Design and Implementation of a Simple Cache Simulator in Java to Investigate MESI and MOESI Coherency Protocols

Somdip Dey
School of Computer Science
The University of Manchester
Manchester, United Kingdom

Mamatha S. Nair
School of Computer Science
The University of Manchester
Manchester, United Kingdom

ABSTRACT
To improve the efficiency of a processor to work with data, cache memories are used to compensate the latency delay to access data from the main memory. But because of the installation of different caches in different processors in a shared memory architecture, makes it very difficult to maintain consistency between the cache memories of different processors. For that reason, having a cache coherency protocol is really essential in those kinds of system. There are different coherency protocols for caches to maintain consistency between different caches in a shared memory system. Few of the famous cache coherency protocols are MSI, MESI, MOSI, MOESI, MERSI, etc. In this paper, the primary focus were to study the working protocols of MESI (Modified-Exclusive-Shared-Invalid) and MOESI (Modified-Owned-Exclusive-Shared-Invalid) cache coherency protocols by designing a simple cache simulator in java, and publish the results and research findings. The main purpose of this paper is to provide new researchers and computer science students the idea regarding how to build and implement a simulator in order to understand the novel cache coherency protocols.

General Terms
Computer Simulation, Algorithms, Hardware, Applied Computing

Keywords
Computer Architecture, Cache Simulator, MESI, MOESI, Cache Coherency Protocol, java

1. INTRODUCTION
In modern processor architecture, because of the speed difference between the main memory and processor, it might take too many cycles for a processor to access the main memory. As a result this may cause hindrance in performance. So to deal with this problem, faster memories can be installed in the system in order to store the data from the main memory for frequent use by the processors. These fast memories, which are either on chip or off-chip to improve latency and performance, are called cache memories [1,2,5,7].
Because of high degree of locality in most programs i.e. if a processor reads or write a memory address (memory location), then there is a high probability that the processor might read or write the same location again very soon. Another feature is that if a processor read or write a memory location then there is a probability to read or write nearby locations also. To exploit the second behavior, caches may operate by holding a group of neighbouring data known as cache lines (also called cache blocks) [1,2,5].
Now, if a system has many processors and those different processors have different cache memories, and the data from the main memory is shared with different caches of different processors then it might give rise to high inconsistency if there is any change in the shared data even in one of the caches. But if the processors only read from the same memory address then there is no problem of inconsistency. For example if one processor tries to update the shared cache line then the copy of that cache line in other processors have to be invalidated in order to make sure that other processors do not read an out-of-date value of the modified cache line. This problem is called cache coherency.
To address the cache coherency problem, there are many protocols to deal with this. In this paper, the authors study the two mostly used cache coherency protocols [4, 5] i.e. MESI and MOESI protocols.
It should be always kept in mind that this research paper is actually meant for new researchers and computer science students, who have started to explore different features of computer architecture and designs, and those who lack the practicality of the implementation level of cache simulation in real practice. This paper is just a guiding stone to the real treasure, i.e. this paper only gives you the basic idea of how to design and implement a simple cache simulator to implement MESI/ MOESI or both protocols.
In the next section, we will be reviewing the basic working of MESI and MOESI protocols. After that we compare the MESI and MOESI protocols (in section 3) and review few of the recent researches in the field of cache coherency (section 4). In section 5 we will see different coherency mechanisms, by which coherency protocols are achieved, and then compare the two mostly used coherency mechanism i.e. snooping based coherency and directory based coherency. In section 7 we are going to review the results and outcomes of the cache simulation, and then discuss the limitations and conclude the project.

2. THE TWO MOSTLY USED CACHE COHERENCY PROTOCOLS: MESI AND MOESI
In this section, the basic working of the two most commonly used cache coherency protocols: MESI (Modified-Exclusive-Shared-Invalid) and MOESI (Modified-Exclusive-Shared-Invalid) are reviewed.

2.1 MESI Coherency Protocol
MESI [3] is one of the mostly used cache coherency protocol, which supports both write-back1 and write-through2 cache.

It is named after the possible cache line states. To deal with cache coherency problem, the cache lines are given a state based on certain characteristics and the four possible states in MESI are the following:

Modified: The cache line, which is present in the current cache, is dirty and the value has been modified from the value in the main memory.

