E0-243 High Performance Computer Architecture (Fall 2020)

Submission - Term Paper 2

Submitted By :

Aman Choudhary (amanc@iisc.ac.in) - SR No:17920
Shashank Singh (shashanksing@iisc.ac.in) - SR No:17887

1 Solution to Question-1

1.1 Part A : FSM State Diagram

Motivation - The issue in MSI protocol with regards to Cache-to-Cache transfer is that either all sharers will reply to the BusRd/BusRdX requests or none of them will. In first case, there will be unnecessary bus traffic and in the second case longer latency LLC access will be incurred. To solve this, we need one of the sharers to own the responsibility to provide any requests with the data and hence a new state C is introduced.

The cache to cache transfer is ensured in the following scenarios :

- Suppose a given core X, not having a block B in its cache wants to read it. If there's no cache in C state for that block B, then X will go into C state for B instead of S state, so that it can aid cache-to-cache transfers for further accesses of B on other cores.
- When a core X not having a block B in cache wants to read/write to it, and B is present in C state in another core Y's cache then, Y will reply with the data to X.
- When a block in C state is evicted, there will be no one in C state. As soon as a cache not having the block tries to fetch it, it will go into C state and fulfill further requests.

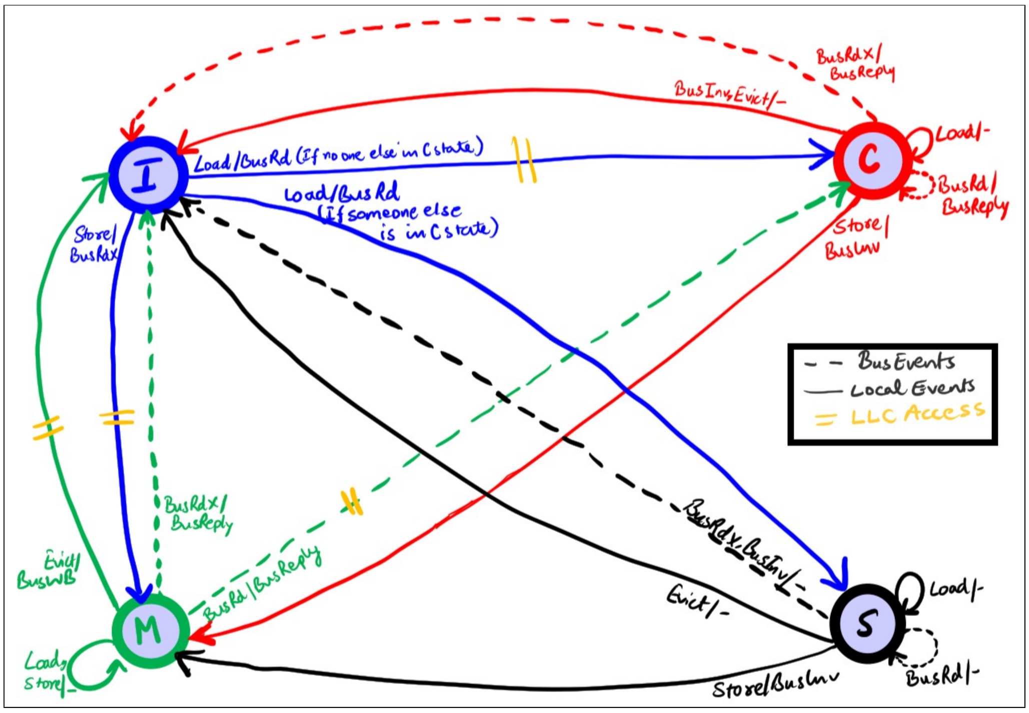
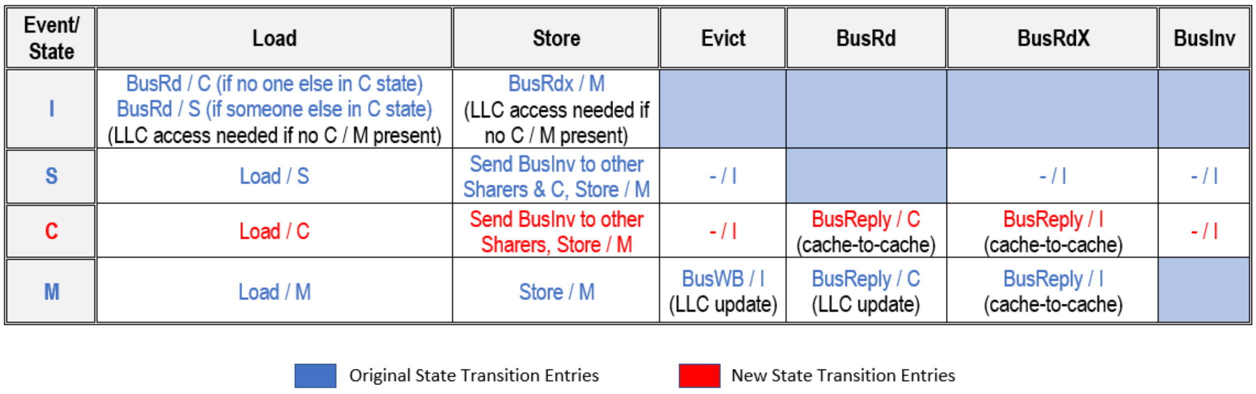


