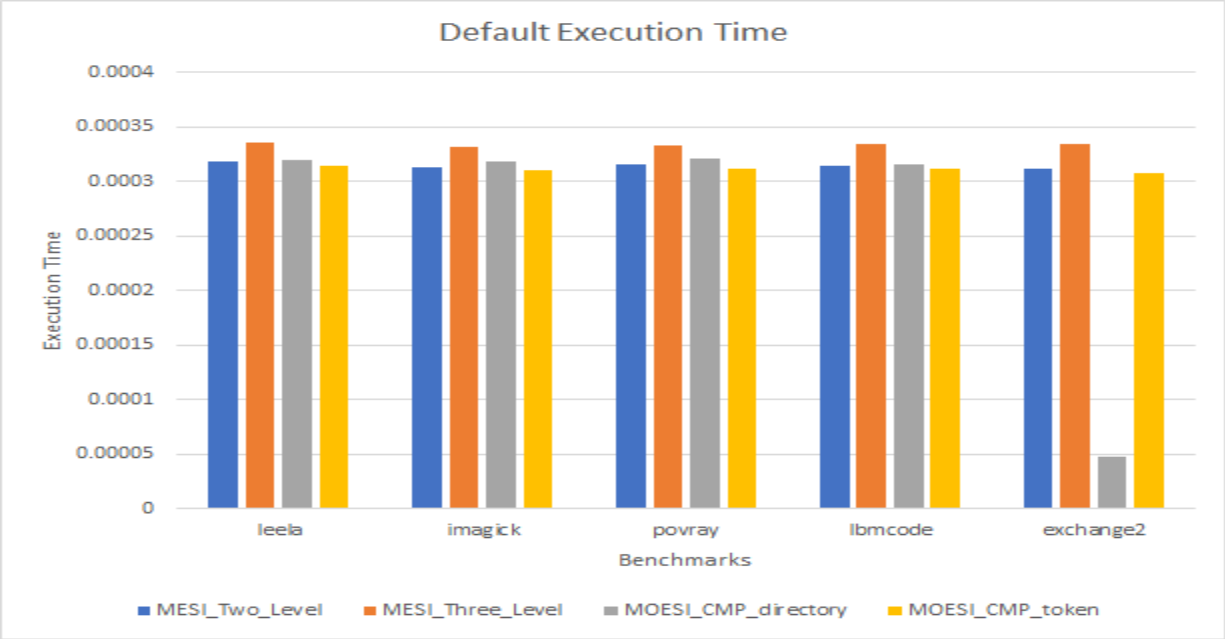


Fig 9: Default Execution Time

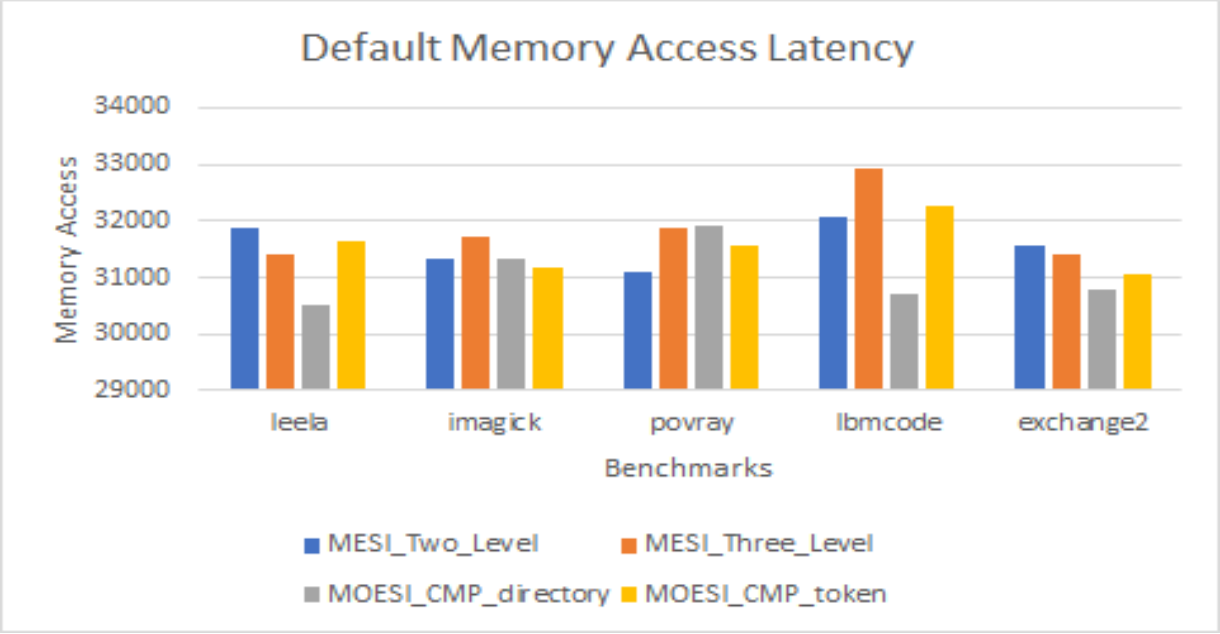