Exclusive: The current cache line is only present in the current cache and no other copies of it are available in other caches, and the value is clean, i.e. the value is same as the value in the main memory.

Shared: The cache line is stored in other caches also and the value is clean, i.e. the value matches the value in the main memory.

Invalid: Indicates that the value of the cache line is not valid and back-dated, so it should not be used.

The state diagram for MESI is shown in figure 2 [2,3].

Operations in MESI Protocol:
The different states in the MESI protocol can be understood from the following example. Let us consider three processors P1, P2 and P3. P1 reads from address x from the main memory and then stores it in its cache in *exclusive* state (figure 1(a)). After that when P2 reads from the same address x and processor P1 detects the address conflict, and P1 respond to P2 with associated data. So the data from x is cached in both the processors P1 and P2 in *shared* state (figure 1(b)). Now, if processor P2 writes to the shared address x then the value of the cache changes to *modified* and it broadcasts a message to P1 to change the cache line state of P1 to *invalid* (figure 1(c)). So to get the updated value of x, P1 sends a request to P2, which has the updated value, and P2 shares the modified value with P1, where caches of both the processors (P1 & P2) again return to *shared* state (figure 1(d)).

If we consider that there are two pairs of caches then the permitted MESI states are shown in Table 1.

Table 1: MESI Allowed States

	M	E	S	I
M	✗	✗	✗	✓
E	✗	✗	✗	✓
S	✗	✗	✓	✓
I	✓	✓	✓	✓

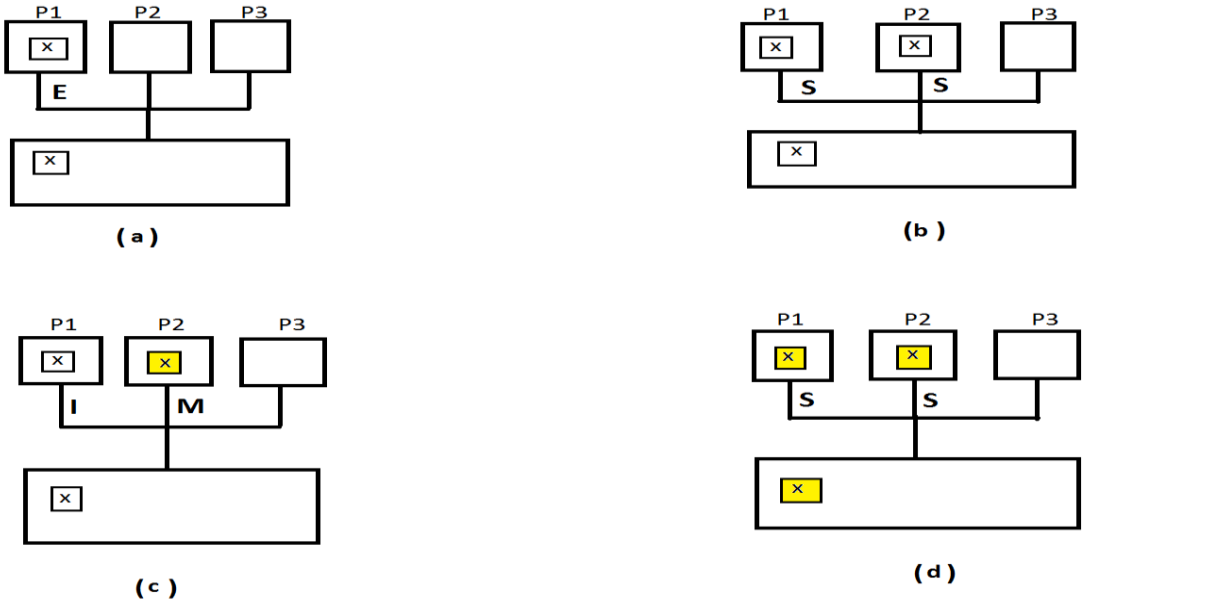

Figure1: (a) Exclusive State, (b) Shared State, (c) Modified and Invalid States, (d) Shared State after modification of the value [1]

[1]write-back: write operation is performed on the cache but the write operation in the main memory is postponed until the data on the cache line is about to be modified/replaced
[2]write-through: write operation is performed synchronously both to the cache and to the main memory

