MOESIF: A MC/MP Cache Coherence Protocol with Improved Bandwidth Utilization

Geeta Patil, Neethu Bal Mallya and Biju Raveendran

Department of Computer Science and Information Systems
BITS Pilani K.K. Birla Goa Campus
Goa, India
Email:{geetapatil, h2009191, biju}@goa.bits-pilani.ac.in

Abstract: This paper proposes a novel cache coherence protocol – MOESIF - to improve the off chip and on chip bandwidth usage. This is achieved by reducing the number of write backs to next level memory and by reducing the number of responders to a cache miss when multiple copies of data exists in private caches. Experimental evaluation of various SPLASH-2 benchmark programs on CACTI 5.3 and CACOSIM simulators reveal that the MOESIF protocol outperforms all other hardware based coherence protocols in terms of energy consumption and access time. MOESIF protocol on an average offers 94.62%, 88.94%, 88.88% and 4.47% energy saving, and 96.37%, 92.83%, 92.77% and 9.21% access time saving over MI, MESI, MESIF and MOESI protocol respectively for different number of cores/processors.

Keywords: cache coherence; MC/MP cache; energy-efficient coherence protocols

Reference to this paper should be made as follows: G. Patil, N. B. Mallya, B. Raveendran (20xx) ‘MOESIF: A MC/MP Cache Coherence Protocol with Improved Bandwidth Utilization’, *Int. J.* Embedded Systems, Vol. X, No. XX, pp. XXX-XXX.

Biographical notes: Ms. Geeta Patil, is a Lecturer in the Department of Computer Science and Information Systems, BITS Pilani K.K. Birla Goa Campus, Goa, India. She received her Bachelor's degree in Computer Engineering in 2001 from Goa University, Goa, India and Master's degree in 2010 from VTU University, Karnataka, India. She is currently pursuing Ph.D. from BITS Pilani University, BITS Pilani K.K. Birla Goa Campus, Goa. Her research interests are in areas of Computer Architecture, Real time systems and Operating systems.

Ms. Neethu Bal Mallya, was a student of BITS Pilani K.K. Birla Goa Campus, Goa, India for six years. She received the B.E.(Hons.) degree in Electronics and Instrumentation and the M.E. degree in Software Systems in 2013 and 2015, respectively. During her studies, she was part of the Systems Research group in the Department of Computer Science and Information Systems for five years and has worked on multiple research projects in the areas of Energy Efficient Multi-core/Multi-processor Real-time Scheduling and Cache Memory Architectures.

Dr. Biju K. Raveendran is currently serving as Assistant Professor in the Department of Computer Science and Information Systems, BITS Pilani K.K. Birla Goa Campus, Goa, India. He received his Ph.D. degree from BITS Pilani, Pilani Campus, Rajasthan in the year 2009. His research area includes Energy Efficient Multi-core / Many-core Real-time Scheduling, Energy Efficient Memory Architecture for Multi-core / Many-core Embedded Systems, Predictable and Dependable Real-time / Embedded System Design, Big Data Systems etc. He was one of the five recipients of Microsoft Research India Fellowship in the year 2005 for his Ph.D. work. He is a recipient of Microsoft young faculty award in the year 2009. He is actively involved in collaborative projects with industries like Microsoft, Aditya Birla Group etc.

1 Introduction

Energy efficiency is one of the major design issues in uniprocessor, multicore (MC) and multiprocessor (MP) systems. Energy efficiency can be addressed at various levels of design hierarchies such as application level [1], operating systems level [2], circuit level [3] and fabrication technology level [4, 5]. 42% of the system level energy consumption is at operating systems level and architecture level [6]. At operating system level, various energy efficient scheduling algorithms are designed to address energy consumption issue [7-9]. The uniprocessor performance is limited by on-chip power dissipation, saturation in instruction level parallelism and bottlenecks in memory technologies [1]. The advancement in technology along with the modern software requirements force the processor architecture to migrate to MC/MP systems. To reduce memory contention, core/processor (node) in MC/MP system replicates data in private cache and shares data in shared cache/main memory. Data replication in private caches reduces memory access latency and contention of shared bus.

Nodes can modify data locally which may lead to data inconsistency if data modified by one of the node is not reflected in other nodes sharing the same copy of data. Data inconsistency issue in MC/MP systems is addressed by cache coherence protocols. Cache coherence protocols maintain data consistency by associating cache line state information with each cache line. The operations performed to maintain consistency includes broadcasting memory address during read and write operation, invalidating private cache line if required, transferring data to other node/next level memory, sending invalidation and acknowledgement signals etc.

Data invalidation to maintain cache coherence leads to another category of cache miss named as coherence miss or invalidation miss. This results in performance degradation as it consumes additional time and energy for maintaining state information and transferring data and signals across MC/MP interconnection network. Researchers have proposed various cache coherence protocols to address these issues in MC/MP system.

The proposed work - Modified-Owned-Exclusive-Shared-Invalid-Forward (MOESIF) cache coherence protocol- aims at reducing the number of write backs to the next level memory. It also aims at reducing the number of responders to a cache misses when multiple copies of data exist in private cache. This results in reducing time, energy and bandwidth usage.

The remainder of the paper is organized as follows: Section 2 discusses the related work on cache coherence solutions. Some of the existing cache coherence protocols like MI, MESI, MOESI and MESIF are briefed in Section 3. Section 4 discusses the MOESIF protocol. Section 5 describes implementation of cache coherence protocols in CaCoSim simulator. Energy and time modeling of various protocols is discussed in Section 6. Experimental setup and performance comparison is presented in Section 7. Section 8 discusses the conclusion and possible applications of the proposed work.

2 Related Work

Cache coherence is well addressed problem in literature. The solutions are either hardware based [10-12], software based [13, 14] or hybrid [15].

Per Stenstrom surveyed various hardware and software based cache coherence protocols for shared memory MP systems [16]. The shared memory MP systems have three major challenges: memory contention, communication contention and latency time. Hardware based cache coherence protocols are categorized as snoopy based and directory based protocols depending on the strategy employed for maintaining data consistency [16, 17]. In directory based protocols, a shared global directory is used for maintaining state of each memory block. The snoopy based protocols maintain cache consistency with the help of local cache line states which updates itself by snooping the signals over the network. Software based cache coherence protocols maintain cache consistency by using compiler analyzed data. If data can be copied as private by multiple nodes then compiler marks the data as cacheable. When data is read only or it is read / written by a single node, the data is cacheable. If any node writes data, it has to update the next level memory before some other node caches it. Data read/written by multiple nodes is non-cacheable. Since performance and hardware requirements of protocols are different, choice of cache coherence protocols become a major design decision [17-19].

Performance analysis of snoopy based, OS based and non-coherent protocols were explored in [20]. Results obtained imply that OS based cache coherence is very much inefficient in terms of energy consumption and performance. Energy consumption of snoopy based protocols is the least among the three categories of protocol explored in [20].

In [15], Chaves et al. discussed combination of software and hardware based protocols to maintain cache coherence. A part of this hybrid cache coherence protocol is implemented in hardware cache controller and remaining part is handled in software by using microkernel. In this scheme, the bandwidth utilization is improved by using multicasting of messages over uni casting. Latency of read request is reduced by using transition state.

In hardware controlled caches, the cache controller has to maintain states of the cache lines based on read and write operations. The widely used read policies are look through and look aside. The widely used writing mechanisms are write back and write through. In case of write through mechanism, every write operation leads to update of the next level memory whereas write back mechanism updates the next level memory on cache line replacement. Garo et al. [21] combined benefits of write back and write through cache update policies. The write back reduces bus contention and dynamic energy consumption when there are frequent updates to a given cache line. Write through is beneficial when there are few writing operations. The cache follows write through policy when cache line is updated rarely and write back policy when cache line is updated heavily. In [21], Garo et al. used frequency shift register and write back bit for each cache line along with global countdown register to keep track of frequency of write back operations.

Either write invalidate or write update is used by the hardware cache coherence protocols [11] to update other nodes sharing the data. The write invalidate based protocols invalidate shared data from other nodes before writing to the shared copy of data. The write update based protocols write the data and updates the data in other shared copies. Write invalidate based protocols offer less network traffic but suffers from high coherence miss latency [22]. The miss latency can be reduced by using write update protocols. When cache line data is updated very frequently, write update mechanism floods the network. In [22], Kayi et al. proposed an adaptive data forwarding protocol on top of write invalidate protocol for producer – consumer sharing pattern. This protocol uses additional Producer-Consumer Predictor Cache (PCPC) to track the sharing pattern. Write update policy is used for all the applications that follows producer - consumer sharing pattern whereas write invalidate policy is used for all other applications.