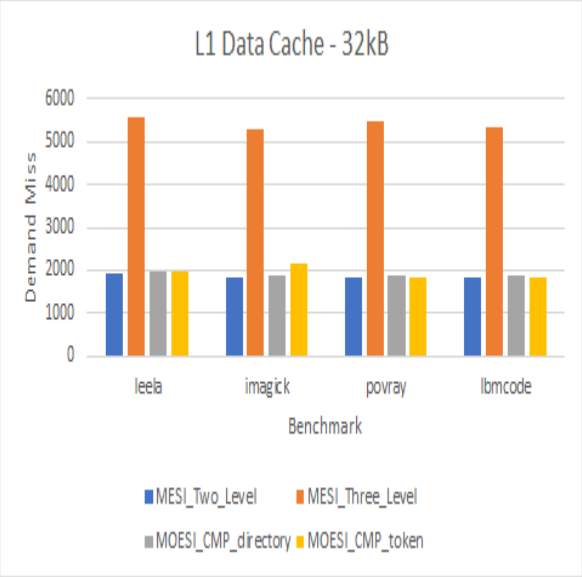
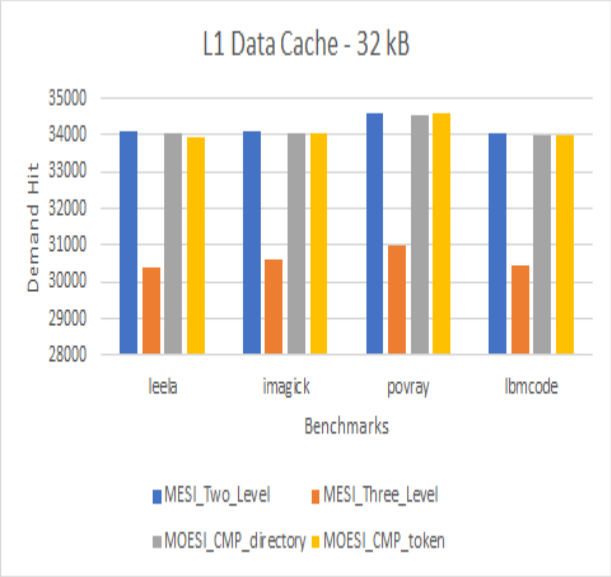


