Asynchronous Systems Cache Coherence - Project Design

Karthik Reddy - 109721778

Project Description:

Implement MESI protocol in DistAlgo[1].
Verify correctness of the MESI protocol implementation.
Comprehensive comparison of cache coherence protocols in DistAlgo.
Measure the performance metrics of various cache coherence protocols on benchmarks and compare with other protocols.

Project Design:

Consider a distributed shared memory system. Each process(think Processor) has its own local cache(think L1,L2 cache) of data, cached from a large memory database(think main memory). We demonstrate and compare the performance of various cache coherence protocols, which are used to make these local caches of memory, coherent.

There are 3 main classes. Processor, MESI_cache_controller and MESI_directory_controller.

Processor: Each Processor runs as a separate process, each having its own trace file. Each Processor process communicates with its own Cache Controller. Cache Controller runs as a separate process. Processor and Cache controller communicate through messages. Processor reads the trace file and issues load/store instructions to the Cache Controller process. Cache Controller once it executes the instruction acknowledges the Processor which sends the next instruction for the cache to execute.

MESI_cache_controller: This is the Cache Controller process associated with each Processor object. Cache controller is internally represented as a state machine. There are two types of transitions possible: local and remote. Remote transitions are in response to the other caches or the directory controller. Each cache controller has a reference to the Directory Controller. The cache controller communicates(unicast message) with the directory controller as per the MESI protocol. The cache controller also receives messages from the directory controller.

MESI_directory_controller: This is the Directory Controller object. Receives messages from cache controllers. Queues up these requests and according to MESI protocol, performs actions. Internally represented as a state machine. Stores “state”, “owner” and a list of “sharers” for every cache line. Uses this information to track the status of the cache and the data they cache.

Brief description of MESI protocol:

Every cache line is marked with one of the four following states (coded in two additional bits):

Modified
The cache line is present only in the current cache, and is dirty; it has been modified from the value in main memory. The cache is required to write the data back to main memory at some time in the future, before permitting any other read of the (no longer valid) main memory state. The write-back changes the line to the Exclusive state.

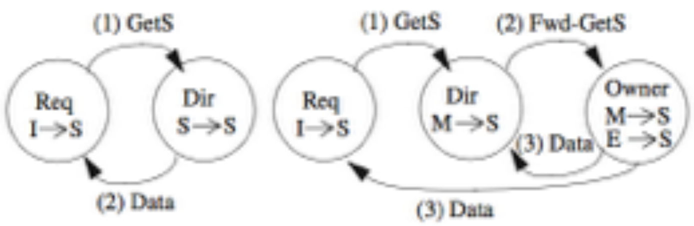
Exclusive
The cache line is present only in the current cache, but is clean; it matches main memory. It may be changed to the Shared state at any time, in response to a read request. Alternatively, it may be changed to the Modified state when writing to it.

Shared
Indicates that this cache line may be stored in other caches of the machine and is clean; it matches the main memory. The line may be discarded (changed to the Invalid state) at any time.

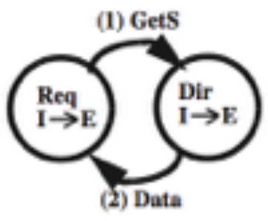
Invalid
Indicates that this cache line is invalid (unused).

More information on the MESI is provided by [2].

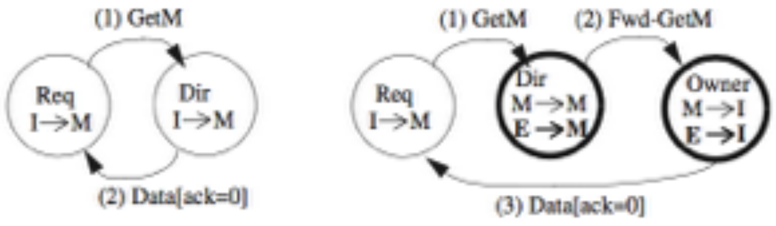
Following are the state diagrams and state machine(cache and directory controller) transitions of the MESI protocol.