Asymmetric cache coherence protocols improve performance of systems where there is asymmetric load distribution among multiple nodes [23]. Node with heavy workload is called primary core and lightly loaded nodes are called secondary cores. Predefined primary core uses write back mechanism to update next level memory and write invalidate mechanism to invalidate other shared copies. Thus primary core performs series of write operations without writing the data back to the next level memory or without updating data in other nodes. Secondary cores use write through mechanism to update next level memory and write update mechanism to update copies of shared data. Thus secondary nodes keep main memory and primary core copy always up to date and reduce cache coherence miss time of primary core. Use of different writing mechanisms based on the workload of the node improves system performance.

This work concentrates on snoopy, write invalidate based cache coherence protocols. Some of the existing cache coherence protocols like MI, MESI, MOESI and MESIF are briefed in Section 3. The MOESI protocol allows dirty sharing of data and reduces number of write backs. The MESIF protocol ensures that only one responder exists for the shared data which reduces the number of responders to 1. To improve the bandwidth utilization further, this paper proposed MOESIF protocol.

3 Cache Coherence Protocols

Invalidation based cache coherence protocols maintain a state for each cache line along with the tag and data. The coherence policy is defined by the finite state machine in each node which changes the state of cache line based on read/write operation. The general description of states in snoopy write-invalidate protocols is given in TABLE I.

State	Valid/Invalid?	Clean/Dirty?
M	Valid	Dirty
O	Valid	Dirty
E	Valid	Clean
S	Valid	Clean/Dirty
F	Valid	Clean
I	Invalid	-

TABLE I. CACHE COHERENCE STATES

Some of the existing invalidation based cache coherence protocols are discussed below.

A. MI Protocol

Modified-Invalid (MI) is the simplest protocol with least hardware complexity among all the coherence protocols for MC/MP systems. In MI, only one node can have valid copy of a data block. The cache line containing this data block is in M state. Read or write miss results in broadcasting invalidation signal across the MC/MP interconnection. If other node contains copy of requested data block, it writes back this data to the next level memory and invalidates its own copy. Writing of cache line data block to the next level happens even when data block is not modified. Thus, MI protocol floods the network with large number of invalidation signals and writes backs of data.

The cache state transitions for a pair of caches (P_I and P_J) for read and write operation in P_I is shown in Table II. The final state of requestor P_I and other cache P_J upon a read or a write operation in P_I is represented as <Cache State in Requestor (P_I) /Cache State in Others (P_J)>. "-" represents that the combination is not possible.

		READ $P_I \rightarrow$		WRITE $P_I \rightarrow$	
		M	I	M	I
SNOOP $\downarrow P_J$	M	R_1 -	R_3 M/I	W_1 -	W_3 M/I
	I	R_2 M/I	R_4 M/I	W_2 M/I	W_4 M/I

TABLE II. CACHE STATE TRANSITIONS IN MI PROTOCOL

B. MESI Protocol

The S state in MESI protocol allows multiple nodes to have the same data block in consistent state with the next level memory. If the cache line is modified, then it is the most up-to-date and the only valid copy available in M state. In case of write hit, the invalidation signal is sent to all other cores only when the cache line is in S state.

Read/write misses are satisfied by transferring data to the cache either from the next level memory or from other nodes. If any node has the requested data in E or S state, the requestor is serviced by the node upon broadcast. If a node contains data with M state then it updates the next level memory with that data. All the cache lines containing the requested data will be in S state for read miss and I state for write miss. The state of requestor's cache line is set to E or S state and M state for read miss and write miss respectively. In read miss the requestor's copy will be in E state only if the data is transferred from next level memory. Presence of multiple shared copies reduces the number of coherence misses in MESI over MI protocol. However, for every state change from M, the data needs to be written to the next level. This result in frequent data writes between node and next level when read and write alternate.

The cache state transitions for a pair of caches (P_I and P_J) for Read and Write operation in P_I is shown in TABLE III.

		READ $P_I \rightarrow$				WRITE $P_I \rightarrow$			
		M	E	S	I	M	E	S	I
SNOOP $\downarrow P_J$	M	R_1 -	R_5 -	R_9 -	R_{13} S/S	W_1 -	W_5 -	W_9 -	W_{13} M/I
	E	R_2 -	R_6 -	R_{10} -	R_{14} S/S	W_2 -	W_6 -	W_{10} -	W_{14} M/I
	S	R_3 -	R_7 -	R_{11} S/S	R_{15} S/S	W_3 -	W_7 -	W_{11} M/I	W_{15} M/I
	I	R_4 M/I	R_8 E/I	R_{12} S/I	R_{16} E/I	W_4 M/I	W_8 M/I	W_{12} M/I	W_{16} M/I

TABLE III. CACHE STATE TRANSITIONS IN MESI PROTOCOL

C. MOESI protocol

The M state in MOESI protocol, allows transfer of data to a read requestor without writing back to the next level memory. The dirty copy sharing between different nodes using O state helps to reduce the frequency of write-backs. Write back in MOESI happens only when the cache line in M or O state is replaced. However, during a cache miss, all the sharers of requested data will respond and the requestor will be serviced by redundant responses. These redundant responses increase the traffic across the network as data transfer takes more bandwidth than signals.

The cache state transitions for a pair of caches (P_I and P_J) for Read and Write operation in P_I is shown in TABLE IV.

		READ $P_I\rightarrow$					WRITE $P_I\rightarrow$				
		M	O	E	S	I	M	O	E	S	I
SNOOP $\downarrow P_J$	M	R_1 -	R_6 -	R_{11} -	R_{16} -	R_{21} S/O	W_1 -	W_6 -	W_{11} -	W_{16} -	W_{21} M/I
	O	R_2 -	R_7 -	R_{12} -	R_{17} S/O	R_{22} S/O	W_2 -	W_7 -	W_{12} -	W_{17} M/I	W_{22} M/I
	E	R_3 -	R_8 -	R_{13} -	R_{18} -	R_{23} S/S	W_3 -	W_8 -	W_{13} -	W_{18} -	W_{23} M/I
	S	R_4 -	R_9 O/S	R_{14} -	R_{19} S/S	R_{24} S/S	W_4 -	W_9 M/I	W_{14} -	W_{19} M/I	W_{24} M/I
	I	R_5 M/I	R_{10} O/I	R_{15} E/I	R_{20} S/I	R_{25} E/I	W_5 M/I	W_{10} M/I	W_{15} M/I	W_{20} M/I	W_{25} M/I

TABLE IV. CACHE STATE TRANSITIONS IN MOESI PROTOCOL

D. MESIF Protocol

The issue of redundant responses from all the sharers in MESI and MOESI is addressed by the F state. MESIF protocol ensures that only the cache line in F state responds to 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 using F state, sharing of dirty data is not allowed in MESIF. Write back happens for all the state change from M state. The cache state transitions for a pair of caches (P_I and P_J) for Read and Write operation in P_I is shown in TABLE V.

		READ $P_I\rightarrow$					WRITE $P_I\rightarrow$				
		M	E	S	I	F	M	E	S	I	F
SNOOP $\downarrow P_J$	M	R_1 -	R_6 -	R_{11} -	R_{16} F/S	R_{21} -	W_1 -	W_6 -	W_{11} -	W_{16} M/I	W_{21} -
	E	R_2 -	R_7 -	R_{12} -	R_{17} F/S	R_{22} -	W_2 -	W_7 -	W_{12} -	W_{17} M/I	W_{22} -
	S	R_3 -	R_8 -	R_{13} S/S	R_{18} F/S	R_{23} F/S	W_3 -	W_8 -	W_{13} M/I	W_{18} M/I	W_{23} M/I
	I	R_4 M/I	R_9 E/I	R_{14} S/I	R_{19} E/I	R_{24} F/I	W_4 M/I	W_9 M/I	W_{14} M/I	W_{19} M/I	W_{24} M/I
	F	R_5 -	R_{10} -	R_{15} S/F	R_{20} F/S	R_{25} -	W_5 -	W_{10} -	W_{15} M/I	W_{20} M/I	W_{25} -

TABLE V. CACHE STATE TRANSITIONS IN MESIF PROTOCOL

4 Proposed Protocol

This paper proposes a cache coherence protocol – MOESIF – to achieve energy efficiency and high performance by optimizing data transfers between caches and with next level memory. This is achieved by reducing the number of write backs to the next level memory and making sure only one responder sends data to the requestor. Introducing O state and F state with MESI states help in achieving this. In MOESI protocol, for read miss or write miss, cache controller of all nodes containing valid data responds to the request. This generates redundant responses. Redundant responses flood the network which results in increased data traffic and thus response time. In MOESIF protocol, only the cache controller of the node either in M, O or F state responds with the requested data. Resulting in reduction of total number of responses thus traffic and response time.

If any node sends read request or write request for the modified data, in MESIF protocol, the data is first written back to L2 cache and the requesting node receives it from L2 cache. In MOESIF protocol, node having modified data forwards dirty copy of data to the requesting node without updating L2 cache. This cache to cache transfer reduces the number of write backs to the next level memory.