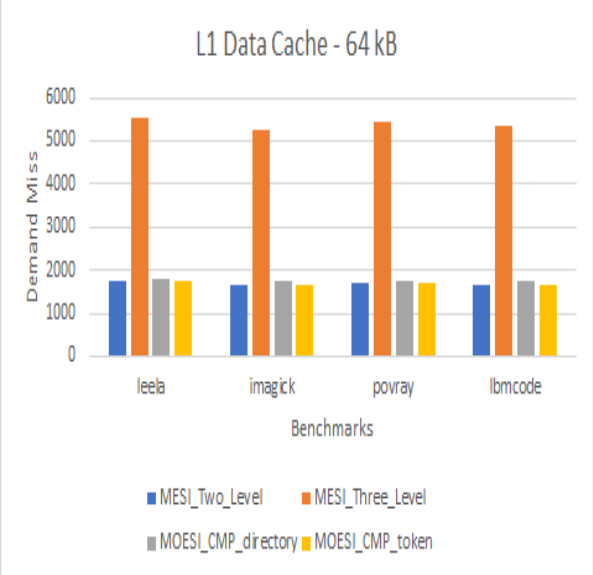
Fig 10: Default Memory Access Latency

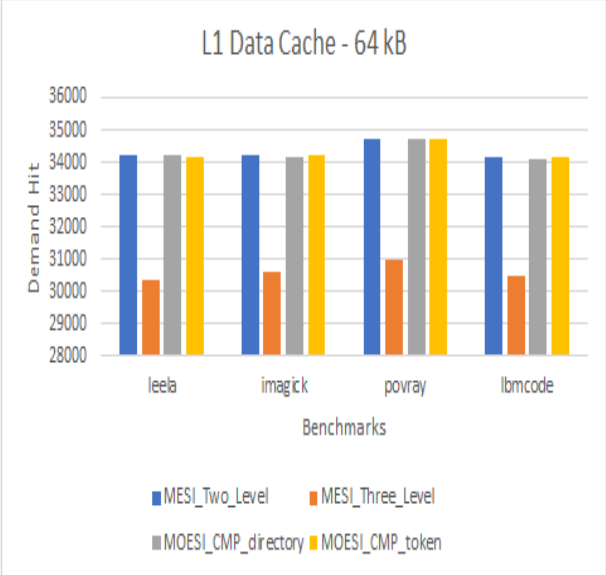
Fig:10 displays the memory access for various Benchmarks with different protocols. Here the MESI_three_level protocol had higher memory access. Each protocol had different values for every benchmark and the comparison to find the best protocol in any

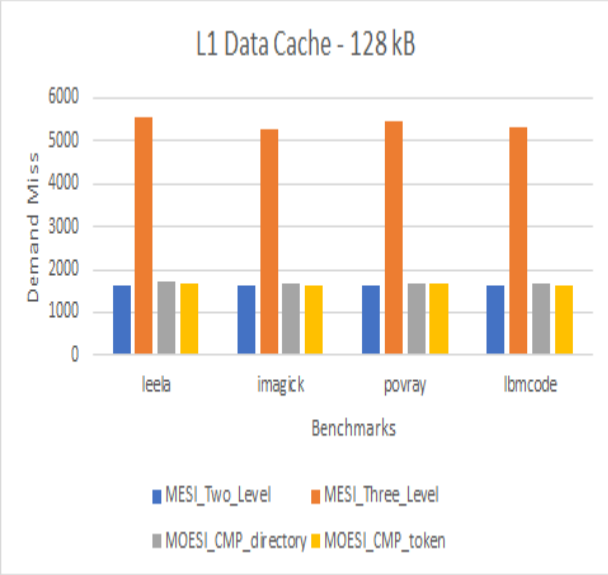
The following screenshots depict the Miss and Hit values for various protocols using different benchmarks by varying sizes. Surprisingly, MESI_three_level had a higher Demand Miss and lesser hit rate even though MESI_three_level has an extra level of cache than MESI_two_level cache. The reason is not known yet.