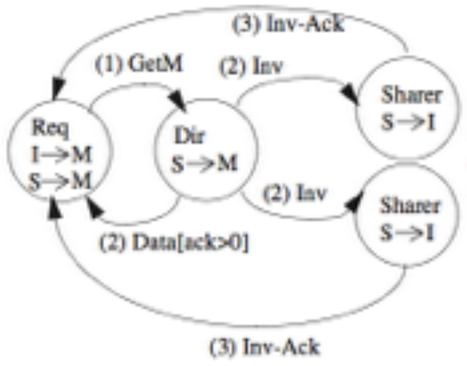


Transitions from I to S.



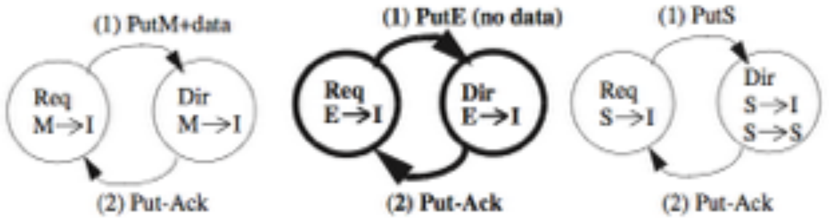
Transition from I to E.





The only sharer might be the requestor, in which case no Invalidation messages are sent and the Data message from the Dir to Req has an AckCount of zero.

Transitions from I or S to M. Transition from E to M is silent.



Transition from M or E or S to I

TABLE 8.4: MESI Directory Protocol—Directory Controller

	GetS	**GetM**	**PutS-NotLast**	**PutS-Last**	**PutM+data from Owner**	**PutM from Non-Owner**	**PutE (no data) from Owner**	**PutE from Non-Owner**	**Data**
I	send Exclusive data to Req, set Owner to Req/E	send data to Req, set Owner to Req/M	send Put-Ack to Req	send Put-Ack to Req	🟩	send Put-Ack to Req	🟩	**send Put-Ack to Req**	🟩
S	send data to Req, add Req to Sharers	send data to Req, send Inv to Sharers, clear Sharers, set Owner to Req/M	remove Req from Sharers, send Put-Ack to Req	remove Req from Sharers, send Put-Ack to Req/I	🟩	remove Req from Sharers, send Put-Ack to Req	🟩	**remove Req from Sharers, send Put-Ack to Req**	🟩
E	**forward GetS to Owner, make Owner sharer, add Req to Sharers, clear Owner/S^D**	**forward GetM to Owner, set Owner to Req/M**	**send Put-Ack to Req**	**send Put-Ack to Req**	**copy data to mem, send Put-Ack to Req, clear Owner/I**	**send Put-Ack to Req**	**send Put-Ack to Req, clear Owner/I**	**send Put-Ack to Req**	🟩
M	forward GetS to Owner, make Owner sharer, add Req to Sharers, clear Owner/S^D	forward GetM to owner, set Owner to Req	send Put-Ack to Req	send Put-Ack to Req	copy data to mem, send Put-Ack to Req, clear Owner/I	send Put-Ack to Req	🟩	**send Put-Ack to Req**	🟩
S^D	stall	stall	remove Req from Sharers, send Put-Ack to Req	remove Req from Sharers, send Put-Ack to Req	🟩	remove Req from Sharers, send Put-Ack to Req	🟩	**remove Req from Sharers, send Put-Ack to Req**	copy data to LLC/mem/S

