Simulation based Performance Study of Cache Coherence Protocols

Neethu Bal Mallya, Geeta Patil and Biju Raveendran
BITS Pilani K.K. Birla Goa Campus, Goa
{h2009191,geetapatil,biju}@goa.bits-pilani.ac.in

***Abstract*—Cache coherence protocol maintains data consistency between different cores / processors in a shared memory multi-core (MC) / multi-processor (MP) system. Coherency can be achieved at the cost of increased miss rate because of invalidations. Coherency misses and the number of signals for maintaining data in consistent state consumes additional time and energy. This paper studies the impact of cache coherence misses, invalidations and additional signals due to MI, MESI and MOESI cache coherence protocols implemented in Gem5 – the most widely used full system simulator. The experimental studies show that the dynamic energy consumption due to cache miss in MI, MESI and MOESI protocols are 53.6%, 31.2% and 31.1% for 32KB L1 cache and 46.3%, 23.0% and 22.1% for 64KB L1 cache respectively. The average number of signals per access in case of MI, MESI and MOESI protocols is 4.23, 4.16 and 4.19 respectively for SPLASH-2 benchmarks suits.**

Keywords—cache memory; coherence protocol; MC/MP cache; gem5 simulator;

I. INTRODUCTION

With advancement in technology, the processor design achieved a paradigm shift from uni-processor systems to MC/MP systems. A shared memory MC/MP system has more than a core/processor (node). L1 cache is dedicated to the node whereas L2 cache, L3 cache and main memory are shared across all the nodes. This results in possibility of having multiple copies of data in different locations. Nodes access data from dedicated caches than accessing it from other memory levels as this transfer is much faster than transferring data from other levels. This reduces the number of L2/L3/main memory accesses as well. It is possible that the cached data is modified in one of the nodes and these modifications are not reflected in other nodes, leading to data inconsistency among nodes. Thus, maintaining data consistency becomes a major challenge in shared memory MC/MP systems. Cache consistency issues lead to another category of cache misses named as coherence misses along with compulsory, capacity and conflict misses. Coherence miss occurs when the local cache copy of a node is updated and all the shared copies are invalidated to maintain data coherence.

The most widely used cache coherence protocols are MI, MSI, MESI, MOSI, MOESI, and MESIF. They differ in stable states, transient states and the communication signals transmitted to maintain consistent data. The energy required to send signals between nodes depends on factors like interconnect length, bus width, bus speed, etc. These factors are constants for a given system architecture hence the additional energy consumption in interconnect of a MC/MP system must be dealt by reducing signal transmissions.

The paper discusses related work on cache coherence solutions in Section II. Some of the existing cache coherence protocols are briefed in Section III. Section IV describes implementation of cache coherence protocols in Gem5 – the most widely used full system simulator. Signal energy modeling for analyzing various protocols is discussed in Section V. Performance evaluation is presented in Section VI. Section VII discusses the implementation issues in current MOESI implementation and suggestions to improve the MOESI performance. Section VIII concludes the paper.

II. RELATED WORK

Cache coherence solutions are either hardware-based [1-3], software-based [4, 5] or a combination of both. Hardware cache coherence schemes are categorized as directory-based, snoopy-based or hybrid. Directory-based schemes maintain a centralized directory to store the sharing status of memory block and thus have lower broadcast requirements. In snoopy-based schemes, cache controller broadcasts the request and monitors the memory bus activity to maintain consistent data. Snoopy schemes are not scalable, while directory schemes incur storage overhead and are more prone to design bugs. Several techniques have been proposed which optimizes the coherence traffic and overhead in conventional directory and snoopy schemes [6-9]. The hybrid approach dynamically adapts to snoopy or directory scheme based on interconnect network utilization [10]. Such schemes broadcast (snoopy) or unicast (directory) requests based on the availability of bandwidth.

Within snoopy there exist two techniques based on write operation: write-invalidate and write-update [11, 12]. In write invalidate, node invalidates all other cached copies of shared data and can then update its own copy without further bus operations whereas in write-update, node broadcasts update data to other shared caches, so that all copies are the most updated ones.