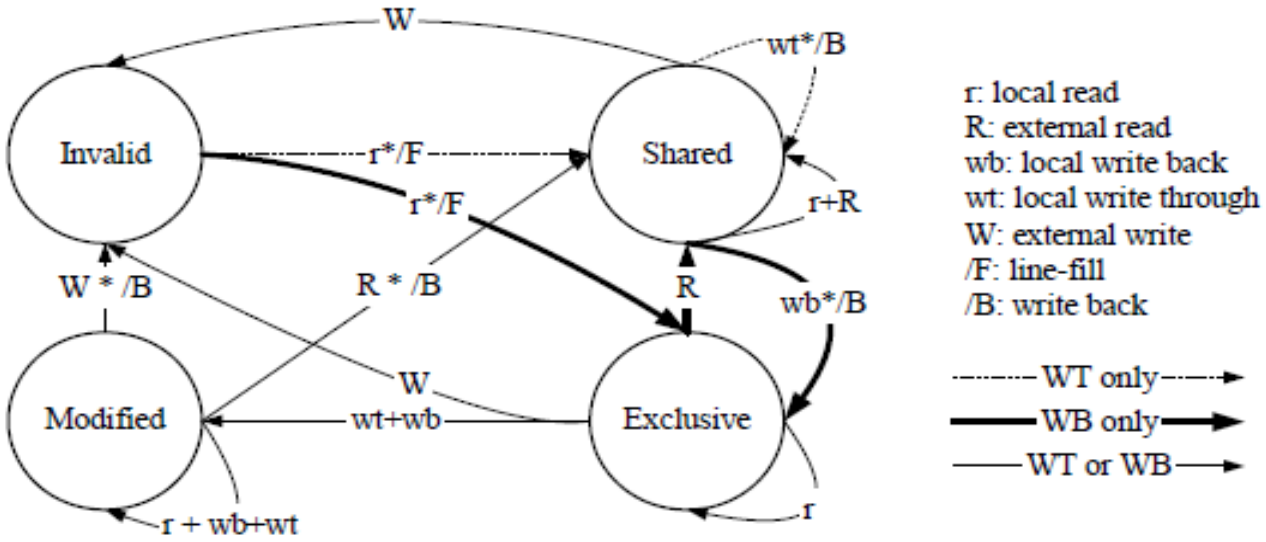

Figure 2: State Diagram for MESI Protocol [2,3]

2.2 MOESI Coherency Protocol

The MOESI coherency protocol has the same states as of MESI protocol except the fact that MOESI has one more state which is '*Owned*' state.

Owned: In this state, the cache line is available/shared in all other caches but the current cache only has exclusive right to make changes to it. So if any modification is done to the cache line then the state of the cache, where it is modified will be set to owned before sharing the modified cache line with other caches. Another benefit of this is that dirty values can be shared with other sharing caches without even updating the value in the main memory.

But the exception for MOESI protocol is that direct cache to cache transfer should be possible in the system.

The state diagram for MOESI is shown in figure 4 (source from North Carolina State University Wiki page).

Operations in MOESI Protocol:
Let us consider the same example of MESI protocol. All the states are same except in the part when the cache line is modified. So if processor P2 writes to the value shared with P1, then first the state of its own cache line will change to *modified* and then it will broadcast P1 to *invalidate* its cache line (figure 3(c)). After that P2 will share its dirty cache line value directly with P1 instead of writing it first to the main memory. While sharing the dirty value with P1, P2 will change its state to *owned* and then the state of the cache line of P1 will change to *shared* (figure 3(d)).

If we consider that there are two pairs of caches then the permitted MOESI states are shown in Table 2.