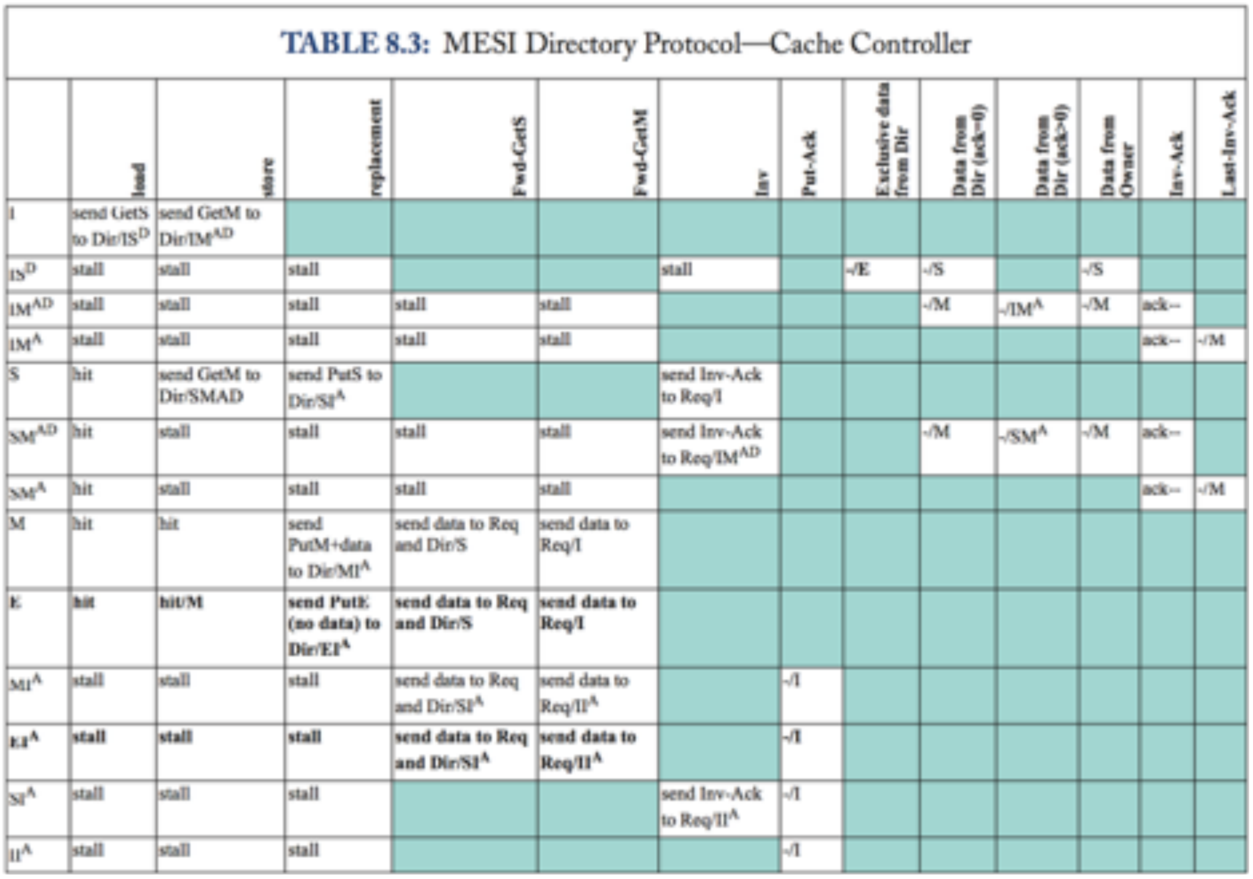


TABLE 8.3: MESI Directory Protocol—Cache Controller

	load	store	replacement	Fwd-GetS	Fwd-GetM	Inv	Put-Ack	Exclusive data from Dir	Data from Dir (ack=0)	Data from Dir (ack>0)	Data from Owner	Inv-Ack	Last-Inv-Ack
I	send GetS to Dir/IS^D	send GetM to Dir/IM^{AD}											
IS^D	stall	stall	stall			stall		-/E	-/S		-/S		
IM^{AD}	stall	stall	stall	stall	stall				-/M	-/IM^A	-/M	ack--	
IM^A	stall	stall	stall	stall	stall							ack--	-/M
S	hit	send GetM to Dir/SMAD	send PutS to Dir/SI^A			send Inv-Ack to Req/I							
SM^{AD}	hit	stall	stall	stall	stall	send Inv-Ack to Req/IM^{AD}			-/M	-/SM^A	-/M	ack--	
SM^A	hit	stall	stall	stall	stall							ack--	-/M
M	hit	hit	send PutM+data to Dir/MI^A	send data to Req and Dir/S	send data to Req/I								
E	**hit**	**hit/M**	**send PutE (no data) to Dir/EI^A**	**send data to Req and Dir/S**	**send data to Req/I**								
MI^A	stall	stall	stall	send data to Req and Dir/SI^A	send data to Req/II^A		-/I						
EI^A	**stall**	**stall**	**stall**	**send data to Req and Dir/SI^A**	**send data to Req/II^A**		-/I						
SI^A	stall	stall	stall			send Inv-Ack to Req/II^A	-/I						
II^A	stall	stall	stall				-/I						

MESI implementation in DistAlgo:

Please refer to "mesi_protocol.txt" for information on the implementation. It is an extracted Pydoc file which explains briefly the implementation, various API within the code and instructions to execute the driver program.