There exist software alternatives on MC/MP memory hierarchies which reduce the hardware complexity of cache coherence using software programming models. DeNoVo [4, 5] is a hardware-software co-designed protocol which uses software programming model on a hardware architecture.

Stenstrom [11], Archibald and Baer [12] analyzed various snoopy based protocols theoretically. This paper does a simulation based performance study of MI, MESI and MOESI protocol implementations in Gem5 simulator [13, 14].

978-1-4673-9692-9/15 $31.00 © 2015 IEEE
DOI 10.1109/iNIS.2015.52

III. CACHE COHERENCE PROTOCOLS

A. MI Protocol

Modified-Invalid (MI) is the simplest protocol in use for maintaining cache coherence in MC/MP systems. It uses two states: Modified (M) and Invalid (I). The cache line in M state implies that the cache line data can be read or written by the node and no other node holds a valid copy of that data block. Data present in a cache line with I is invalid. The hardware complexity of MI protocol is the least among all the coherence protocols. Read/write in a node results in invalidating the data in other node if the same data exists. This floods the network with large number of invalidation signals which is reduced by using S state in Modified-Shared-Invalid (MSI) protocol.

B. MSI Protocol

In MSI protocol, the read misses are satisfied by transferring the requested data from the next level memory and setting the cache line state to S. One or more nodes can have valid copy of data. Data sharing reduces the number of coherence misses as compared to MI. If the cache line is in M state then it is modified and is the most up-to-date and only valid copy available. In case of write on a shared copy of data, the invalidation signal is sent to all other nodes irrespective of whether there exist a shared copy or not. This result in sending unwanted invalidation signals if modifying node has the exclusive valid copy of data, which is addressed by using E state in Modified-Exclusive-Shared-Invalid(MESI) protocol.

C. MESI protocol

MESI is one of the most widely used cache coherence protocol for MC/MP systems. In MESI, for every state change from M, the data needs to be written back. This may result in frequent data writes between the node and next level when read and write alternate. This can be eliminated by using O state in Modified-Owned-Shared-Invalid (MOSI) protocol.

D. MOSI protocol

In MOSI protocol, the node can transfer modified data block to the read requestor without writing it back. The state of the cache line in the requesting node and the responding node is updated to S and O respectively. Thus, O state allows sharing of dirty copy between different nodes. But the absence of E state results in sending invalidation signals to other nodes while attempting write to an exclusive copy. This is eliminated by combining the states of MESI and MOSI protocols in Modified-Owned-Exclusive-Shared-Invalid(MOESI) protocol.

E. MOESI Protocol

In MOESI protocol, the E state avoids sending invalidation signals to other nodes whenever the node upgrades its state to M. The existence of O state reduces the frequency of write-back operations by sharing the dirty data. However, on a cache miss, all the sharers of requested data respond and the requestor is serviced by redundant responses. These redundant responses increase the traffic across the network. This is addressed by introducing F state in Modified-Exclusive-Shared-Invalid-Forward (MESIF) protocol.

F. MESIF Protocol

MESIF protocol ensures that only the cache line in F state responds to the read/write request of other nodes. Among the sharers, the last recipient of data is assigned with F state. During read miss, the cache line in F state transfers the data and updates its state to S. Although redundant responses are eliminated by using F state, sharing of dirty data is not allowed in MESIF.

IV. CACHE COHERENCE PROTOCOLS IN GEM5

Gem5 simulator provides a flexible memory system model with multiple coherence protocols [13,14]. This paper considers MI_example, MESI_Two_Level and MOESI_CMP_directory implementations for the case study. Specification Language for Implementing Cache Coherence (SLICC) in Gem5, implements coherence protocols as a set of states, events, transitions and actions. The cache coherence events common for all protocols are described in Table I.

TABLE I. CACHE COHERENCE EVENTS

Event (E)	*Description*	*E#*
Load	Load request from the node	1
Ifetch	I-fetch request from the node	2
Store	Store request from the node	3
Inv	Invalidate request from L2/directory	4
WB_Ack	ACK from L2/directory for a write back	5
WB_Nack	NACK from L2/directory for a write back	6

A. MI Protocol

In Gem5, MI protocol is implemented with single level private cache. Table II and III describe the states and the cache events in Gem5 MI implementation.