MOESIF protocol achieves energy savings and performance improvement over MOESI and MESIF protocol by reducing number of responses and writes backs respectively. Number of write backs of dirty data in MOESIF is further reduced by transferring ownership to the latest requestor over MOESI protocol.

The memory subsystem of energy efficient multi-core embedded processors usually contains private split L1 cache and unified shared L2 cache for the cores of the processor. This paper concentrates on improving L1 cache performance by redesigning coherency protocols. The cache state transitions for a pair of caches (P_I and P_J) for Read and Write operations in P_I is shown in TABLE VI.

		READ $P_I\rightarrow$						WRITE $P_I\rightarrow$					
		M	O	E	S	I	F	M	O	E	S	I	F
SNOOP $\downarrow P_J$	M	R_1 -	R_7 -	R_{13} -	R_{19} -	R_{25} O/S	R_{31} -	W_1 -	W_7 -	W_{13} -	W_{19} -	W_{25} M/I	W_{31} -
	O	R_2 -	R_8 -	R_{14} -	R_{20} S/O	R_{26} O/S	R_{32} -	W_2 -	W_8 -	W_{14} -	W_{20} M/I	W_{26} M/I	W_{32} -
	E	R_3 -	R_9 -	R_{15} -	R_{21} -	R_{27} F/S	R_{33} -	W_3 -	W_9 -	W_{15} -	W_{21} -	W_{27} M/I	W_{33} -
	S	R_4 -	R_{10} O/S	R_{16} -	R_{22} S/S	R_{28} F/S	R_{34} F/S	W_4 -	W_{10} M/I	W_{16} -	W_{22} M/I	W_{28} M/I	W_{34} M/I
	I	R_5 M/I	R_{11} O/I	R_{17} E/I	R_{23} S/I	R_{29} E/I	R_{35} F/I	W_5 M/I	W_{11} M/I	W_{17} M/I	W_{23} M/I	W_{29} M/I	W_{35} M/I
	F	R_6 -	R_{12} -	R_{18} -	R_{24} S/F	R_{30} F/S	R_{36} -	W_6 -	W_{12} -	W_{18} -	W_{24} M/I	W_{30} M/I	W_{36} -

TABLE VI. CACHE STATE TRANSITIONS IN MOESIF PROTOCOL

In MOESIF protocol, the sharers (S state) exist either with an owner (O state) or with a forwarder (F state). If multiple copies of shared data exist then the broadcasted cache miss signal is satisfied either by the owner or by the forwarder. If the cache line is in O state, it and all the copies in S state are the most up to date copy and the next level memory contains stale data. The copy in O state only will act as forwarder which reduces the number of write backs and number of responders.

The replacement of owner cache line in MOESI and MOESIF protocol writes the dirty cache line back to the next level memory. The number of write backs due to replacement of cache line in O state is reduced in MOESIF protocol because the ownership of shared dirty data is transferred to the read requestor serviced by the owner. If forwarder or owner gets replaced, then MOESIF protocol randomly picks one of the sharers as forwarder. This guarantees existence of a single responder in the system.

5 CACOSIM Implementation

The CAche COherence SIMulator (CACOSIM) models various invalidation based snoopy protocols for L1 cache of MC/MP systems. The framework takes trace files generated using Intel's PINTOOL [24] as input along with memory subsystem configuration details. The trace file consists of 40 bit addresses, node number and the operation performed on it. The memory subsystem configuration details include number of nodes, cache size, cache line size and associativity of each node in all levels. LRU replacement policy is followed across all the levels. The implementation of cache coherence protocols in CACOSIM is described below.

A. MI Protocol

As shown in TABLE II, the cache line state will remain unchanged for read hit and write hit (cases R_2 and W_2 respectively). The cache read misses (cases R_3 and R_4), and write misses (cases W_3 and W_4) broadcast the request through the bus. If other node has the requested data (cases R_3 and W_3), it will perform write back operation. Upon receiving write back acknowledgement from L2, it will invalidate its copy. For all cache misses the requestor will receive the data from L2 and update its cache line to M state.
Fig. 1(a) shows the access function of the requestor node in MI implementation of CACOSIM. The cache hits and misses are represented with → and ---> respectively. The cases which results in broadcast are marked as R_J or W_J in red color. Fig. 1(b) shows the response of remaining N-1 nodes of MI implementation in CACOSIM.

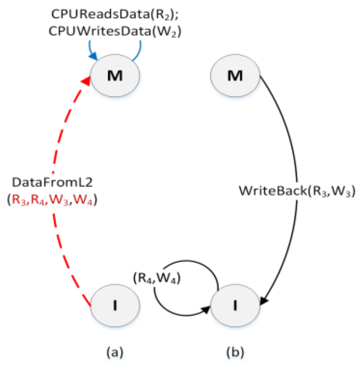
CPUReadsData(R_2); CPUWritesData(W_2)
M
M
DataFromL2 (R_3,R_4,W_3,W_4)
WriteBack(R_3,W_3)
(R_4,W_4)
I
I
(a)
(b)

Fig. 1 (a) Access and (b) Snoop Function of MI CACOSIM Implementation

B. MESI Protocol

As shown in TABLE III, the cache line states will remain unchanged for read hit (cases R_4, R_8, R_{11} and R_{12}) and write hit (case W_4). The cache line state updates to M for write hit (cases W_8, W_{11} and W_{12}). The cache read misses (cases R_{13}, R_{14}, R_{15}, and R_{16}), write misses (cases W_{13}, W_{14}, W_{15} and W_{16}) and the write hits (cases W_{11} and W_{12}) in S state will broadcast the request.
In case of write hit in S state (cases W_{11} and W_{12}), the sharer invalidate its copy and invalidation acknowledgement is sent to the requestor. Upon receiving acknowledgements from N-1 nodes, requestor updates its cache line state to M.
In case of cache miss, if other nodes have the requested data in S or E state, those nodes will transfer the data to the requesting node. For read request (cases R_{14} and R_{15}), both responder and requestor updates cache line state to S. For write request (cases W_{14} and W_{15}), the responder invalidates its copy and upon receiving acknowledgements from N-1 nodes, the requestor updates its cache line state to M.
If any node has the requested data in M state (cases R_{13} and W_{13}), then that node updates L2. Upon receiving write back acknowledgement from L2, responder invalidates its copy for W_{13} whereas responder changes the state to S state for R_{13}. The requestor receives the cache line from L2 and changes its cache line state to M for W_{13} and to S for R_{13}.
For R_{16} and W_{16} where no node has the requested data, the requestor receives the requested data from L2 and sets the cache line state to E and M for read request and write request respectively.
When a cache line in M state is the replacement victim for a cache miss, the cache line data is written back to L2.
Fig. 2 (a) and (b) show the access and snoop function in MESI implementation of CACOSIM.

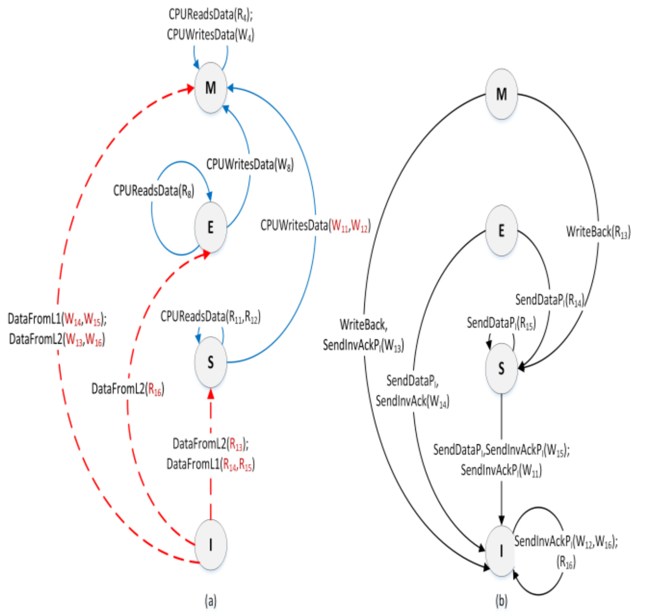
CPUReadsData(R_4); CPUWritesData(W_4)
M
CPUWritesData(W_8)
CPUReadsData(R_8)
E
CPUWritesData(W_{11},W_{12})
DataFromL1(W_{14},W_{15}); DataFromL2(W_{13},W_{16})
CPUReadsData(R_{11},R_{12})
S
DataFromL2(R_{16})
DataFromL2(R_{13}); DataFromL1(R_{14},R_{15})
I
(a)
M
E
WriteBack(R_{13})
SendDataP$_I$(R_{14})
SendDataP$_I$(R_{15})
WriteBack, SendInvAckP$_I$(W_{13})
S
SendDataP$_I$, SendInvAck(W_{14})
SendDataP$_I$,SendInvAckP$_I$(W_{15}); SendInvAckP$_I$(W_{11})
I
SendInvAckP$_I$(W_{12},W_{16}); (R_{16})
(b)