My DistAlgo implementation completely implements the above state diagrams, handling almost if not all of the race conditions between different states and different messages.

Code repository: https://github.com/karthikbox/cache_coherence/tree/mesi_implementation
Driver file: https://github.com/karthikbox/cache_coherence/blob/mesi_implementation/main.da
Total lines of Code: 888
API Pydoc: mesi_protocol.txt in my project page or
https://github.com/karthikbox/cache_coherence/blob/mesi_implementation/mesi_protocol.txt
Traces: https://github.com/karthikbox/cache_coherence/tree/mesi_implementation/traces

Instructions to run the driver file:

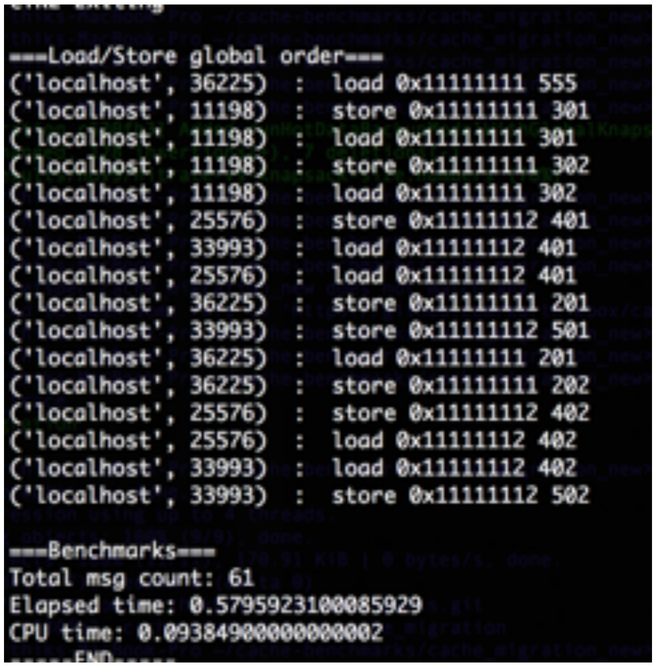
```
$ python3 -m da main.da <#processors> ./traces
```

Correctness:

MESI protocol ensures sequential consistency. Sequential consistency has the following requirements:

1. Instructions issued at each Processor are executed in the same order
2. Read instruction on a particular address reflects the most recent write on that address by anyone in the system

In my implementation, logical clocks are used to get the global order of reads and writes issued by each processor. Following is an instance of the global order. With the help of this global order, we can verify sequential consistency properties 1 and 2 easily. Thus correctness is verified.



```
===Load/Store global order===
('localhost', 36225)  :  load 0x11111111 555
('localhost', 11198)  :  store 0x11111111 301
('localhost', 11198)  :  load 0x11111111 301
('localhost', 11198)  :  store 0x11111111 302
('localhost', 11198)  :  load 0x11111111 302
('localhost', 25576)  :  store 0x11111112 401
('localhost', 33993)  :  load 0x11111112 401
('localhost', 25576)  :  load 0x11111112 401
('localhost', 36225)  :  store 0x11111111 201
('localhost', 33993)  :  store 0x11111112 501
('localhost', 36225)  :  load 0x11111111 201
('localhost', 36225)  :  store 0x11111111 202
('localhost', 25576)  :  store 0x11111112 402
('localhost', 25576)  :  load 0x11111112 402
('localhost', 33993)  :  load 0x11111112 402
('localhost', 33993)  :  store 0x11111112 502

===Benchmarks===
Total msg count: 61
Elapsed time: 0.5795923100085929
CPU time: 0.09384900000000002
```