TABLE II. BASE AND TRANSIENT STATES IN MI PROTOCOL

State	*Access Permission*	*Description*
I	Invalid	Data in cache line is invalid.
II	Busy	Data in cache line is invalid, issued PUT request
M	Read_Write	Data can be read/written by the node exclusively. No other node holds a copy.
MI	Busy	Node issued PUT request. Waiting to write data to directory.
MII	Busy	Node issued PUTX, received NACK. Waiting to forward data to directory.
IS	Busy	Node is waiting for response after issuing LOAD/IFETCH request
IM	Busy	Node is waiting for response after issuing STORE request

TABLE III. CACHE COHERENCE EVENTS IN MI PROTOCOL *

Event (E)	*Description*	*E#*
Data	Data from network	7
Fwd_GETX	Data forward request from network	8
Replacement	Replacing a cache line (from self)	9

*In addition to the events in Table. I

When request from the home node is *Load*, *Ifetch* or *Store,* if the cache line is in M or any of the transient states, it continues to be in the same state. If the cache line is in I state, then the cache controller issues a GETX request to the next level and allocates a temporary buffer in the node. The state of the cache line is changed to the transient state IS or IM for read and write respectively. Once controller receives the data from directory, cache line state changes to M

If the cache controller receives *Inv* event and the cache line is in I, IS, IM or MI states, then it remains in the same state. If the event is *Replacement* in I or any of the transient states, it continues in the same state. If the cache line is in M state and the controller receives *Inv* or *Replacement* events, then the node issues PUT request to the next level, copies data from cache to the temporary buffer (TBE) and changes the cache line state to MI. From MI, cache line state changes to I or MII upon receiving ACK or NACK for write back.

If the cache controller receives *Fwd_GetX* event in M state, then the cache sends data and changes to I state whereas if the cache line is in IS or IM states with invalid data, it continues in the same state. On receipt of *Fwd_GetX* event in MI state, the node sends data from TBE to the requester and changes its state to II. From II on receiving NACK signal for write back, the cache line state changes to I. On receipt of *Fwd_GetX* event in MII state, the controller sends the data from TBE to the requestor and the cache line state changes to I. Fig. 1 shows the state transition diagram of MI protocol.

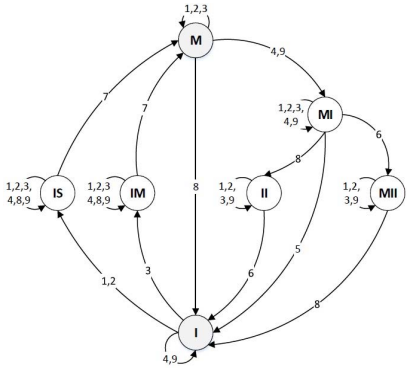

Fig. 1. State Transition Diagram of MI Protocol

B. MESI Protocol

In Gem5, MESI protocol is implemented for 2 level cache hierarchies consisting of L1 and L2 caches. L1 is a private split cache for each node and L2 is a unified shared cache. Table IV and V describe the states and the cache events in Gem5 MESI implementation.

The transitions from I or NP state are same as the transitions from I state in MI. If the cache line is in M, E or S states, the *Load* or *Ifetch* request from the node will be satisfied with no change in state. For a *Store* event in E state, it changes to M whereas if the state is M, it remains in the same state. Upon receiving *Store* request in S state, the UPGRAGE request is sent to the next level and the state changes to SM. When the cache line is in M or E state and the event is a forward request from next level, data is sent to the network. For a read request (*Fwd_GETS or Fwd_GET_INSTR*), the state of the cache line changes to S whereas for a write request (*Fwd_GETX*), it changes to I. When the cache line is in M, E or S state, if the event is *L1_Replacement* or *Inv*, the controller sends the eviction information to the node. From M or E state, *L1_Replacement* results in issuing a PUTX request and the cache line changes to M_I state. In all other cases, the cache line changes to I state.