Table 2: MOESI Allowed States

	M	O	E	S	I
M	✗	✗	✗	✗	✓
O	✗	✗	✗	✓	✓
E	✗	✗	✗	✗	✓
S	✗	✓	✗	✓	✓
I	✓	✓	✓	✓	✓

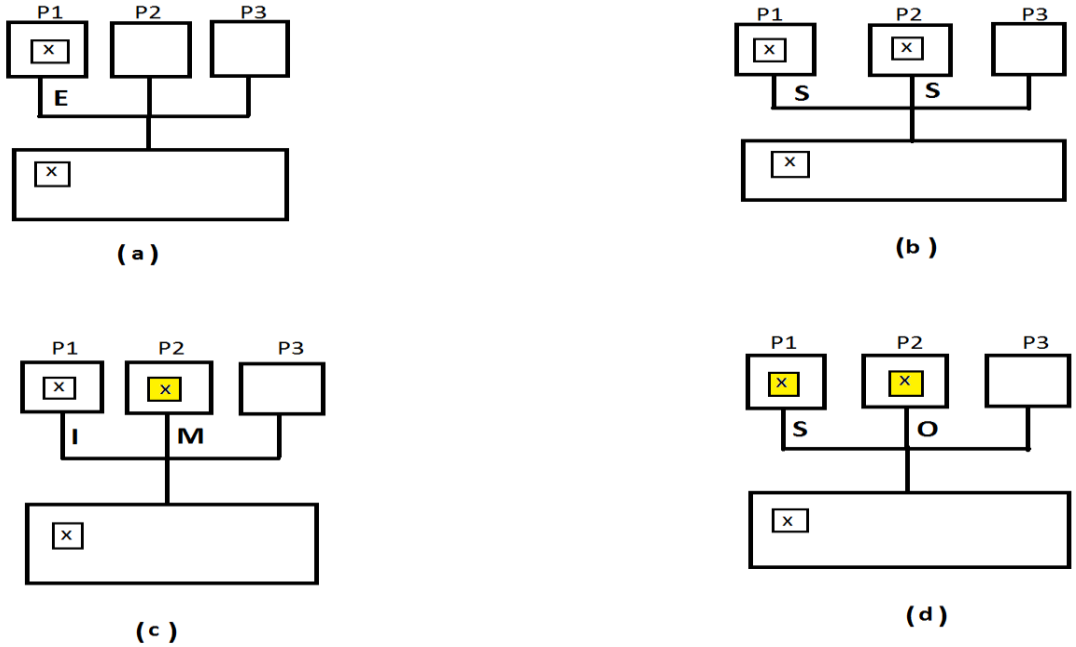

Figure3: (a) Exlcusive State, (b) Shared State, (c) Modified and Invalid States, (d) Owned and Shared State after modification and sharing of the value [1]

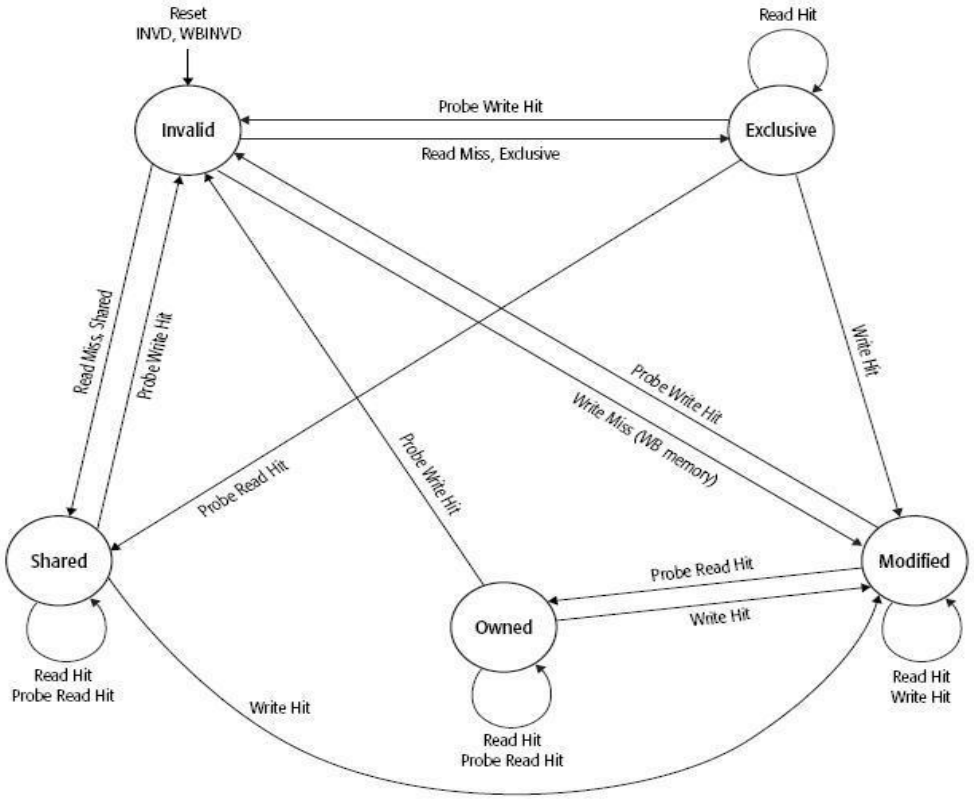

Figure 4: State Diagram for MOESI Protocol
Source: http://wiki.expertiza.ncsu.edu