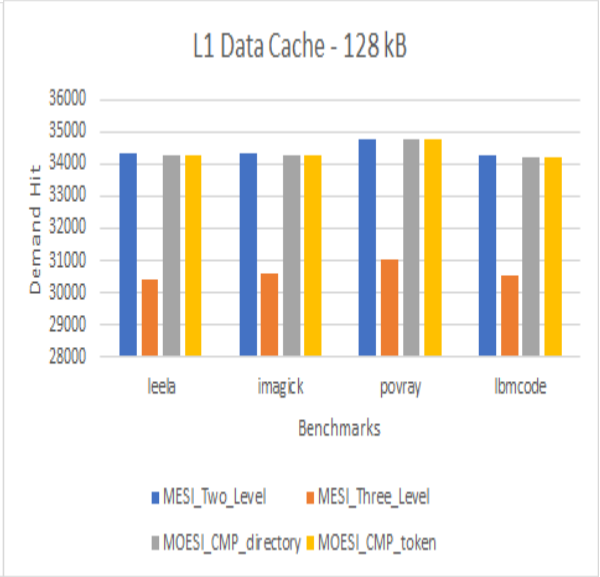


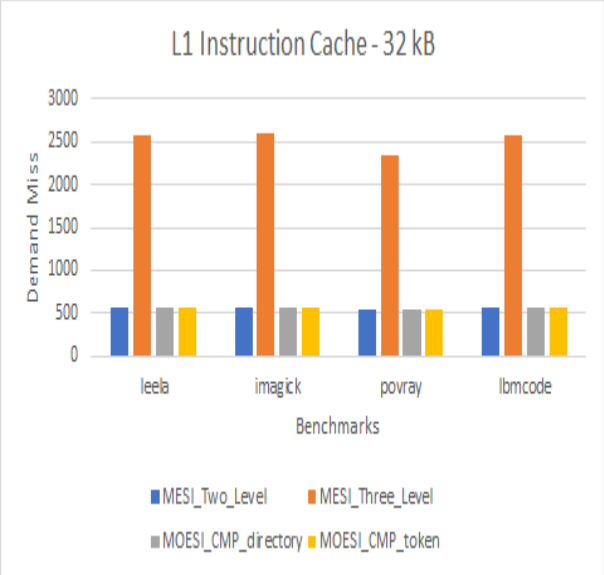


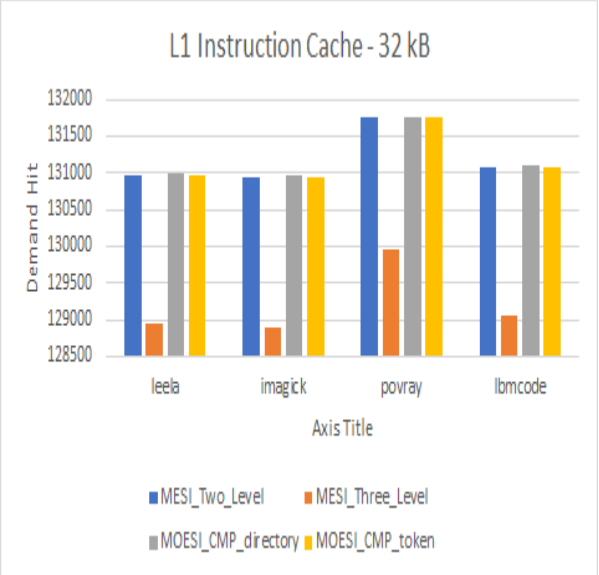


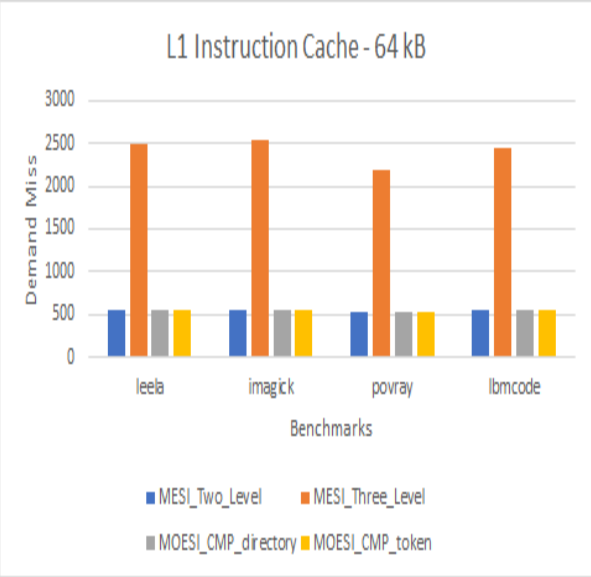


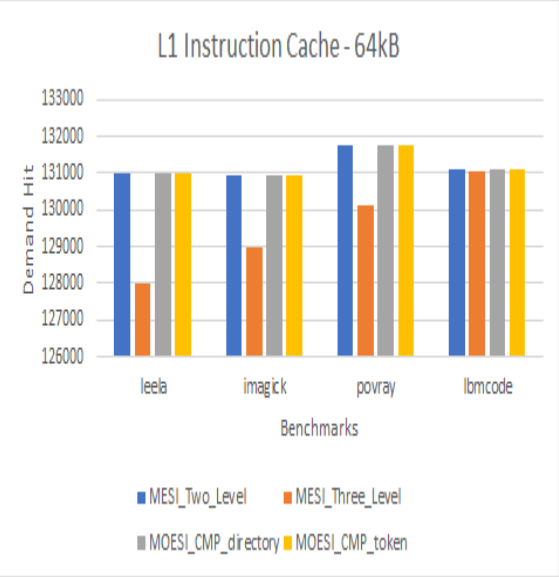

L1 Data Cache - 128 kB
Demand Miss
6000
5000
4000
3000
2000
1000
0
leela
imagick
povray
lbmcode
Benchmarks
MESI_Two_Level
MESI_Three_Level
MOESI_CMP_directory
MOESI_CMP_token

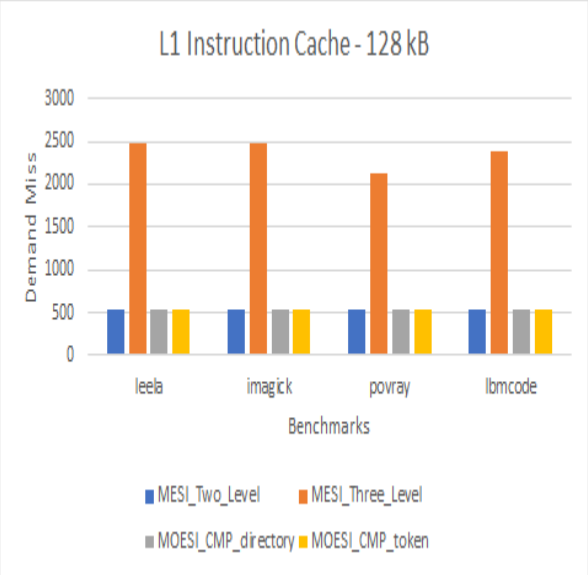