Transient states stall and wait in same state for *Load*, *Ifetch*, *Store* and *L1_Replacement* events. From M_I state, *WB_Ack* changes the state to I. Upon *Inv* or any forwarded requests in M_I state, controller sends the data to requestor from TBE and changes the state to SINK_WB_Ack. From SINK_WB_Ack, the *WB_Ack* event changes the state to I.

TABLE IV. BASE AND TRANSIENT STATES IN MESI PROTOCOL

State	***Access Permission***	***Description***
NP	Invalid	The cache entry has not been allocated.
I	Invalid	The cache entry is invalid.
S	Read_Only	Data can be present in more than 1 node.
E	Read_Only	No other node holds a copy of the cache line.
M	Read_Write	Data has been modified by the node and no other node holds a copy.
IS	Busy	Issued GETS, have not received response yet. Cache is neither readable nor writeable.
IM	Busy	Issued GETX, have not received response yet. Cache is neither readable nor writeable.
SM	Read_Only	Issued GETX, have not received response yet. Cache line is readable.
IS_I	Busy	Issued GETS, received *Inv* before data as L2 doesn't block on GETS
M_I	Busy	Issued Write Back to L2 and waiting for ACK
SINK_WB_ACK	Busy	This is to sink WB_Acks from L2; Reaches when L1 cache receives forwarded request while waiting for WB_Ack from L2.

TABLE V. CACHE COHERENCE EVENTS IN MESI PROTOCOL*

Event (E)	***Description***	***E#***
Data	Data for node	10
Data_Exclusive#	Exclusive data for node	11
DataS_fromL1	Data for GETS request, need to unblock L2	12
Data_all_Acks	Data for node, all ACKs	13
Ack#	ACK for node	14
Ack_all	Last ACK for node	15
Fwd_GETS#	GETS from other node	16
Fwd_GETX#	GETX from other node	17
Fwd_GET_INSTR	GET_INSTR from other node	18
L1_Replacement#	L1 Replacement	19

*In addition to the events in Table. I #Signals common to MESI and MOESI

Upon receiving data from the next level or from other L1's in IS state, the controller writes the data to L1 cache and changes the state to S. If the event is *Data_Exclusive* from the next level, the state changes to E. If the event is *Inv*, then the state changes to IS_I. From IS_I, on receiving *Data_Exclusive*, the state changes to E whereas on receiving *DataS_from_L1* or *Data_All_Acks*, the state changes to I. If the state is IM, on *Data* event it changes to SM whereas on

receiving *Data_All_Acks* event, it changes to M. In SM state, the controller waits until it receives ACKs from all other nodes having the requested data. Upon receiving *Ack_all*, the state changes to M. If *Inv* event occurs while in SM, the state changes back to IM. Fig. 2 shows the state transition diagram of MESI.

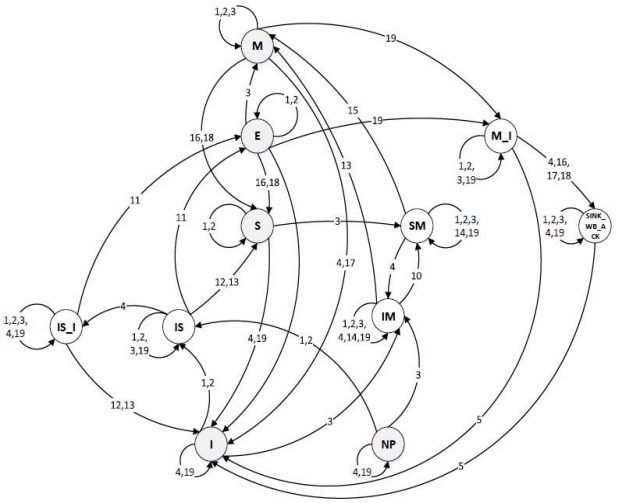

Fig. 2. State Transition Diagram of MESI Protocol

C. MOESI Protocol

Like MESI, the MOESI protocol implementation in Gem5 also has two level memory hierarchies consisting of L1 and L2 caches. Table VI and VII describe the states and the cache events in Gem5 MOESI implementation.