3. COMPARISON BETWEEN MESI AND MOESI PROTOCOLS

The main comparison between MESI and MOESI is the fifth state in MOESI, which is the *Owned* state. Because of the owned state, now cache lines can be shared with other caches after any modification being performed on them without even updating the value in the address in the main memory at that moment. For some programs such as for example a program in multiprocessor to find the sum array of numbers, if MOESI protocol is used then the cache will perform better because every time after updation of the transition value in each cell of the array, the cache do not need to access the main memory.

4. RECENT RESEARCH AND ADVANCEMENT IN CACHE COHERENCY PROTOCOLS

In this section we are going to review different cache coherency protocols, that are used apart from MESI and MOESI protocols, and the recent development or research in the field of cache coherency.

Although, MESI and MOESI are widely used cache coherency protocols but there are also other coherency protocols that are used in different computer architecture to achieve different level of performance.

In the year, 2002, Nicoletta et al. (the concept was proposed in 1999) [6] published a modified version of MESI protocol along with a new RISC architecture, and named it MERSI cache coherency protocol, which was mostly used in PowerPC G4 computers by Apple Inc. In this protocol, the state 'R' was introduced, which stands for Read only or Recent state, other than that all the other states are similar to MESI protocol. The main key point of R state in the system is that it is similar to the E state i.e. Exclusive state, but the only constraint is that this copy of data is the only clean and valid in the computer system. And the processor, which wants to modify the cache line, first has to claim ownership of the cache line in R state, and then it can change the state of that cache line to M state.

Again, in the year 2007, David Kanter [8] published about the cache coherency protocol that Intel has built for its ccNUMA (cache coherent non uniform memory architecture). This coherency protocol was named as MESIF protocol. All the states such as M, E, S, I are same as MESI protocol but the F state i.e. the Forward state, is a specialized form of Shared or S state, and this enables the cache with F state to respond to any request for the given cache line. Since, MESI protocol, which is used in ccNUMA, would send a lot of redundant messages among different nodes, often with high latency, and that results in wasting a bit of bandwidth. For example when a processor request for a cache line, which is stored in multiple locations, then every location might respond with the data, and that may lead to waste of bandwidth. To solve this issue, Intel modified the MESI protocol with a new F state, which actually defines that the cache with that state can forward the cache line when it is requested for, instead of other locations to respond back with the same cache line when the request comes in.

Again, as technology advances, the processors are becoming powerful yet smaller in size, and with the usher of the mobile technology, we can see the production of powerful processors for mobile systems such as mobile phones. In the mobile smartphone market, the most prevalent architecture is the Shared-memory Multi-core System-on-chip (MC SoC) because they provide low communication latency, simplified programming model and decentralized topology, which is well suited for heterogeneous processing cores. In MCSoC, the most common cache coherency protocol used is the MESI protocol. But since the MESI protocol is most commonly used in MCSoC system, that may give rise to bandwidth loss over snoopy bus, and for that reason, Bournoutian [9] of the University of California, San Diego, proposed in 2011, a powerful optimized mixture of write-back and write-through MESI protocol with an extra hardware modification in order to increase performance and power efficiency of the mobile processors. In the proposal, write-back is used when there are numerous updates on the cache line and that gives more performance, whereas write-through is used in order to achieve power optimization. Another addition to this proposal is that it introduced small number of bits annotated to each cache line in order to keep track of recent state change, specially by introducing small shift register on each cache line. Whenever there is a change in the state the shift register stores bit by performing shift operators.