Fig. 2 (a) Access and (b) Snoop Function of MESI CACOSIM Implementation

C. MOESI Protocol

As shown in TABLE IV, cache line state will remain unchanged for read hit (cases R_5, R_9, R_{10}, R_{15}, R_{17}, R_{19}, R_{20}) and write hit (case W_5). The cache read misses (cases R_{21}, R_{22}, R_{23}, R_{24} and R_{25}), write misses (cases W_{21}, W_{22}, W_{23}, W_{24} and W_{25}) and the write hits (cases W_9, W_{10}, W_{17}, W_{19}, and W_{20}) in O or S state broadcasts the request.
When write request is broadcasted, the cache controllers containing shared data copy invalidates it and sends acknowledgement to the requestor. Upon receiving acknowledgements from N-1 nodes, requestor updates its cache line state to M.
In case of cache miss, if other nodes have the requested data, those nodes perform L1 to L1 transfer to the requestor. If the read request is served by the node in dirty state (M state or O state), the final state of the responder will be O. If the read request is served by the node with clean state (E state or S state), the final state of the responder will be S.

For write request, the responder invalidates itself and sends invalidation acknowledgement to the requestor. For read request, the requestor changes its state to S. For write request, after receiving N-1 invalidation acknowledgements the requestor changes its state to M.

For R_{25} and W_{25} where no node has the requested data, the requestor will receive the data from L2 and sets the cache line state to E and M for read and write respectively.

In this protocol, the write back happens only on replacement of cache line in M state or O state.

Fig. 3(a) and 3(b) show the access and snoop function in MOESI implementation of CACOSIM.

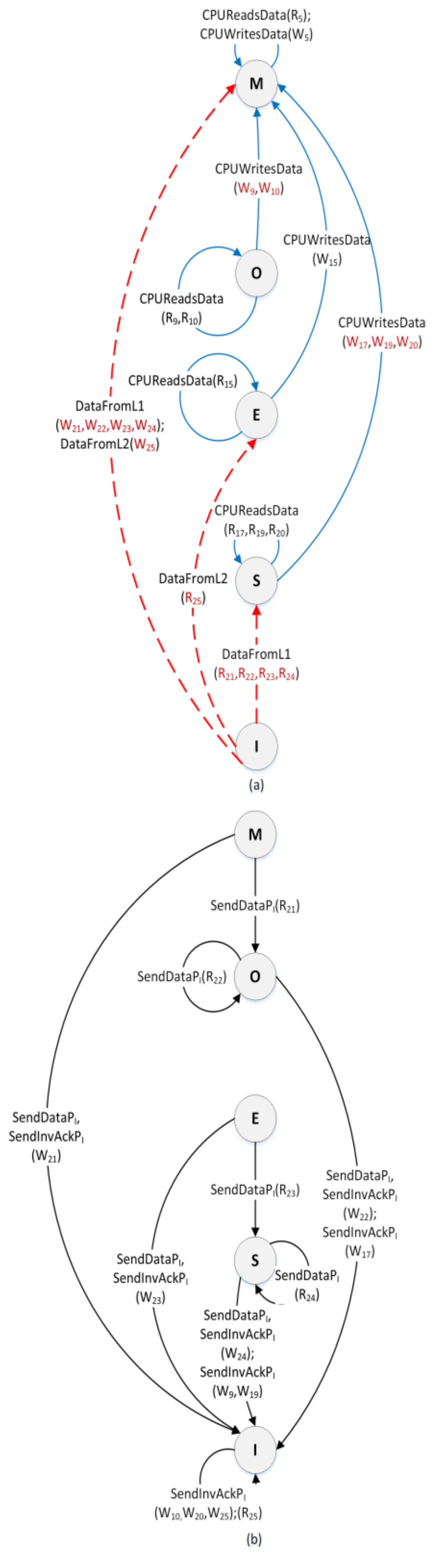
CPUReadsData(R_5); CPUWritesData(W_5)
M
CPUWritesData (W_9,W_{10})
CPUWritesData (W_{15})
O
CPUReadsData (R_9,R_{10})
CPUWritesData (W_{17},W_{19},W_{20})
CPUReadsData(R_{15})
E
DataFromL1 (W_{21},W_{22},W_{23},W_{24}); DataFromL2(W_{25})
CPUReadsData (R_{17},R_{19},R_{20})
DataFromL2 (R_{25})
S
DataFromL1 (R_{21},R_{22},R_{23},R_{24})
I
(a)
M
SendDataP$_i$(R_{21})
SendDataP$_i$(R_{22})
O
E
SendDataP$_i$, SendInvAckP$_i$ (W_{21})
SendDataP$_i$(R_{23})
SendDataP$_i$, SendInvAckP$_i$ (W_{22}); SendInvAckP$_i$ (W_{17})
SendDataP$_i$, SendInvAckP$_i$ (W_{23})
S
SendDataP$_i$ (R_{24})
SendDataP$_i$, SendInvAckP$_i$ (W_{24}); SendInvAckP$_i$ (W_9,W_{19})
I
SendInvAckP$_i$ (W_{10},W_{20},W_{25});(R_{25})
(b)

Fig. 3 (a) Access and (b) Snoop Function of MOESI CACOSIM Implementation

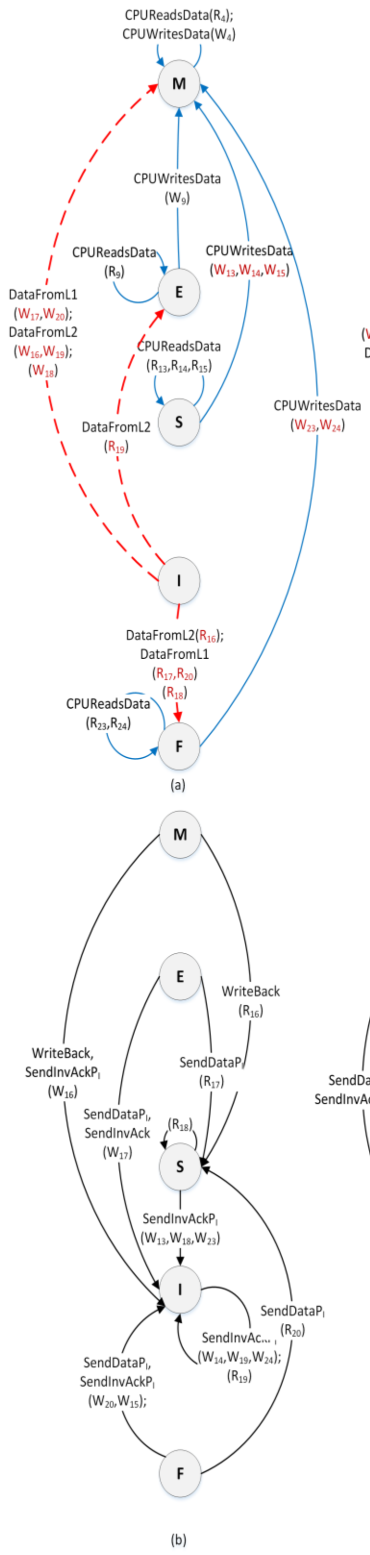
CPUReadsData(R_4); CPUWritesData(W_4)
M
CPUWritesData (W_9)
CPUReadsData (R_9)
CPUWritesData (W_{13},W_{14},W_{15})
E
DataFromL1 (W_{17},W_{20}); DataFromL2 (W_{16},W_{19}); (W_{18})
CPUReadsData (R_{13},R_{14},R_{15})
DataFromL2 (R_{19})
S
CPUWritesData (W_{23},W_{24})
I
DataFromL2(R_{16}); DataFromL1 (R_{17},R_{20}) (R_{18})
CPUReadsData (R_{23},R_{24})
F
(a)
M
E
WriteBack (R_{16})
WriteBack, SendInvAckP$_i$ (W_{16})
SendDataP$_i$ (R_{17})
SendDataP$_i$, SendInvAck (W_{17})
(R_{18})
S
SendInvAckP$_i$ (W_{13},W_{18},W_{23})
I
SendDataP$_i$ (R_{20})
SendInvAc... (W_{14},W_{19},W_{24}); (R_{19})
SendDataP$_i$, SendInvAckP$_i$ (W_{20},W_{15});
F
(b)