From I state, for *Load or Ifetch* event, the cache controller issues a GETS request and the state is changed to IS. Upon *Load* or *Ifetch* request in S, O, MM, M, M_W or MM_W state, the data is supplied to the node without change in state. From I state, for Store request, the cache controller issues a GETX and the state changes to IM. Node's request for *Store* in MM or MM_W state is satisfied without change in state. *Store* request in S or O state allocates TBE, issues GETX and changes the state to SM or OM. *Store* request in M state and M_W state changes the state to MM and MM_W respectively.

Upon receiving *L1_Replacement* event in S, O or M/MM state, the controller issues PUTS, PUTO or PUTX signal to the next level. The controller also sends eviction information to the node and changes the state to SI, OI and MI from S, O and M/MM respectively. Upon receiving *Inv* event from next level in S state, the controller sends ACK to the requestor and eviction information to the node. The timeout states M_W and MM_W wait in the same state for *L1_Replacement*, *Inv* and forwarding events. On timeout, M_W and MM_W changes to M and MM respectively.

For *Fwd_GETS* request in M, S or O state, the controller sends the data from cache to the requestor. The state changes from M to O while it remains in same state for S and O. For forwarded read request in MM or forwarded write request in M, O or MM states, the controller sends the exclusive data to requestor, self-invalidates its own copy and changes to I state.

TABLE VI. BASE AND TRANSIENT STATES IN MESI PROTOCOL

State	*Access Permission*	*Description*
I	Invalid	Data in cache line is invalid.
S	Read_Only	Data can be present in more than 1 node.
O	Read_Only	Owned
M	Read_Only	Similar to conventional E state
M_W	Read_Only	Automatic transition to M state on timeout. Replacements are not allowed.
MM	Read_Write	Similar to conventional M state
MM_W	Read_Write	Automatic transition to MM state on timeout. Replacements are not allowed.
IM	Busy	Issued GETX
SM	Read_Only	Issued GETX, old copy of the line still exists
OM	Read_Only	Issued GETX, received data
IS	Busy	Issued GETS
SI	Busy	Issued PUTS, waiting for ACK
OI	Busy	Issued PUTO, waiting for ACK
MI	Busy	Issued PUTX, waiting for ACK
II	Busy	Issued PUTX/PUTO, received Fwd_GETS or Fwd_GETX, waiting for ACK

TABLE VII. CACHE COHERENCE EVENTS IN MOESI PROTOCOL*

Event (E)	*Description*	*E#*
Data	Received data, responder has a shared copy	20
All_acks	Received all required data and message ACKS	21
WB_Ack_Data	Write back O.K. from directory	22
Fwd_DMA	A GetS from another node	23
Own_GETX	Own GetX forwarded back to the node.	24
Use_Timeout	Lockout period ended	25

* In addition to the events in Table. I and IV

Transient states wait in the same state for *Load*, *Ifetch*, *Store* and *L1_Replacement* events. *Inv* event in IS, IM, SI or II state sends ACK to the requestor. If *Inv* event is in SM state, the controller sends ACK to the requestor as well as it sends eviction information to the node. It changes the state to IM.

Upon receiving data and exclusive data, if the cache line is in IS state, the controller writes the data to cache and changes the state to S and M_W respectively. If the cache line is in IM state, the controller writes the data to cache and changes its state to OM. If the cache line is in SM state, the controller checks for completion and changes to OM state.The state continues in IM, SM or OM on *Ack* event. Upon receiving *All_acks* in OM state, the controller sends unblock event to the next level and changes the state to MM_W.

For *Fwd_GETS* request in SM or OM state, the controller sends the data to the requestor and remains in the same state. If the state is MI, SI or OI, the controller sends the data from TBE to cache. MI state changes to OI whereas SI and OI remains in same state. For *Fwd_GETX* request in OM and MI/OI state, the controller sends the exclusive data to requestor and changes the state to IM and II respectively.