5. CACHE COHERENCY MECHANISMS

Cache coherency mechanism [5] is the way in which cache coherency protocols are implemented in the multiprocessor system and the consistency of memory is managed. There are 3 different ways of implementing cache coherency protocols and they are the following:

Directory Based Coherency: In this method, the data being shared are kept in a common directory, which maintains the coherency between caches by filtering different request for data from different processors. If any changes are made to an entry then the directory either updates it or invalidate it in all other caches with that entry.

Snooping Based Coherency: In this method, each cache monitors the address lines so that to gain access to main memory which they have cached. Any activity on cache line will trigger message, which will be broadcasted to all the caches to update the cache line with the activity.

Snarfing: In this method, caches monitor both the address and the data in order to update its cache line from the main memory when another cache tries to update that cache line.

The two most commonly used coherency mechanisms are directory based and snooping based coherency. both have their own pros and cons and it depends on the system itself.

For our project the author used snooping based coherency mechanism.

6. COMPARISON BETWEEN SNOOPING BASED COHERENCY AND DIRECTORY BASED COHERENCY

The basic difference can be said that in snooping each cache broadcast about its activity on the cache line to all other caches of other processors. Whereas, in directory based coherency, directories are maintained in order to monitor the activities on the cache lines. In a system where there is high bandwidth, snooping performs better because it requires to broadcast the activities. But snooping will perform worse than directory based in big systems because bus size will increase. That also means that snooping is non scalable whereas directory performs faster in big systems due to its scalable characteristic.

7. RESULTS OF THE CACHE SIMULATION

7.1 Implementation Details

A simulator for a write-back cache, which implements both MESI and MOESI protocols is written in java. The cache has a two level cache hierarchy-a private L1 cache for which the cache size is configurable and a shared main memory cache for which the cache size is assumed to be infinite for simplicity. The cache line in both the L1 cache is assumed to be 64 bytes in size. The application simulates only data caches and does not take into account instruction caches, set-associativity or latencies.

An LRU replacement strategy is used to flush a cache line. The cache supports cache-to-cache sharing of data. In MESI protocol, when a processor P1 requests for a cache line shared by cache of another processor P2, if P1 is in modified state, it writes back the modified value to the main memory and also send to the processor P2 and in MOESI protocol, in a similar situation, P1 updates it state from MODIFIED to OWNED and sends the cache line to P2 without updating the main memory. The main memory is updated only when the cache line is flushed from the cache of P1.

The cache simulator application simulates a chip multiprocessor environment and takes the number of cores as an input from command line. When the application is started, the program creates a number of threads which act as cores and each core starts sending out random read and write requests for memory addresses in the range 1000-10000 to its cache. A snoopy bus is also implemented which handles the coherency and consistency of the data in the caches. All communication between the caches is synchronized so as to maintain validity of the data being shared.

The data used in the implementation is of data type *long* which means each cache line can hold 8 values (64 bytes/8 bytes per long value). The core sends out write request to write random long values to the cache and also when a core reads for the first time from an address, a random value is generated, which is updated in the main memory and then loaded to the private cache of the core.

The whole program is provided in the appendix section in order to completely understand the basic implementation level of the simulator.

7.2 Performance Analysis

The application, was tested for 15 minutes for 8 cores, which sends random read or write requests every 10 milliseconds for both MESI and MOESI protocols for cache size of 64 kilobytes (1000 cache lines) and the following results were obtained, which is shown in Table 3.

Table 3: Performance of the Simulator

	Total Requests	Read Hits	Write Hits	Read Miss	Write Miss	Main Memory Access
MESI	720050	555939	61175	55650	47286	74849
MOESI	720040	534634	13475	77395	94536	124315

Contrary to what is expected, the number of memory accesses is greater in MOESI protocol as compared to that in MESI protocol. This can be attributed to the fact that there is a chance that many more cache lines can move to invalid state when a number of cores are accessing the same data. Even if a cache line is in Owned state, it may move to invalid state due to a write in another shared cache. Theoretically the number of memory accesses is higher in the case of MESI protocol as compared to MOESI protocol when spatial and temporal locality is considered. But when multiple processors are updating same data, there is more read misses in MOESI protocol as the modified bits are not stored into the lower level cache or main memory when another processor requests the data.

Both the protocols were tested in 2 processors, 4 processors and 8 processors for the same configuration as the first experiment for 2 minutes and the following results were obtained, which is shown in Table 4.