Fig. 4 (a) Access and (b) Snoop Function of MESIF CACOSIM Implementation

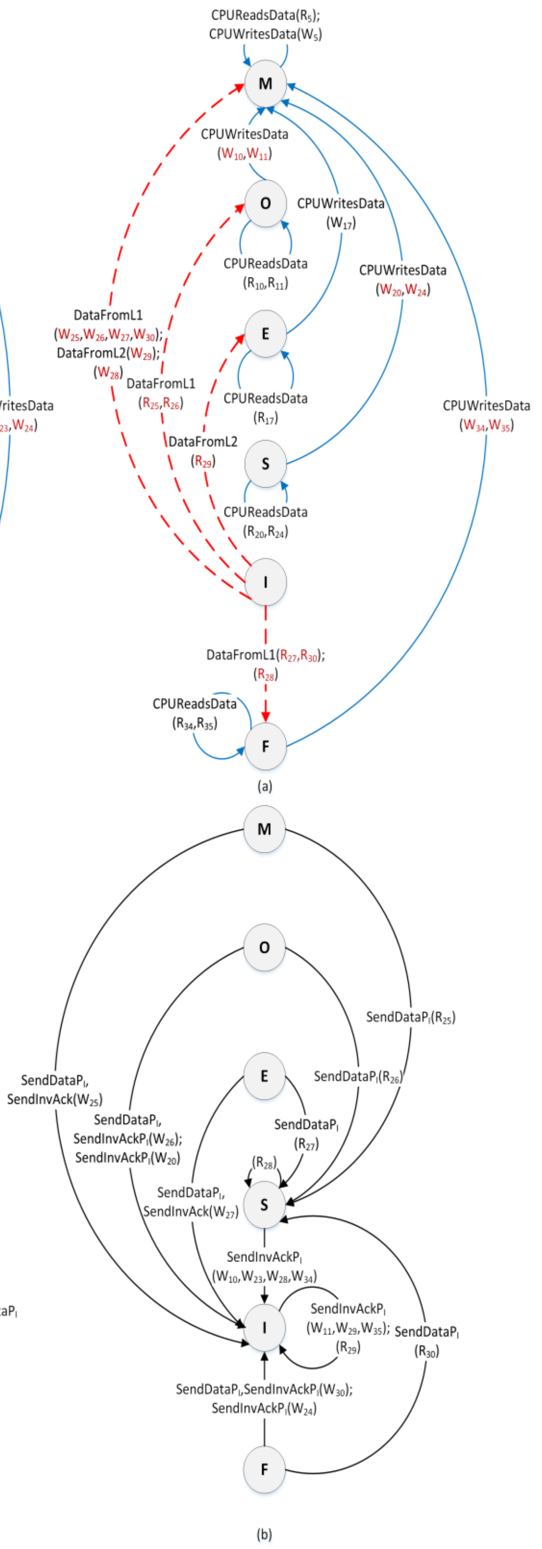
CPUReadsData(R_5); CPUWritesData(W_5)
M
CPUWritesData (W_{10},W_{11})
O
CPUWritesData (W_{17})
CPUReadsData (R_{10},R_{11})
CPUWritesData (W_{20},W_{24})
DataFromL1 (W_{25},W_{26},W_{27},W_{30}); DataFromL2(W_{29}); (W_{28})
DataFromL1 (R_{25},R_{26})
E
CPUReadsData (R_{17})
CPUWritesData (W_{34},W_{35})
DataFromL2 (R_{29})
S
CPUReadsData (R_{20},R_{24})
I
DataFromL1(R_{27},R_{30}); (R_{28})
CPUReadsData (R_{34},R_{35})
F
(a)
M
O
SendDataP$_i$(R_{25})
E
SendDataP$_i$(R_{26})
SendDataP$_i$, SendInvAck(W_{25})
SendDataP$_i$, SendInvAckP$_i$(W_{26}); SendInvAckP$_i$(W_{20})
SendDataP$_i$ (R_{27})
(R_{28})
S
SendDataP$_i$, SendInvAck(W_{27})
SendInvAckP$_i$ (W_{10},W_{23},W_{28},W_{34})
SendInvAckP$_i$ (W_{11},W_{29},W_{35}); (R_{29})
SendDataP$_i$ (R_{30})
I
SendDataP$_i$,SendInvAckP$_i$(W_{30}); SendInvAckP$_i$(W_{24})
F
(b)

Fig. 5 (a) Access and (b) Snoop Function of MOESIF CACOSIM Implementation

D. MESIF Protocol

As shown in TABLE V, cache line state will remain unchanged for read hit (cases R_4, R_9, R_{13}, R_{14}, R_{15}, R_{23}, R_{24}) and write hits (case W_4). The cache line state updates to M for write hits (cases W_9, W_{13}, W_{14}, W_{15}, W_{23}, W_{24}). The cache read misses (cases R_{16}, R_{17}, R_{18}, R_{19} and R_{20}), write misses (cases W_{16}, W_{17}, W_{18}, W_{19} and W_{20}) and write hits (cases W_{13}, W_{14}, W_{15}, W_{23} and W_{24}) in S or F state broadcasts the request.

When write request is issued by the node for cache line in S state or F state, the corresponding cache controller invalidates the shared copies and the invalidation acknowledgement is sent to the requestor. Upon receiving acknowledgements from N-1 nodes, requestor updates its cache line state to M.

If any node has the requested data in M state, that node writes it back to L2. Upon receiving write back acknowledgement from L2, it invalidates its copy for W_{16} whereas it changes to S state for R_{16}. The requestor receives the data from L2 and changes its cache line state to M for W_{16} and F for R_{16}.

If other nodes have the copy of requested data, the node with cache line in F state performs L1 to L1 transfer. For the read request (cases R_{17}, R_{18} and R_{20}), the responder and requestor changes their state to S and F respectively. For the write request, the responder invalidates itself and sends invalidation acknowledgement to the requestor. The requestor changes its state to M after receiving N-1 invalidation acknowledgements.

For R_{19} and W_{19} where no node has the requested data, the requestor receives the requested data from L2 and sets the cache line state to E and M for read request and write request respectively.

Fig. 4 (a) and (b) show the access and snoop function in MESIF implementation of CACOSIM.

E. MOESIF Protocol

As shown in TABLE VI, the cache line state remains unchanged for read hits (cases R_5, R_{10}, R_{11}, R_{17}, R_{20}, R_{22}, R_{23}, R_{24}, R_{34} and R_{35}) and write hit (case W_5). The cache line state updates to M state for write hits (cases W_{10}, W_{11}, W_{17}, W_{20}, W_{22}, W_{23}, W_{24}, W_{34} and W_{35}). The cache read misses (cases R_{25}, R_{26}, R_{27}, R_{28}, R_{29} and R_{30}), write misses (cases W_{25}, W_{26}, W_{27}, W_{28}, W_{29} and W_{30}) and write hits (cases W_{10}, W_{11}, W_{20}, W_{22}, W_{23}, W_{24}, W_{34}, and W_{35}) in O, S or F state broadcasts the request.

When write request is broadcasted, the cache controllers containing shared data copy invalidates it and sends acknowledgement to the requestor. Upon receiving acknowledgements from N-1 nodes, requestor updates its cache line state to M.

If any other node has the requested data in M state, that node transfers the data to the requestor and invalidates the copy for W_{25} whereas it changes its state to S for R_{25}.

If any node has the requested data in cache line with E, O or F state, that node performs L1 to L1 transfer of data. For read request (cases R_{26}, R_{27} and R_{30}), the responder changes its state to S. If the node with O state forwards the data, the requestor changes its state to O otherwise the requestor changes its state to F. For write request (cases W_{26}, W_{27} and W_{30}), the responder invalidates itself and sends invalidation acknowledgement to the requestor. The requestor changes its state to M after receiving N-1 invalidation acknowledgements. For R_{29} and W_{29} where no node has the requested data, the requestor will receive the data from L2 and sets the cache line state to E and M for read and write operation respectively.

The major modification implemented in MOESIF protocol in comparison with MOESI and MESIF protocol are read miss cases (R_{25}, R_{26}, R_{27} and R_{28}) and case(R_{25}) respectively. In MOESIF protocol read miss cases (R_{25} and R_{26}) transfers the ownership of the data to the requestor node and read miss cases (R_{27} and R_{28}) updates the requestor cache line state to F. Whereas, in these miss scenarios , the requestor node is in shared state in case of MOESI. Transfer of ownership to the latest requestor reduces the number of write backs.

Read miss case (R_{25}) in MOESIF differs from read miss case (R_{16}) in MESIF protocol. In MESIF protocol in case of read miss case (R_{16}), write back is done and updated data is written to next level memory. The number of write backs is reduced in MOESIF protocol by sharing the dirty data.

Fig. 5 (a) and (b) show the access and snoop function in MOESIF implementation of CACOSIM. To maintain cache in coherent state MI and MESI requires 1 and 2 bits per cache line respectively, whereas MOESI, MESIF and MOESIF require 3 bits per cache line as shown in TABLE VII. Although MOESIF protocol has an overhead of additional bits per cache line to maintain cache coherency state over MI and MESI protocol, it reduces network traffic by sharing of dirty data among nodes. MOESIF protocol achieves energy savings and performance improvement over MESIF and MOESI protocol without additional hardware overhead and communication signal requirement.