L1 Data Cache - 128 kB
Demand Hit
36000
35000
34000
33000
32000
31000
30000
29000
28000
leela
imagick
povray
lbmcode
Benchmarks
MESI_Two_Level
MESI_Three_Level
MOESI_CMP_directory
MOESI_CMP_token

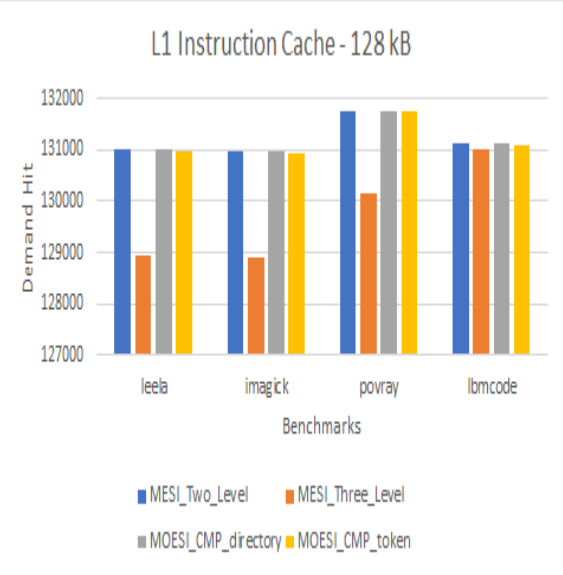

L1 Instruction Cache - 32 kB
Demand Miss
3000
2500
2000
1500
1000
500
0
leela
imagick
povray
lbmcode
Benchmarks
MESI_Two_Level
MESI_Three_Level
MOESI_CMP_directory
MOESI_CMP_token