Figure 1: MCSI State Transition Diagram



Event/ State	Load	Store	Evict	BusRd	BusRdX	BusInv
I	BusRd / C (if no one else in C state) BusRd / S (if someone else in C state) (LLC access needed if no C / M present)	BusRdx / M (LLC access needed if no C / M present)				
S	Load / S	Send BusInv to other Sharers & C, Store / M	- / I		- / I	- / I
C	Load / C	Send BusInv to other Sharers, Store / M	- / I	BusReply / C (cache-to-cache)	BusReply / I (cache-to-cache)	- / I
M	Load / M	Store / M	BusWB / I (LLC update)	BusReply / C (LLC update)	BusReply / I (cache-to-cache)	

🟦 Original State Transition Entries 🟥 New State Transition Entries

Figure 2: MCSI State Transition Table

1.2 Part B : Additional Hardware Support

We do require an additional support from hardware to implement *MCSI* protocol. Whenever a cache not having a block tries to fetch it, we need to identify whether any other cache has it in C state. In order to do this, we use a mechanism outlined below:

C-Line : This is a shared line which the caches use to assert whether the block B in context, is present with them in C state or not. A cache having the block in C state will assert a value 1 on the line, while those who don't, assert 0. A simple *OR* logic of all the asserted values signifies whether any cache has the block in C state or not.

- Core X tries to read a block B, but misses in the cache. Consequently, it will issue a *BusRd*.
- When the sharers of the block see this *BusRd*, they assert on the **C-Line**. Now, X's cache controller knows whether to go into C state or the S state when it receives the block.