Protocol	Number of Bits
MI	1
MESI	2
MOESI	3
MESIF	3
MOESIF	3

TABLE VII. NUMBER OF BITS TO MAINTAIN COHERENCE STATUS

6 Energy and Time Model

The cache components used in the evaluation of energy consumption and access time based on CACTI 5.3 simulator [25] are shown in TABLE VIII [26].

Cache Component	**Energy Components**		**Total Energy**	**Time Components**	
	Data Side	Tag Side		Data Side	Tag Side
Decoder	E_{ds_dec}	E_{ts_dec}	$E_{dec} = E_{ds\_dec} + E_{ts\_dec}$	T_{ds_dec}	T_{ts_dec}
Bit lines	$E_{ds\_bline}^*$	$E_{ts\_bline}^*$	$E_x = E_{ds\_bline} + E_{ts\_bline} + E_{ds\_sa} + E_{ts\_sa} + E_{ds\_pre} + E_{ts\_pre} + E_{ts\_cmp}$	T_{ds_bline}	T_{ts_bline}
Sense Amplifier	$E_{ds\_sa}^*$	$E_{ts\_sa}^*$		T_{ds_sa}	T_{ts_sa}
Pre-charge Circuit	$E_{ds\_pre}^*$	$E_{ts\_pre}^*$		T_{ds_pre}	T_{ts_pre}
Comparator	-	$E_{ts\_cmp}^*$		-	T_{ts_cmp}
Output Driver	$E_{ds\_op\_drvr}$	$E_{ts\_op\_drvr}$	$E_{op\_drvr} = E_{ds\_op\_drvr} + E_{ts\_op\_drvr}$	$T_{ds\_op\_drvr}$	$T_{ts\_op\_drvr}$

**- Energy dissipation for the components is for 1-way*

TABLE VIII. CACHE COMPONENTS FOR ENERGY AND TIME MODELING

The per access energy and cycle time of L1 cache is calculated as:

$$E_{access} = E_{dec} + \text{associativity} * E_x$$
$$T_{1cycle} = \max\{\max(T_{ds\_dec}, T_{ts\_dec}), \max[(T_{ds\_bline} + T_{ds\_sa}), (T_{ts\_bline} + T_{ts\_sa} + T_{ts\_cmp} + T_{ts\_opdrvr})], (T_{ds\_opdrvr})\}$$

The processor energy and time for reading and writing cache are:

$$E_{PRead} = E_{op\_drvr} \qquad T_{PRead} = 3 * T_{1cycle}$$
$$E_{PWrite} = 2 * E_{PRead} \qquad T_{PWrite} = 4 * T_{1cycle}$$

E_{Inv}, $E_{Tx\text{-}L1toL1}$, $E_{Tx\text{-}L1toL2}$ and $E_{Tx\text{-}L2toL1}$ represent the energy consumed for cache line invalidation and sending acknowledgements, L1 to L1 transfer, L1 to L2 transfer and L2 to L1 transfer respectively. T_B, T_{AckAll}, T_{L1Read}, $T_{L1Write}$, T_{L2Read}, $T_{L2Write}$, $T_{Tx\text{-}L1toL1}$, $T_{Tx\text{-}L1toL2}$ and $T_{Tx\text{-}L2toL1}$ represent the time for broadcasting address, receiving all the acknowledgements, reading L1, writing L1, reading L2, writing L2, L1 to L1 transfer, L1 to L2 transfer and L2 to L1 transfer respectively. The transfer time among L1 and L2 caches is calculated as:

$$T_{x11} = T_{L1Read} + T_{Tx\text{-}L1toL1} + T_{L1Write}$$
$$T_{x21} = T_{L2Read} + T_{Tx\text{-}L2toL1} + T_{L1Write}$$
$$T_{x12} = T_{L1Read} + T_{Tx\text{-}L1toL2} + T_{L2Write}$$

When replacement results in write back of dirty data, the additional energy and access time is added up in the energy and time calculation respectively. The values considered for energy and time parameters are shown in TABLE IX. The energy and time model for cache read operation and write operation in an N-node system is shown in TABLE X.

Energy Component	Value
E_{Inv}	$1*E_{access}$
$E_{Tx\text{-}L1toL1}$	5*Eaccess
$E_{Tx\text{-}L1toL2}$	100*Eaccess
$E_{Tx\text{-}L2toL1}$	100*Eaccess

Time Component	Value
T_B	$3*T_{1cycle}$
T_{AckAll}	$5*T_{1cycle}$
T_{x11}	$15*T_{1cycle}$
T_{x12}	$300*T_{1cycle}$
T_{x21}	$300*T_{1cycle}$

TABLE IX. ENERGY AND TIME PARAMETERS

7 Experimental Setup and Evaluation

A. Experimental Setup

This work uses CACOSIM simulator, a MC/MP simulation framework to model the L1 cache coherence protocols as discussed in Section 5. The simulator estimates the energy consumption, access time, cache miss rate, L1 to L1 transfers, write backs, L2 to L1 transfers, invalidations and invalidation acknowledgements. The CACOSIM simulations are performed on a high performance computing facility with 96 cores connected as 6 tightly coupled nodes. SPLASH-2 benchmark programs [27] ran on the same high performance infrastructure to evaluate the performance of various cache coherence protocols.
The experimentation uses cache size, cache line size and associativity as 4KB to 32KB, 8B to 32B and direct mapped to 16-way respectively. CACTI 5.3 time and power estimations for the given cache configuration in 32nm technology is imported to the framework for performance evaluation

B. Experimental Evaluation

1) Energy Consumption

Fig. 6 shows the energy saving of MESI, MESIF, MOESI and MOESIF cache coherence protocols over MI protocol for a 4 node system with 4-way set-associative cache of 32KB and 32B line size.

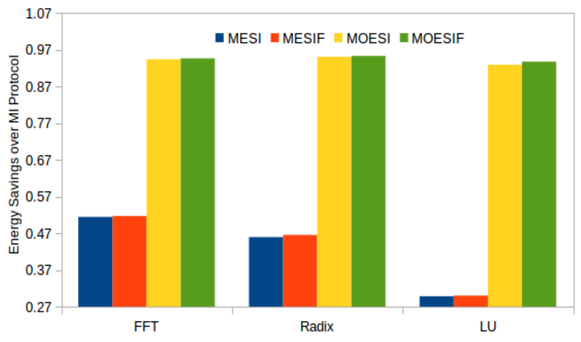

Fig. 6 Energy Savings over MI protocol for different SPLASH-2 benchmark programs

Energy consumption is directly proportional to miss rate and number of coherency signals exchanged. Miss rate of MI is 12.14% higher than other protocol. Per access write backs in MI, MESI, MESIF, MOESI and MOESIF protocol is 27.04%, 12.64%, 12.64%, 0.09% and 0.08% respectively.

Cache Operation		Energy	Time
Read HIT	-	$E_{access} + E_{PRead}$	T_{PRead}
Read MISS	*When no L1 copies exist*	$N*E_{access} + E_{Tx\text{-}L2toL1} + E_{PRead}$	$T_B + T_{x21} + T_{PRead}$
	When k-L1 copies exist (1<=k<=N-1)	$N*E_{access} + k*E_{Tx\text{-}L1toL1} + E_{PRead}$	$T_B + k*T_{x11} + T_{PRead}$
	When a single modified copy exist in a L1	$N*E_{access} + E_{Tx\text{-}L1toL2} + E_{Tx\text{-}L2toL1} + E_{PRead}$	$T_B + T_{x12} + T_{x21} + T_{PRead}$
Write HIT	*When no other L1 copies exist*	$E_{access} + E_{PWrite}$	T_{PWrite}
	When other L1 copies exist	$N*E_{access} + E_{Inv} + E_{PWrite}$	$T_B + T_{AckAll} + T_{PWrite}$
Write MISS	*When no L1 copies exist*	$N*E_{access} + E_{Tx\text{-}L2toL1} + E_{Inv} + E_{PWrite}$	$T_B + \max(T_{x21}, T_{AckAll}) + T_{PWrite}$
	When k-L1 copies exist (1<=k<=N-1)	$N*E_{access} + k*E_{Tx\text{-}L1toL1} + E_{Inv} + E_{PWrite}$	$T_B + \max(k*T_{x11}, T_{AckAll}) + T_{PWrite}$
	When a single modified copy exist in a L1	$N*E_{access} + E_{Tx\text{-}L1toL2} + E_{Tx\text{-}L2toL1} + E_{Inv} + E_{PWrite}$	$T_B + \max(T_{x12} + T_{x21}, T_{AckAll}) + T_{PWrite}$

TABLE X. ENERGY AND TIME MODELING FOR CACHE OPERATIONS

Per access invalidations in MI is 27.27% and all in other protocols it is 15.07%. Per access number of signals from next level memory in MI, MESI, MESIF, MOESI and MOESIF protocol is 27.54%, 12.82%, 12.82%, 0.27% and 0.27% respectively. Due to high miss rate and higher number of signal transfer in MI protocol, result shows that MI protocol consume more energy compared to other protocols. For various benchmark programs tested, the average energy savings of MESI, MESIF, MOESI and MOESIF protocols over MI protocol is 42.40%, 42.75%, 94.10% and 94.55% respectively. Energy savings of MOESIF protocol is the highest as it allows sharing of dirty data. Additionally MOESIF protocol ensures existence of only one forwarder in the system which reduces the number of data traffic in the network.