Table 4: Test Results

		Total Requests	Read Hits	Write Hits	Read Miss	Write Miss	Main Memory Access
2 cores	**MESI**	240127	194904	29136	9172	6915	11499
	MOESI	240138	188237	22549	15779	13573	26130
4 cores	**MESI**	480330	377119	45020	31379	26812	46497
	MOESI	480823	358727	18244	49659	54193	84429
8 core	**MESI**	960460	742840	80468	73050	64102	102158
	MOESI	960214	715670	17888	100693	125963	165984

From these results it can be noticed that the MOESI protocol is slower than MESI protocol as it handles lesser number of requests in the same time as compared to MESI protocol, which is caused by the fact that MOESI takes more cycles to complete a read or write transaction. Also, in MOESI protocol, it can be observed that the ratio of write hits decreases and more and more write misses occur as the number of cores increase which is due to more invalidations as explained before.

The invalidation of cache lines in both the protocols form a major part of the bus traffic and may increase very much as the number of cores increase thus slowing down the system. More advanced protocols can be implemented with invalidation queues to avoid the entire system getting stalled due to the invalidate signals.

8. LIMITATIONS

In this paper, the authors only focused on the working of MESI and MOESI cache coherency protocols over a snoopy bus, and the correctness of the states and transitions related to those protocols. The authors did not focus on the latencies or set associativity or replacements. Hence, all the results and discussions are focused on novel cache coherency protocols only. Another limitation is that the authors only studied the cache coherency protocols on private caches of each core, so there is only level 1 cache for each core. One more thing that should be kept in mind that this paper does not deal with a full featured cache simulator and the main motivation of this paper is to provide new researchers and students of computer science, the basic idea to design and implement cache simulator in real systems.

9. CONCLUSION

Hence, in this paper we review both the advantages and disadvantages of MESI and MOESI protocols, and how they are used in different architectures in order to achieve cache coherency. The cache coherence protocol is one of the major

factors influencing the performance of multi-core computer systems. The coherence protocol must be selected based on the chip architecture and the performance that the system wants to achieve. MOESI protocol is an extension for MESI protocol, which minimizes access to lower level cache or main memory by introducing an ownership state. The results provided in this paper, prove that MOESI will have significant advantage over MESI when remote memory accesses are much costlier than L1 memory access. However, it takes more cycles to complete transactions and is not advantageous when considering sharing of clean cache lines. The program code to build the simulator is provided in the appendix section. For further tinkering with the simulator, it can be referred from the appendix section.

10. ACKNOWLEDGMENTS

SD and MSN would like to thank the teaching and research staffs, especially Professor Norman Paton and Professor Uli Sattler of the School of Computer Science, The University of Manchester for their enthusiastic support. This mini research project was supported by the funds available at the School of Computer Science of The University of Manchester.

11. REFERENCES

[1] Herlihy, M., Shavit, N., "The Art of Multiprocessor Programming", Elsevier.

[2] Hwang, K., Xu, Z., "Scalable Parallel Computing: Technology, Architecture, Programming". McGraw-Hill, New York, NY, 1998. ISBN 0-07-031798-4.

[3] Papamarcos, M. S., Patel, J. H., "A low-overhead coherence solution for multiprocessors with private cache memories". Proceedings of the 11th annual international symposium on Computer architecture - ISCA '84 (1984). p. 348.

[4] Neupane, M., "Cache Coherence", California State University San Bernardino, 2004, Online. [Available at http://cse.csusb.edu/schubert/tutorials/csci610/w04/MN_Cache_Coherence.pdf] [Accessed on 07/12/2013]

[5] Patterson, D., Hennessy, J., "Computer Organization and Design (4th ed.)". Morgan Kaufmann, 2009.

[6] Nicoletta, C., Alvarez, J., Barkin, E., Chai-Chin Chao, Johnson, B. R., Lassandro, F. M., Patel, P., Reid, D., Sanchez, H., Seigel, J., Snyder, M., Sullivan, S., Taylor, S. A., Minh Vo., (November 1999). "A 450-MHz RISC microprocessor with enhanced instruction set and copper interconnect". *IEEE Journal of Solid-State Circuits* **34** (11): pp. 1478–1491.