1.3 Part C : MESI vs MCSI

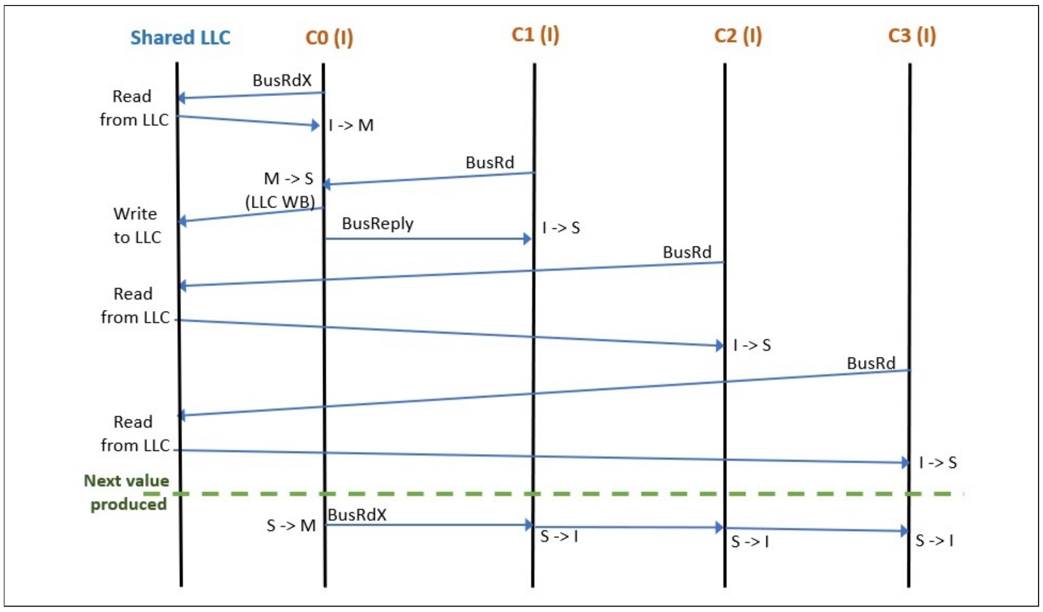


Figure 3: Events in MESI Protocol

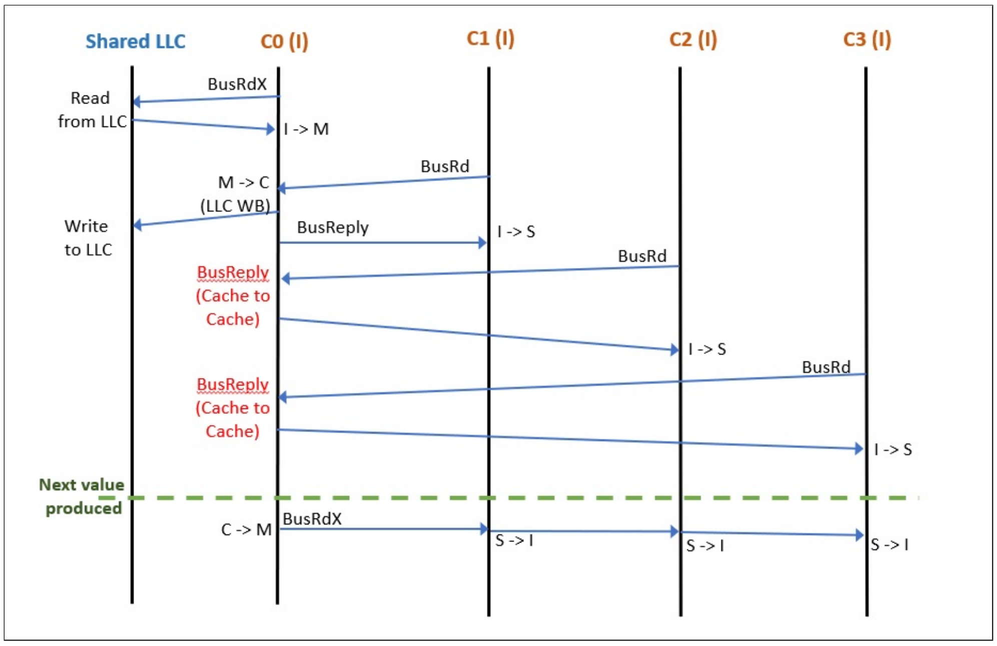


Figure 4: Events in MCSI Protocol

- We **assume** that consumption is of non-destructive nature i.e. the consumers can read a value of shared variable multiple times.
- In every iteration the producer ($C0$) generates the data in shared variable ready to be consumed by consumers ($C1, C2, C3$). We **assume** the processes/threads are synchronized properly and consumption attempts happen strictly after production of data.
- From the diagrams, it is evident that in the case of *MCSI* protocol, **cache-to-cache** transfers are maximized as two reads in every iteration can be served from $C0$ in contrast to memory/LLC access required for every read in the case of *MESI* protocol.

2 Solution to Question-2

2.1 Directory Side

New notations added

S^E : Intermediate state after eviction from S state, waiting for E-Ack

M^E : Intermediate state after eviction from M state, waiting for dirty data

$Evict$: A local event indicating eviction needed for a dir entry

E : Evict signal sent to caches

E-Ack : Acknowledgement received from caches for E signal

Highlights

- Any entry being evicted from dir needs to invalidate its sharers/owners
- A directory entry in I state can simply be replaced when an eviction of the entry is needed
- A directory entry in S state needs to send E signal to all the sharers and initialize a counter E-Ack=$|S|$ to wait for caches to acknowledge.
- A directory entry in M state needs to send E signal to the owner (cache having the block in M state) and wait for response having dirty data. The dirty data needs to be updated in the memory/shared LLC
- The directory serves only those GetS/GetM requests (for the entry being evicted) which have been received before eviction was initiated. Any request post this will be queued and served after eviction is complete