Fig. 7 shows the total energy consumption of all cache coherence protocols for different number of nodes using FFT benchmark program. Each node has a 4-way set-associative L1 cache of 32KB cache size and 32B line size. Energy consumption of MI is the highest in all cases. MESI and MESIF have almost comparable energy consumption when there are two nodes. For larger number of nodes, MESIF consumes less energy than MESI with a single responder for forwarded requests. With the O state, energy consumption is further reduced in MOESI and MOESIF. The F state offers the same trend for MOESIF as observed for MESIF. It is also evident that energy consumption increases exponentially with increase in number of nodes. However, the increase is substantially small for MOESI and MOESIF protocols over other protocols because of dirty sharing and reduced number of write backs.

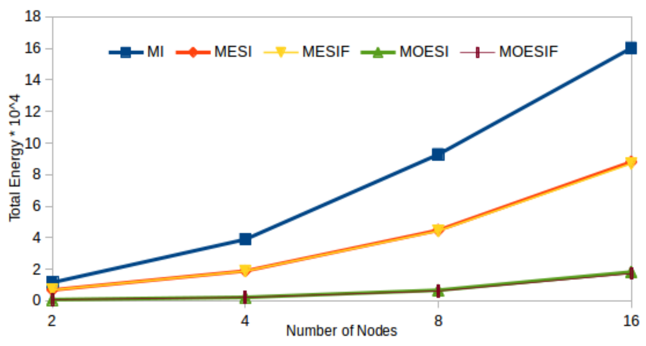

Fig. 7 Total Energy of Coherence Protocols for different number of Nodes

Fig. 8 shows the energy saving of MOESI and MOESIF protocols over MESI protocol. It is observed that the saving reduces with increase in number of nodes. It is also observed that MOESIF outperforms MOESI with increase in number of nodes. The saving of MOESIF over MOESI is 0.1% for 2 nodes and 1.154% for 16 nodes.

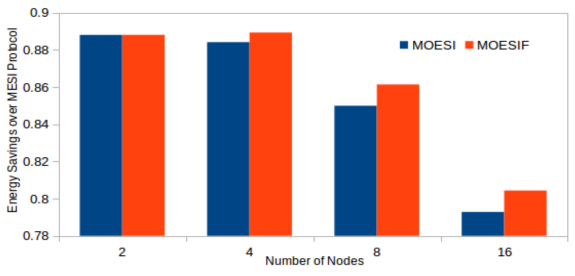

Fig. 8 Energy Savings over MESI protocol for different number of Nodes

Fig. 9 shows the total energy consumption of all cache coherence protocols for a 4 node system with 4-way set-associative cache of 32B line size and varying cache size of 4K to 32K. Dirty copy of data is shared in MOESI and MOESIF protocol, whereas MESI protocol do not share dirty copy of data. From the analysis, it is observed that the savings of MOESI protocol over MESI increases from 87.30% for 4K to 88.42% for 32K L1 cache. The saving of MOESIF protocol over MESI increases from 87.82% for 4K to 88.94% for 32K. The energy savings of MOESIF protocol over MOESI increased from 4.02% in 4K cache to 4.47% in 32K L1 cache.

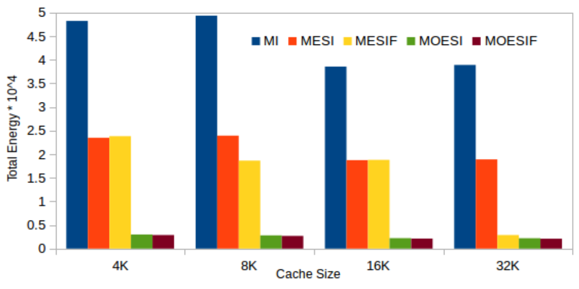

Fig. 9 Total Energy of Coherence Protocols for different Cache Sizes

It is evident from Fig. 10 that with increase in cache line size, the total energy consumption reduces due to reduction in compulsory misses. Fig. 11 shows the energy saving of MOESI and MOESIF protocol over MESI protocol for varying cache line sizes. It is observed that the savings of MOESIF over MOESI increases with increase in cache line size and are 0.19%, 0.41% and 0.52% for 8B, 16B and 32B respectively.

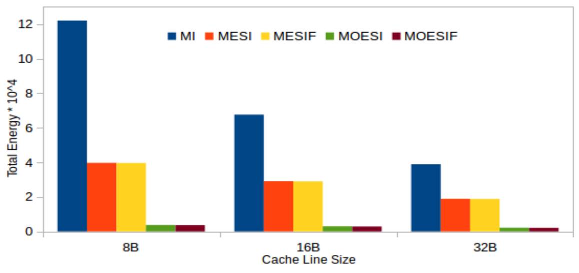

Fig. 10 Total Energy of Coherence Protocols for different Cache Line Sizes

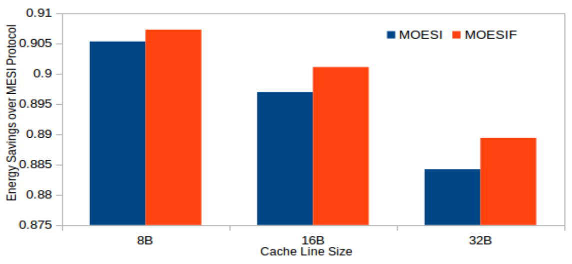

Fig. 11 Energy Savings over MESI Protocol for different Cache Line Sizes

2) Access Time

Fig. 12 shows access time of MI, MESI, MESIF, MOESI and MOESIF cache coherence protocols for a 4 node system with 4-way set-associative cache of 32KB and 32B line size. It is observed that irrespective of benchmark programs, the access time follows the same trend:

MI > MESI > MESIF > MOESI > MOESIF

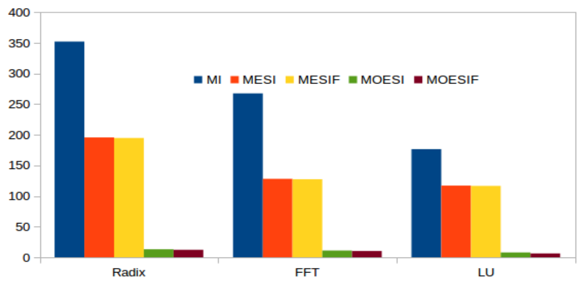

Fig. 12 Access Time of Cache Coherence Protocols for different SPLASH-2 benchmark programs

Fig. 13 shows that irrespective of benchmark programs in use MOESIF always outperforms MOESI in terms of access time. MOESI and MOESIF gives on an average 93.03% and 93.79% over MESI protocol for all SPLASH-2 benchmarks programs.

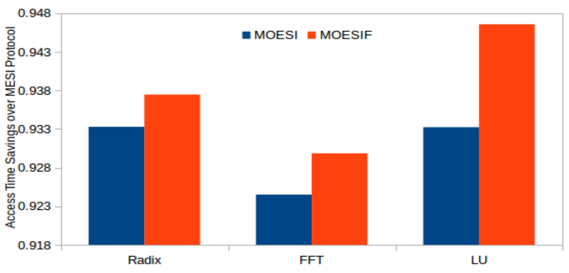

Fig. 13 Access Time Savings over MESI protocol for different SPLASH-2 benchmark programs

Fig. 14 shows the access time with different number of nodes with 4-way set-associative cache of 32KB cache size and 32B line size. As expected, for all protocols with increase in number of nodes, the access time reduces exponentially.