Performance:

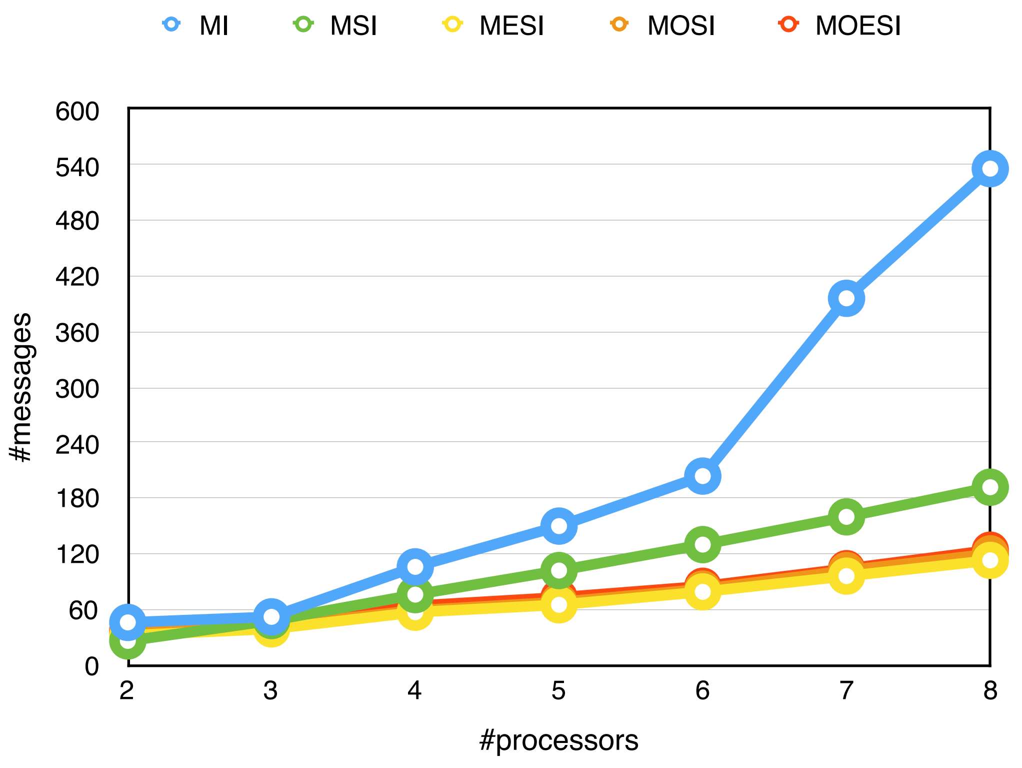
All of the following measurements are taken on the following machine specifications.
Mac OSX, 2.7 GHz Intel Core i5, 8 GB 1867 MHz DDR3.

Message Count:

The following table and plot indicates the number of messages exchanged between ’n’ processors in a particular protocol. Notice that the number of messages in MESI protocol are lower than all other protocols. This is because: MOSI,MOESI because of OWNED state have to exchange a lot more messages than MESI. MI is snooping based implementation, hence the number of messages will grow exponentially compared to a directory based protocol such as MESI. MSI, doesn't have the EXCLUSIVE state, hence uses more messages to obtain write permission immediately after read permission.

#messages in each protocol vs #processors

#processors	MI	MSI	MESI	MOSI	MOESI
2	46	26	32	36	38
3	52	48	39	42	46
4	106	76	57	59	64
5	150	102	65	68	73
6	204	130	79	82	85
7	396	160	96	102	104
8	536	192	113	120	124