2.2 Cache Side

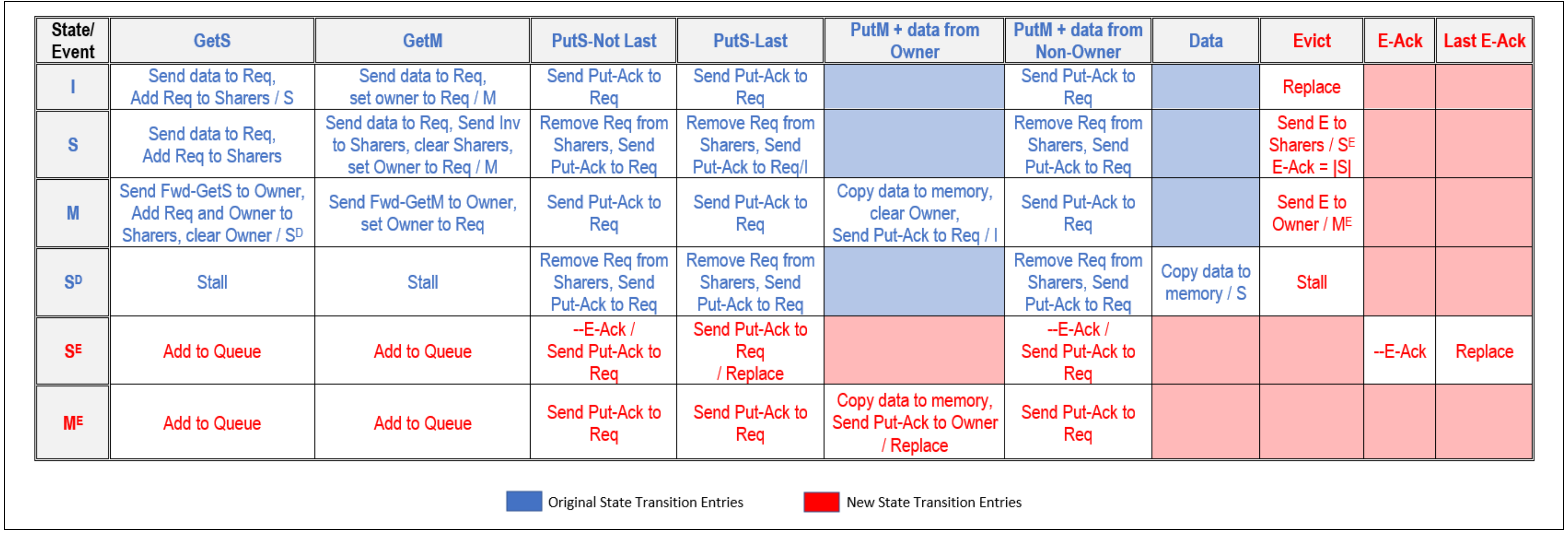
New notations added

$IS^{DE}, IM^{ADE}, IM^{AE}, SM^{ADE}, SM^{AE}$: Intermediate state upon receiving E from dir in $IS^D, IM^{AD}, IM^A, SM^{AD}, SM^A$ state respectively

Highlights

- If an E signal is received for some block not in I state then cache controller must initiate the process of invalidating it.
- When E is received for a block in S state, it's invalidated and E-Ack is sent to dir.
- When E is received for a block in M state, it's invalidated (intermed. state II^A) and dirty data is sent to dir and Put-Ack is awaited from dir.
- The behaviour for the new intermediate states $IS^{DE}, IM^{ADE}, IM^{AE}, SM^{ADE}, SM^{AE}$ is very similar to the corresponding old intermediate states.
- IS^{DE} : While transitioning from I to S the cache received an E signal and moved to IS^{DE} state. Now, it waits for the data from dir/owner and upon receiving it, will fulfill the load request and then transition to I state.