Upon receiving ACK for write back in SI, OI, MI or II state, the controller sends unblock to the next level, deallocates TBE and changes the state to I. For

Writeback_Ack_Data event in SI, OI or MI states, the controller sends data from TBE to L2, deallocates TBE and changes the state to I. If the state is II, the controller sends unblock to the next level and changes the state to I. Upon receiving NACK for write back in MI or OI state, the controller issues PUTO and changes the state to OI. If the state is SI, the controller issues PUTS to the next level and remains in same state. If the state is in II, the controller deallocates TBE and changes the state to I. Fig. 3 shows the state transition diagram of MOESI.

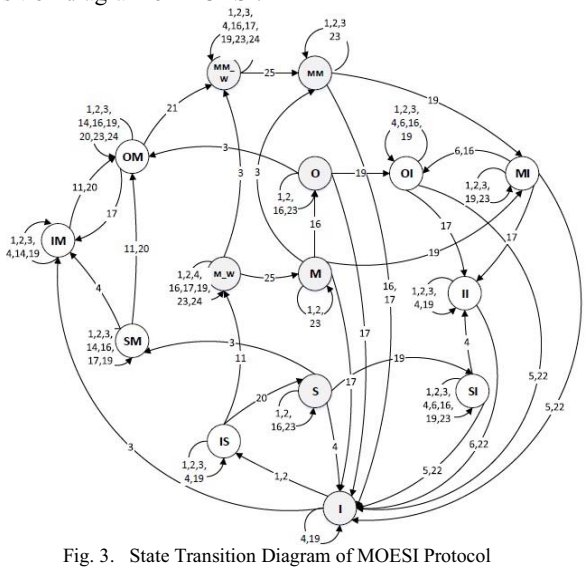

Fig. 3. State Transition Diagram of MOESI Protocol

V. SIGNAL ENERGY MODELING

The coherence events and actions are classified into 8 different categories based on the components in communication. The categories are shown in Table VIII.

TABLE VIII. SIGNAL CATEGORIES FOR COHERENCE EVENTS & ACTIONS

	Signal Category	*Description*
Events	E1	Generated from the node
	E2	Generated from other node
	E3	Generated from the next level memory
Actions	A1	Communication within the node
	A2	Communication within L1's
	A3	Communication from L1 to L2
	A4	Either A2 or A3
	A5	Node Stall

Action A4 occurs only during forwarding requests in MOESI implementation. It is observed that most of the time forward requests are satisfied by other L1's. Hence, as an optimization measure, action A4 is considered to have same energy consumption as A2.

The total energy consumption due to different signal categories is calculated as:

$$E_{Event} = \{0.04*E1 + 0.1*E2 + 0.2*E3\} * E_{Node\_Runtime\_Dynamic}$$
$$E_{Action} = \{0.04*A1 + 0.1*A2 + 0.2*A3\} * E_{Node\_Runtime\_Dynamic}$$

Stall energy (from action A5) is not considered as part of the signal energy calculation as it is calculated in execution phase energy calculation.

VI. PERFORMANCE EVALUATION

Cache coherence overhead becomes a major performance bottleneck for shared memory MC/MP systems. As the number of nodes per chip increases, coherence misses, invalidations, additional signals to maintain coherency, cache latency and energy consumption increases.

The objective of analyzing various coherency protocols is to measure efficiency of various protocols with respect to parameters like number of cache misses, total program execution time, number of invalidations, additional signals and additional energy required for different protocols.

A. Evaluation Methodology

The evaluation of cache coherence protocols is done by using Gem5 full system simulation of SPLASH-2 benchmarks [15]. The simulation used Ruby memory model and ALPHA instruction set architecture in Gem5. Analysis of the generated statistics file is done to find the number of cache misses, invalidations, the different categories of coherence events and actions. The Gem5 statistics and configuration files were imported to McPAT[16-18] framework to obtain energy consumption in various scenarios.

B. Simulation Results

Simulation results obtained with 2-way set-associative split L1 cache of 32KB I-Cache, 32KB D-Cache and unified L2 cache of 8MB for FFT benchmark is shown in Table IX.