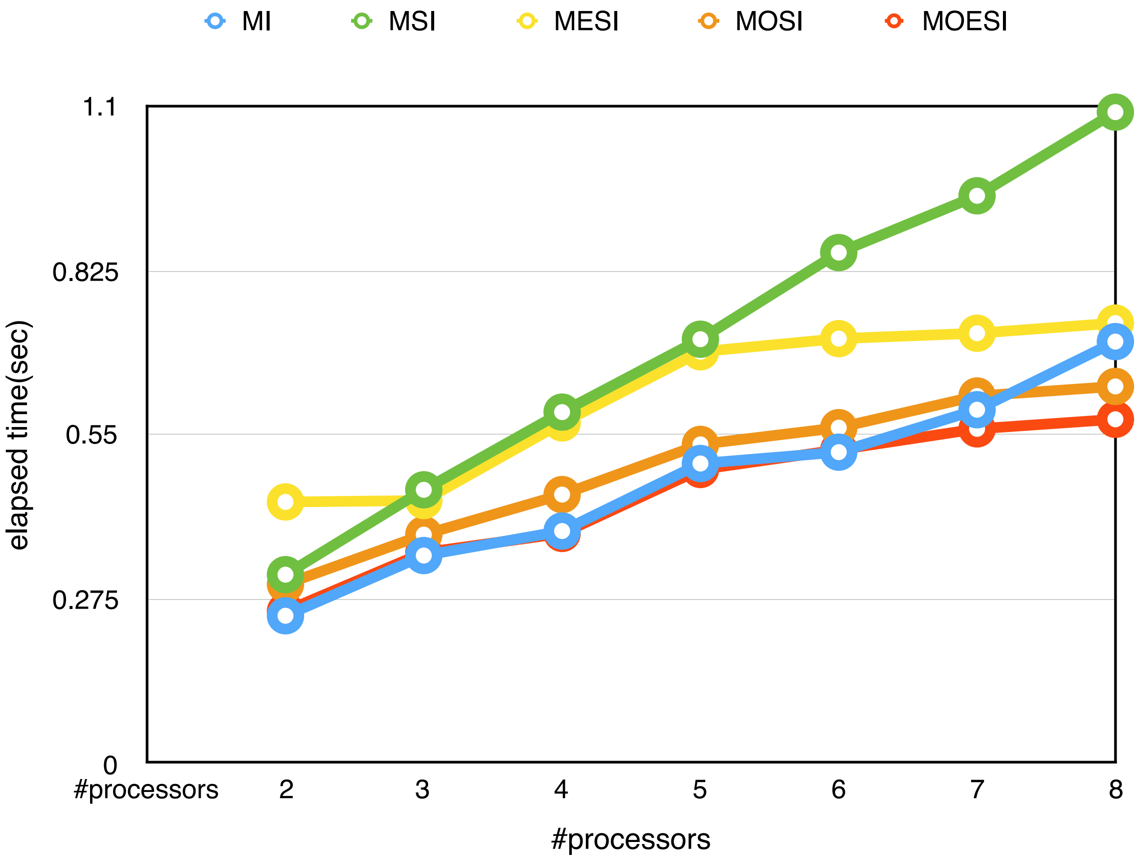


Elapsed Time:

The following table and plot indicates the elapsed time for the trace to finish between 'n' processors in a particular protocol. Notice that the elapsed time in the case of MESI protocol is lower than MSI but greater than MOESI,MOSI and MI. This is as expected. The trace we use in the experiments are write dominated. In such situations, the MI protocol works fast because the SHARED state is absent(if present, then each sharer has to invalidate his copy and send an acknowledgement).