L1 Instruction Cache - 32 kB
Demand Hit
132000
131500
131000
130500
130000
129500
129000
128500
leela
imagick
povray
lbmcode
Axis Title
MESI_Two_Level
MESI_Three_Level
MOESI_CMP_directory
MOESI_CMP_token

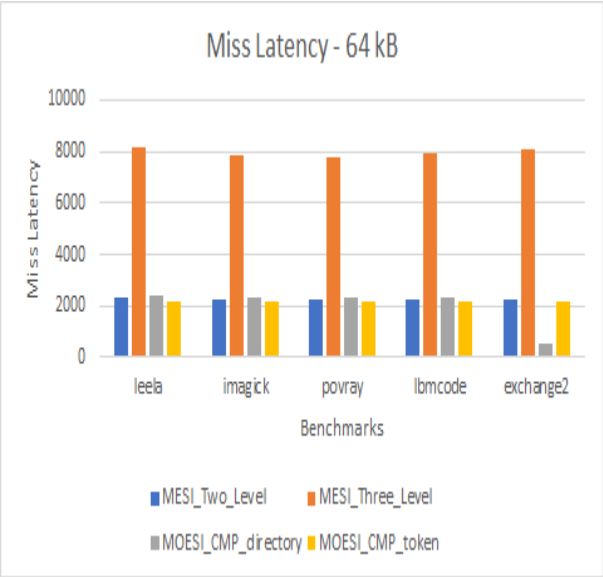


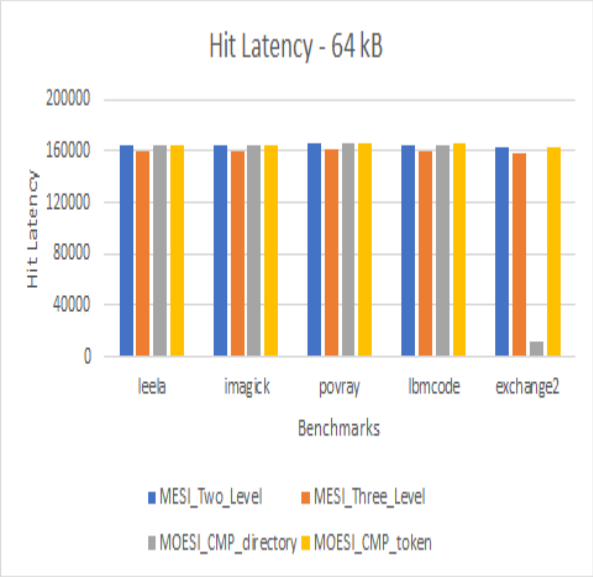


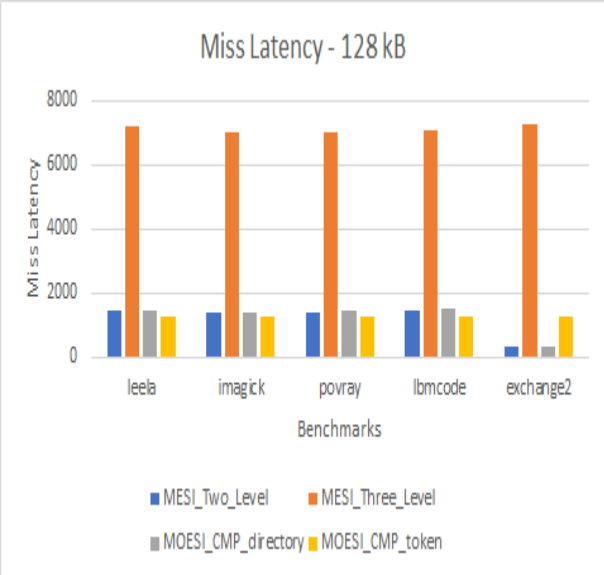


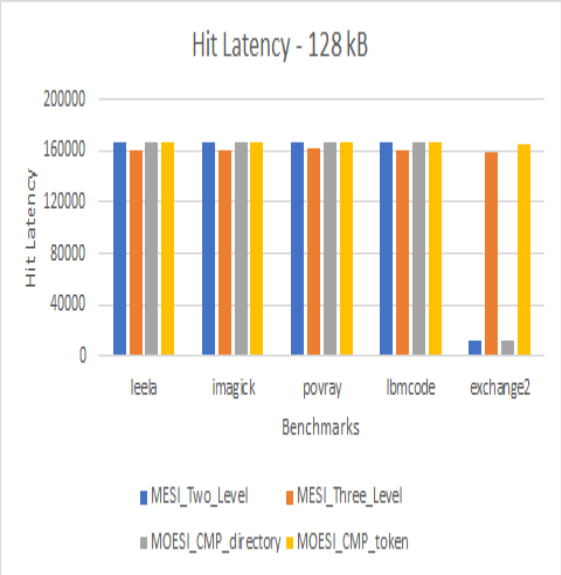


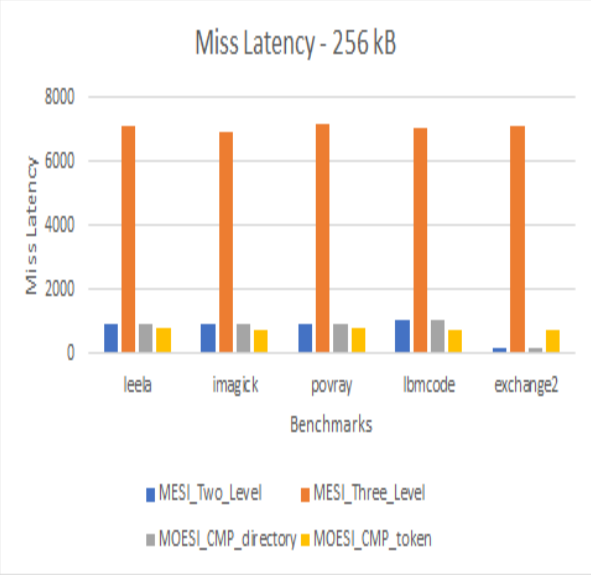