It is evident from Table IX that the miss rate is highest for MI and the least for MOESI with 2 and 4 nodes. It is also observed that for 2 nodes, MOESI has higher invalidations per total accesses as compared to MESI. This is because the implementation of MOESI in Gem5 does not consider sharing of dirty data i.e., forwarded read requests in modified state invalidates itself instead of sharing the data with the requestor.

TABLE IX. MISSES AND INVALIDATIONS FOR FFT BENCHMARK

#Nodes	*Protocol*	*Miss Rate*	*Invalidations/ Total Accesses*
2	MI	0.0547797	0.0553302
	MESI	0.0318877	0.0317746
	MOESI	0.0317905	0.0318745
4	MI	0.1436105	0.1459334
	MOESI	0.0289639	0.0290480

The miss rate of the protocols for different benchmarks is shown in Fig. 4. For all the benchmarks analyzed, miss rate is the highest for MI and the least for MOESI. Invalidations per accesses also shows similar trend for different benchmarks.

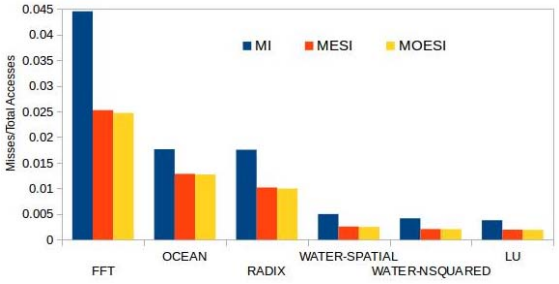

Fig. 4. Miss Rate for different SPLASH-2 benchmarks

Signal runtime dynamic energy consumed for different SPLASH2 benchmark suits is shown in Fig. 5. Energy consumption in MI protocol is the highest due to highest number of invalidation signals. Energy consumption in MOESI is higher than MESI due to more number of states and more number of transitions in MOESI implementation as compared to MESI implementation.

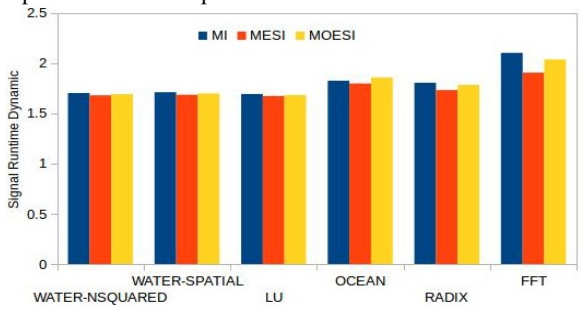

Fig. 5. Signal Runtime Dynamic Energy for SPLASH-2 benchmarks

Table X shows the signal runtime dynamic energy for all three protocols for FFT benchmark. It is evident that dynamic energy consumed by 64KB L1 cache is less than dynamic energy consumed by 32KB L1 cache as miss rate reduces with increase in cache size.

TABLE X. SIGNAL RUNTIME DYNAMIC ENERGY FOR FFT BENCHMARK

	Size of L1 Cache	
Protocol	***32KB***	***64KB***
MI	2.2197	2.1242
MESI	1.9666	1.8808
MOESI	2.1701	1.9947

VII. MOESI IMPLEMENTATION ISSUES AND SUGGESTIONS

From analysis, it is observed that MESI outperforms MOESI in some scenarios. Performance of MOESI is not as expected because of non-shared dirty data and unnecessary write backs from S/E state. In current MOESI implementation, the cache data is invalidated upon receiving *Fwd_GETS* in modified state. Hence, the request for modified data requires a write back although L1 to L1 transfer is allowed in MOESI. This increases the miss rate and number of invalidations. Instead of invalidating the cache data sharing of dirty data should be allowed to overcome this issue.

In current implementation, *Store* event in S state transits the state to SM where the controller waits for data. This wait is unnecessary as S state has the consistent copy. Store in S state should wait for invalidation acknowledgements from other sharers or from the next level. Upon receiving *All_acks* event, the cache line state should change to M.