[7] Hennessey J. L., Patterson, D. A., "Computer Architecture: A Quantitative Approach".

[8] Kanter D., "The Common System Interface: Intel's Future Interconnect". Real World Tech: 5, Online. [Accessed on 07/12/2013]

[9] Bournoutian, G., Orailoglu, A., "Dynamic, multi-core cache coherence architecture for power-sensitive mobile processors", Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2011, pp. 89-97.

[10] Dey, S., Nath, J., Nath, A., "An Integrated Symmetric Key Cryptographic Method – Amalgamation of TTJSA Algorithm, Advanced Caesar Cipher Algorithm, Bit Rotation and Reversal Method: SJA Algorithm", IJMECS, vol.4, no.5, pp.1-9, 2012.

[11] Dey, S., "SD-REE: A Cryptographic Method To Exclude Repetition From a Message", Proceedings of The International Conference on Informatics & Applications (ICIA 2012), Malaysia, pp. 182 – 189.

[12] Dey, S., "SD-AREE: A New Modified Caesar Cipher Cryptographic Method Along with Bit- Manipulation to Exclude Repetition from a Message to be Encrypted", Journal: Computing Research Repository - CoRR, vol. abs/1205.4279, 2012.

[13] Dey, S., "An Image Encryption Method: SD-Advanced Image Encryption Standard: SD-AIES", International Journal of Cyber-Security and Digital Forensics (IJCSDF) 1(2), pp. 82-88.

[14] Dey, S., Nath, J., Nath, A., "An Advanced Combined Symmetric Key Cryptographic Method using Bit Manipulation, Bit Reversal, Modified Caesar Cipher (SD-REE), DJSA method, TTJSA method: SJA-I Algorithm". *International Journal of Computer Applications46(20):* 46-53, May 2012. Published by Foundation of Computer Science, New York, USA.

[15] Dey, S., "SD-EQR: A New Technique To Use QR CodesTM in Cryptography", Proceedings of "1st International Conference on Emerging Trends in Computer and Information Technology (ICETCIT 2012)", Coimbatore, India, pp. 11-21.

[16] Dey, S., "SD-EI: A Cryptographic Technique To Encrypt Images", Proceedings of "The International Conference on Cyber Security, CyberWarfare and Digital Forensic (CyberSec 2012)", held at Kuala Lumpur, Malaysia, 2012, pp. 28-32.

[17] Dey, S., "SD-AEI: An advanced encryption technique for images", 2012 IEEE Second International Conference on Digital Information Processing and Communications (ICDIPC), pp. 69-74.

[18] Dey, S., "Amalgamation of Cyclic Bit Operation in SD-EI Image Encryption Method: An Advanced Version of SD-EI Method: SD-EI Ver-2", International Journal of Cyber-Security and Digital Forensics (IJCSDF) 1(3), pp. 238-242.

[19] Dey, S., Mondal, K., Nath, J., Nath, A., "Advanced Steganography Algorithm Using Randomized Intermediate QR Host Embedded With Any Encrypted Secret Message: ASA_QR Algorithm", IJMECS, vol.4, no.6, pp. 59-67, 2012.

[20] Dey, S., Nath, A., "Modern Encryption Standard (MES) Version-I: An Advanced Cryptographic Method", Proceedings of IEEE 2nd World Congress on Information and Communication Technologies (WICT- 2012), pp. 242-247.

[21] Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg. G., "Simics: A full system simulation platform". IEEE Computer, 35(2):50-58, Feb. 2002.

[22] Nurvitadhi, E., Chalainanont, N., Lu, S. L., "Characterization of L3 Cache Behavior of SPECjAppServer2002 and TPC-C." In Proceedings of the 19th International Conference on Supercomputing (ICS), Boston, Massachusetts, 2005.

[23] Uhlig. R. A., Mudge, T. N., "Trace-driven Memory Simulation: A Survey", In ACM Computing Surveys, Vol. 29, 1997.

[24] Luk, C. K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., Hazelwood, K., "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation." In Proceedings of Programming Language Design and Implementation (PLDI), Chicago, Illinois, 2005.

APPENDIX

Code for the Program of Cache Simulator in Java is available from the following link:

https://www.escholar.manchester.ac.uk

Manchester eScholar ID: uk-ac-man-scw:218836