The following graph depict the miss latency and Hit latency with varying block sizes. We noticed the same pattern. MESI_TWO_LEVEL protocol had lower miss latency than MESI_THREE_level protocol despite having less level of caches.

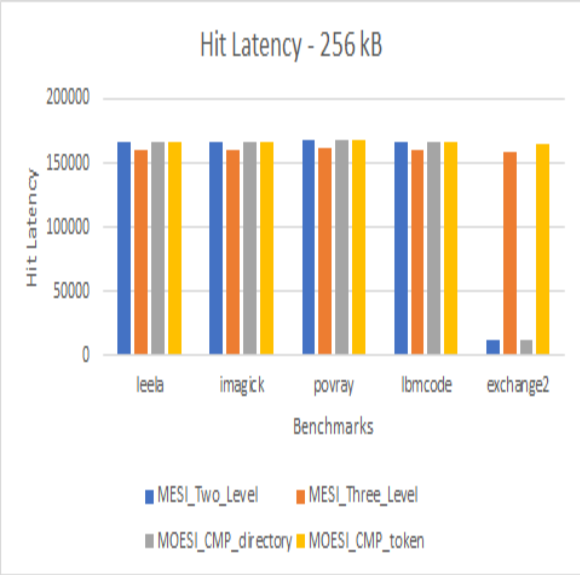

Miss Latency - 64 kB
Miss Latency
10000
8000
6000
4000
2000
0
leela
imagick
povray
lbmcode
exchange2
Benchmarks
MESI_Two_Level
MESI_Three_Level
MOESI_CMP_directory
MOESI_CMP_token