L1_Replacement in S, O and E states result in writing back to the next level. However, the cache line is consistent with the next level when it is in S, O or E states. Thus write back is an additional overhead.

VIII. CONCLUSION

Cache coherence protocols significantly impacts system performance with respect to latency, bandwidth utilization and energy consumption. A simulation based performance study of MI, MESI and MOESI protocol is presented in this paper. MI protocol has least hardware requirements and hence it is suitable in a scenario where there is no data sharing between applications executing on different nodes. As the degree of data sharing between applications increase, total number of invalidations and hence cache miss rate of MI protocol increases exponentially with increase in number of nodes. MESI protocol can be used when read only data is shared between nodes whereas when read-write operation alternate between two nodes MOESI outperforms MESI.

References

[1] A. Agarwal, R. Simoni, J. Hennessy and M. Horowitz, "An evaluation of directory schemes for cache coherence", in Proc. of the 15th Annu. Inter. Symp. On Computer Architecture (ISCA), May 1988,pp. 280-289.

[2] D. E. Culler, J.P. Singh and A. Gupta, "Parallel Computer Architecture: A Hardware/Software Approach", Morgan Kaufmann Publishers Inc., San Francisco, CA, 1997, pp. 259-341.

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

[4] B. Choi, R.Komuravelli, et.al., "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism", in Proc. of the 20th Inter. Conf. on Parallel Architectures and Compilation Techniques, Oct 2011,pp. 155-166.

[5] R. Komuravelli, S.V. Adve and C. Chou, "Revisiting the Complexity of Hardware Cache Coherence and Some Implications", ACM Trans. on Architecture and Code Optimization, Vol. 11, Issue. 4, Article No. 37, Jan 2015.

[6] N.D.Jerger, "Chip Multiprocessor Coherence and Interconnection System Design", Ph.D. Dissertation, Dept. Elect. Eng., Uni. Of Wisconsin-Madison, 2008.

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

[8] A. Gupta, W. Weber and T. Mowry, "Reducing Memory and Traffic Requirements for Scalable Directory-based Cache Coherence Schemes", in Proc. of the Inter. Conf. on Parallel Processing, Aug 1990,pp. 312-321.

[9] J. A. Brown, R. Kumar and D. Tullsen, "Proximity-Aware Directory-based Coherence for Multi-core Processor Architectures", in Proc. of 19th Annu. ACM Symp. on Parallel Algorithms and Architectures (SPAA '07), June 2007,pp. 126-134.

[10] M. K. Martin, D. J. Sorin, M. D. Hill and D. A. Wood, "Bandwidth Adaptive Snooping", in Proc. of the 8th Annual Inter. Symp. on High-Performance Computer Architecture (HPCA-8), Feb 2002,pp. 251-262.

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

[12] J. K. Archibald and J. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model", ACM Trans. on Computer Systems, Vol. 4, Issue. 4, Nov 1986, pp. 273-298.

[13] Available at: www.gem5.org, cited on 28th March 2015.

[14] N. Binkert, B. Beckmann, G. Black, et. al., "The gem5 Simulator", ACM SIGARCH Computer Architecture News, May 2011.

[15] S. C. Woo, M. Ohara, et. al., "The SPLASH-2 programs: characterization and methodological considerations", in Proc. of 22nd Annu. Inter. Symp. On Computer Architecture (ISCA), May 1995, pp. 24-36.

[16] Available at: www.hpl.hp.com/research/mcpat cited on 28th March 2015.

[17] S. Li, J.H. Ahn, R.D. Strong, et. al., "McPAT: An Integrated Power, Area and Timing Modeling Framework for Multicore and Manycore Achitectures", 42nd Annu. IEEE/ACM Int. Symp. on Microarchitecture, Dec 2009, pp. 469-480.

[18] S. Li, J.H. Ahn, R.D. Strong, et. al., "The McPAT Framework for Multicore and Manycore Architectures: Simultaneously Modelling power, Area and Timing", ACM Trans. on Architecture and Code Optimization, Vol. 10, Issue. 1, Article No. 5, April 2013.