Table 1

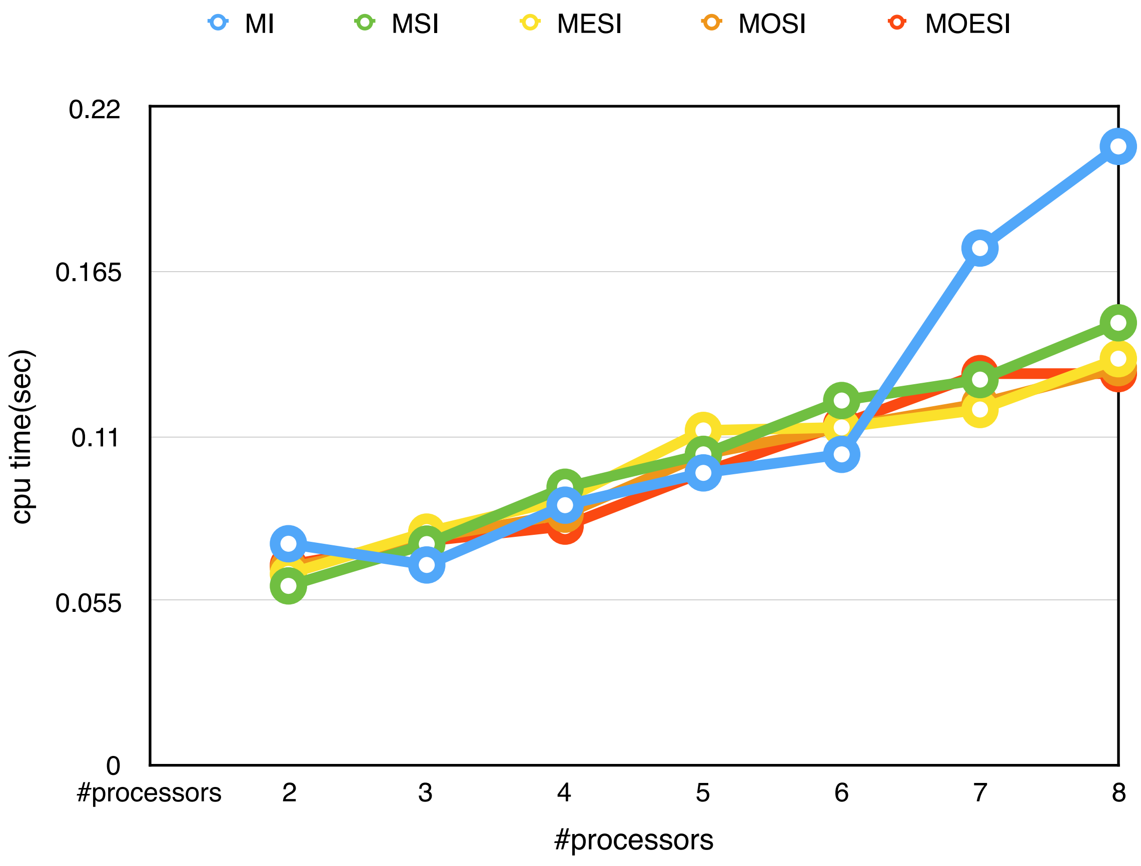
#processors	MI	MSI	MESI	MOSI	MOESI
2	0.2472	0.316	0.4377	0.300	0.255
3	0.348	0.458	0.44	0.383	0.353
4	0.389	0.588	0.57	0.450	0.385
5	0.502	0.710	0.689	0.534	0.492
6	0.521	0.855	0.711	0.562	0.525
7	0.592	0.950	0.72	0.615	0.560
8	0.706	1.09	0.737	0.631	0.576



CPU Time:
The following table and plot indicates the elapsed time for the trace to finish between 'n' processors in a particular protocol. No particular trend is observed as there all the protocols perform better sometime and worse the rest of the time.

Table 1-1

#processors	MI	MSI	MESI	MOSI	MOESI
2	0.0741	0.06	0.064	0.066	0.067
3	0.067	0.074	0.078	0.076	0.075
4	0.087	0.093	0.089	0.084	0.080
5	0.0978	0.104	0.112	0.104	0.098
6	0.104	0.122	0.113	0.113	0.114
7	0.173	0.129	0.119	0.121	0.131
8	0.207	0.148	0.136	0.133	0.131



Team Member’s role

Parag Gupta : Implement MSI in DistAlgo
Karthik Reddy : Implement MESI in DistAlgo
Paul Mathew : Implement MI, feasibility Token Coherence in DistAlgo
Amit Khandelwal : Implement MOESI in DistAlgo
Garima Gehlot : Implement MOSI in DistAlgo

References

[1] Yanhong A. Liu, Bo Lin, and Scott Stoller. “DistAlgo Language Description”. https://github.com/DistAlgo/distalgo
[2] Martin, Milo MK, Mark D. Hill, and David Wood. "Token coherence: Decoupling performance and correctness." Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on. IEEE, 2003.
[3] karthik reddy.   https://github.com/karthikbox/cache_coherence/tree/mesi_protocol