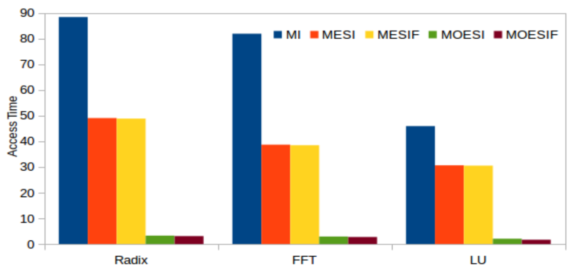

Fig. 14 Access Time of Coherence protocols for different number of Nodes

3) Number of Write backs and Transfers from L2

Fig. 15 shows the reduction in number of write backs in MESI, MESIF, MOESI and MOESIF protocols as compared to MI protocol. Write backs happen when a cache line in M or O state is getting replaced or when a block in M state is accessed on a miss in other node. With MOESI or MOESIF where dirty data can be shared, the number of write backs is reduced by 99.64% as compared to MI.
The percentage reduction in transfer of data from L2 to L1 also follows the same trend. Analysis shows that the transfer from L2 is on an average 50.28% for MESI and MESIF protocols. In MOESI and MOESIF protocols, the average reduction in L2 to L1 transfers is 98.89% because of dirty sharing among the nodes.

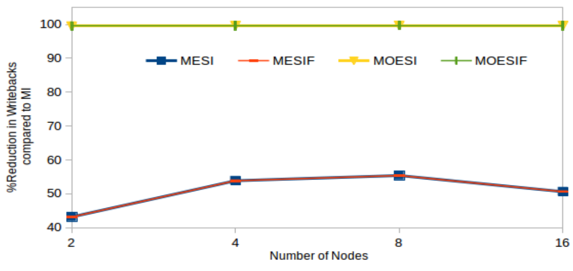

Fig. 15 Percentage reduction in Write backs for different protocols

4) Number of Responders per L1 to L1 transfer

Fig. 16 shows the number of responders per L1 to L1 transfer in MESI, MOESI, MESIF and MOESIF cache coherence protocols. In MESI and MOESI, the number of responders increases with increase in number of nodes. Whereas in MESIF and MOESIF, there is only one responder for an L1 to L1 transfer irrespective of the number of nodes in the system.

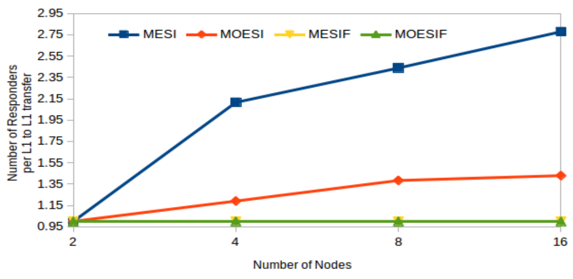

Fig. 16 Number of responders per L1 to L1 transfers for different protocols

5) Number of Invalidations

Invalidations are sent to all the N-1 nodes in the system and the writing node receives acknowledgements from all other nodes irrespective of whether the node contains the cache line or not. Analysis shows that MI protocol has the highest number of invalidations as compared to other protocols. The number of invalidation signals increases as the number of nodes increase and the count is same for MESI, MESIF, MOESI and MOESIF protocols.

8 Conclusion

Cache coherence protocols achieve data consistency at the cost of performance degradation with respect to time and energy. The additional overhead can be minimized by optimizing the usage of interconnection bandwidth. The paper proposed MOESIF protocol which improves the off chip bandwidth by reducing write backs to next level memory and the on chip bandwidth by reducing the number of responders to a cache miss when multiple copies of data exists in private L1 caches. Hit rate of MOESIF protocol is 12.14% higher than the hit rate of MI protocol. Per access number of write backs, invalidations and data transfer rate from next level memory is higher in MI protocol as compared to MOESIF protocol by 27.32%, 12.20% and 27.27% respectively. Experimental evaluation shows that MOESIF offers 92.64% energy saving and 96.69% access time saving over MI protocol. Number of write backs and data transfer rate from next level memory is higher in MESI and MESIF protocol by 12.55%, in comparison with MOESIF protocol. MOESIF protocol offers 86.08% and 85.97% energy savings and 96.69% and 92.99% access time savings over MESI and MESIF protocol respectively. MESIF protocol outperforms MESI protocol due to reduction in number of responders. Hit rate, number of invalidations and data transfers from L2 cache of MOESIF protocol is comparable to MOESI protocol. MOESIF offers 4.42% energy savings and 7.08% access time savings over MOESI protocol. Marginal improvement of MOESIF protocol over MOESI protocol is due to reduction in number of responders.

References

[1] S. Benedict, R.S. Rejitha, C. Preethi, C. B.Bright; W.S. Judyfer, “Energy analysis of code regions of HPC applications using EnergyAnalyzer tool” , Int. J. of Computational Science and Engineering, 2017 Vol.14, No.3, pp.267 – 278

[2] K. S. Hasan, J. K. Antonio, S. Radhakrishnan, “A model-driven approach for predicting and analysing the execution efficiency of multi-core processing” , Int. J. of Computational Science and Engineering, 2017 Vol.14, No.2, pp.105 – 125

[3] H. Sriraman and P. Venkatasubbu, “On the field design bug tolerance on a multi-core processor using FPGA” , Int. J. of High Performance Computing and Networking, 2017 Vol.10, No.1/2, pp.34 – 43

[4] G. Jheng, D. Duh, C. Lai, “Real-time reconfigurable cache for low-power embedded systems” , Int. J. of Embedded Systems, 2010 Vol.4, No.3/4, pp.235 – 247

[5] J. Yan, W. Zhang, “Priority L2 cache design for time predictability” , Int. J. of Embedded Systems, 2016 Vol.8, No.5/6, pp.427 – 439

[6] O. S. Unsal, I. Koren, C. Mani Krishna and C. A. Moritz , The Minimax Cache: An Energy-Efficient Framework for Media Processors, In proceedings 8th International Symposium on High-Performance Computer Architecture, pp. 131 – 140, 2002.

[7] J. Wu, “IASA: an energy-efficient scheduling algorithm for real time task with lock-free objects”, Int. J. of Embedded Systems, 2016 Vol.8, No.5/6, pp.504 – 508.

[8] M. Digalwar, P. Gahukar, B. Raveendran and S. Mohan, “Energy-efficient real time scheduling algorithm for mixed task set on multi-core processors”, International Journal of Embedded Systems (in press).

[9] S. Gawali and B. Raveendran, “DPVFS: a dynamic procrastination cum DVFS scheduler for multicore hard real time systems” , International Journal of Embedded Systems (in press).

[10] 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.

[11] 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.

[12] 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.

[13] 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 (PACT), Oct 2011.

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

[15] T. M. Chaves, E.A. Carara and F.G. Moraes, “Energy-efficient cache coherence protocol for NoC-based MPSoCs”. In Proc. of the 24th ACM symp. on Integrated circuits and systems design (SBCCI '11), pp. 215-220, Aug 2011.

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

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

[18] L. Bai, J. Wang and B. Huang, "The Effect of Shared L2 Cache on Determinism in Airborne Embedded System," in Proc. of the 12th Inter. Conf. on Computational Intelligence and Security (CIS), Wuxi, pp.697-700,2016.

[19] Marino, M.D. and Li, KC., “Last level cache size heterogeneity in embedded systems”, Journal of Supercomput (2016) 72: 503. Vol. 72, Issue 2, pp. 503-544, February 2016.

[20] M. Loghi, M. Poncino and L. Benini, “Cache Coherence Tradeoffs in Shared-Memory MPSoCs”, ACM Trans. on Embedded Computing Systems, Vol. 5, No. 2, pp. 383-407, May 2006.

[21] G. Bournoutian and A. Orailoglu, “Dynamic, Multi-core Cache Coherence Architecture for Power-sensitive MobiProcessors”, in Proc. of CODES+ISSS, pages 89-98, 2011.

[22] A. Kayi and T. El-Ghazawi, “An adaptive cache coherence protocol for chip multiprocessors”, In Proc. of the 2nd Inter. Forum on Next-Generation Multicore/Manycore Technologies (IFMT '10), ACM, Article 4, June 2010.

[23] J. Shield, J. Diguet and G. Gogniat, “Asymmetric Cache Coherency: Policy Modifications to Improve Multicore Performace”, ACM Trans. on Reconfigurable Technology and Systems, Vol. 5, No. 3, Article No. 12, Oct 2012.

[24] C. Luk, R. Cohn, R. Muth, et. al., “Pin: building customized program analysis tools with dynamic instrumentation”, in Proc. of the 2005 ACM SIGPLAN Conf. on Programming language design and implementation (PLDI '05), Vol. 40, Issue 6, pp. 190-200, June 2005.

[25] S. J. E. Wilton and N. P. Jouppi, “CACTI: an enhanced cache access and cycle time model,” In IEEE Journal of Solid-State Circuits, Vol. 31, Issue: 5, pp.677 –688, May 1996.

[26] N. B. Mallya, G. Patil and B. Raveendran, “Way Halted Prediction Cache: An Energy Efficient Cache Architecture for Embedded Processors ”, in Proc. of 28th Inter. Conf. on VLSI Design (VLSID), pp. 65-70, Jan 2015.

[27] 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.