Hit Latency - 64 kB
Hit Latency
200000
160000
120000
80000
40000
0
leela
imagick
povray
lbmcode
exchange2
Benchmarks
MESI_Two_Level
MESI_Three_Level
MOESI_CMP_directory
MOESI_CMP_token


Miss Latency - 128 kB
Miss Latency
8000
6000
4000
2000
0
leela
imagick
povray
lbmcode
exchange2
Benchmarks
MESI_Two_Level
MESI_Three_Level
MOESI_CMP_directory
MOESI_CMP_token


Hit Latency - 128 kB
Hit Latency
200000
160000
120000
80000
40000
0
leela
imagick
povray
lbmcode
exchange2
Benchmarks
MESI_Two_Level
MESI_Three_Level
MOESI_CMP_directory
MOESI_CMP_token





Complications Faced

The initial implementation plan for this research project was to follow the base papers implementation to study and analyse the different Cache coherence protocols in the ruby model of the Gem5 simulator. But things did not work out as we planned. First we installed Ubuntu on a virtualbox and installed gem5 on it. But compiling the binaries for the gem5 took a lot of time as the VM uses only one CPU in order to compile the gem5 binaries and after trying to run a few simulations the ubuntu installed on the VM crashed as it did not have enough processing power to run gem5. Hence, we had to install Ubuntu again, but this time we did it on a dual boot along with windows. By using dual boot we were able to make use of all the cores of the computer and building the binaries were much faster. The reason we were concerned about the time taken for building the binaries is that, we had to build them each time if we want to run a different protocol. After successfully building the binaries we tested the MOESI protocol in the classic model using a python file (memtest.py) provided by the gem5 simulator. Now we had to test the the other cache coherence protocols in the ruby model. For testing the different cache coherence protocols, we wanted to use SPLASH-2 benchmark. We were not able to integrate SPLASH-2 with gem5. We tried it in both SE mode and FS mode and still we failed. We did get the binaries of the SPLASH-2 which was precompiled but it was compiled for ALPHA ISA, even that did not work. Then we moved on to try and use PARSEC benchmark suite. PARSEC can be used with Gem5 only in FS model. We wanted to try it out even if running gem5 in FS mode was a bit complicated. But we could not get a proper disk image to run PARSEC in FS on gem5. After searching through the internet, we were able to find a disk image that seemed to work, but PARSEC threw some error saying incompatible ISA. Finally, as our last resort, we planned on Integrating SPEC2017 with gem5 to test the protocols as SPEC2017 can be run on SE mode and the document page of spec2017 [6], question 6, specified that the spec2017 is now used to test caches. Therefore, we

went ahead to integrate SPEC2017 with gem5 and the integration worked. After the integration was successful, we tested out the various configurations to do the study and analysis.

Conclusion

In terms of performance, MOESI CMP Token cache coherence protocol is the most effective and MESI 3 level protocol is the least efficient. Performance of directory coherence is better than snooping coherence protocols because of the creation of bottlenecks.

References

[1]. N. B. Mallya, G. Patil, and B. Raveendran, "Simulation-based Performance Study of Cache Coherence Protocols," 2015 IEEE International Symposium on Nanoelectronic and Information Systems, Indore, 2015, pp. 125-130.doi: 10.1109/iNIS.2015.52.

[2]. J. K. Archibald, “The Cache Coherence Problem in Shared-Memory Multiprocessors”, Ph.D. Dissertation, Dept. Comput. Sci., Uni. Of Washington, 1987.

[3]. M.R. Marty, “Cache Coherence Techniques for Multicore Processors”, Ph.D. Dissertation, Dept. Comput. Sci., Uni. Of Wisconsin-Madison, 2008.

[4]. http://gem5.org/Cache_Coherence_Protocols

[5]. https://www.spec.org/cpu2017/

[6] https://www.spec.org/cpu2017/Docs/overview.html#benchmarks