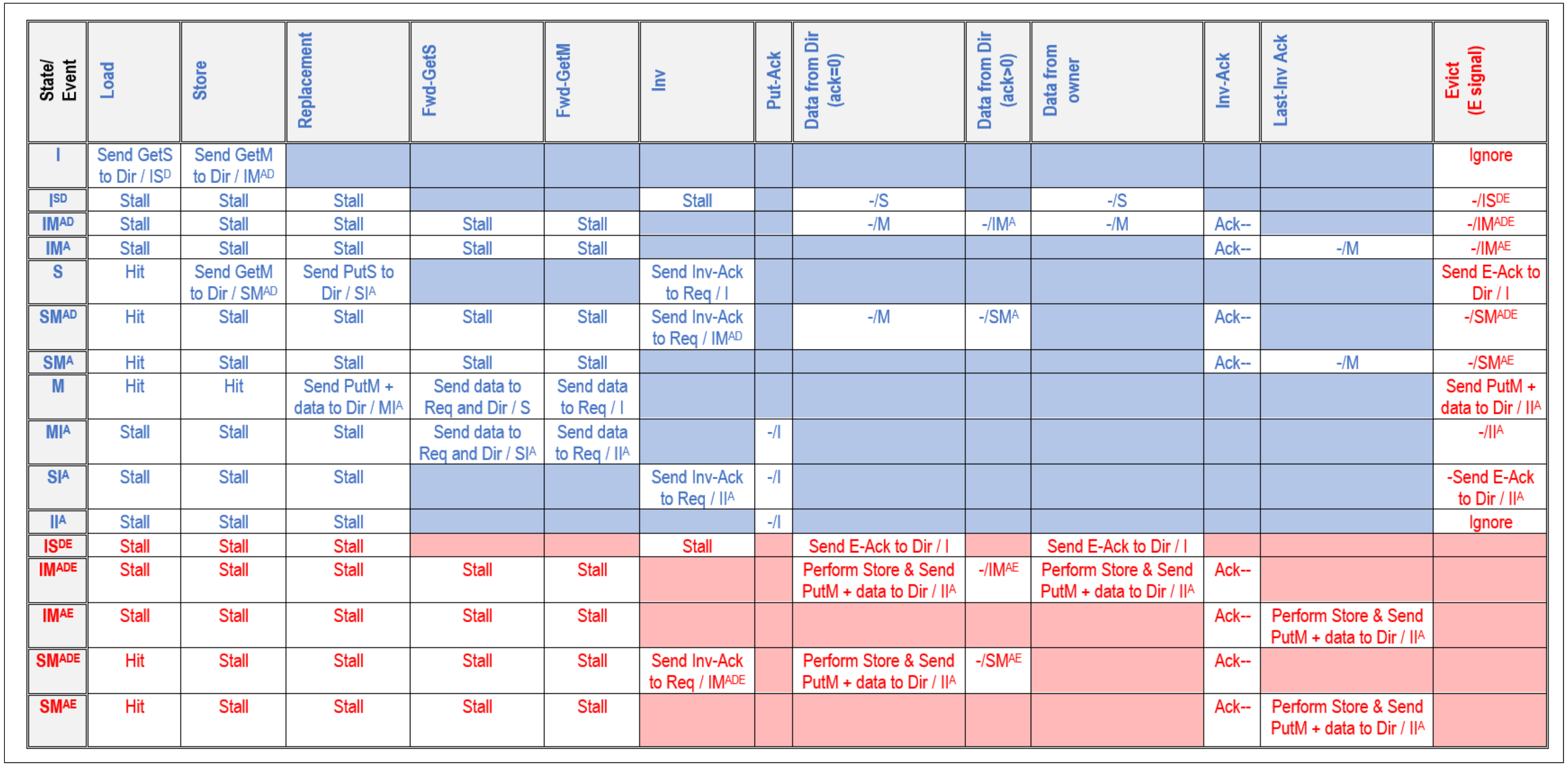
- IM^{ADE}, IM^{AE} : While transitioning from I state to M, the cache received an E signal and moved to IM^{ADE} or IM^{AE} state depending upon the previous state. Now, it waits for the Inv-Ack and perhaps data from dir/owner. Upon receiving them, it will fulfill the store request, send the data with PutM to directory and then transition to I state.
- SM^{ADE}, SM^{AE} : Similarly, while transitioning from S state to M, the cache received an E signal and moved to SM^{ADE}, SM^{AE} state depending upon the previous state. Now, it waits for the Inv-Ack and perhaps data from dir/owner. Upon receiving them, it will fulfill the store request, send the data with PutM to directory and then transition to I state.



State/ Event	GetS	GetM	PutS-Not Last	PutS-Last	PutM + data from Owner	PutM + data from Non-Owner	Data	Evict	E-Ack	Last E-Ack
I	Send data to Req, Add Req to Sharers / S	Send data to Req, set owner to Req / M	Send Put-Ack to Req	Send Put-Ack to Req	🟦	Send Put-Ack to Req	🟦	Replace	🟥	🟥
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	🟦	Send E to Sharers / S^E E-Ack = |S|	🟥	🟥
M	Send Fwd-GetS to Owner, Add Req and Owner to Sharers, clear Owner / S^D	Send Fwd-GetM to Owner, set Owner to Req	Send Put-Ack to Req	Send Put-Ack to Req	Copy data to memory, clear Owner, Send Put-Ack to Req / I	Send Put-Ack to Req	🟦	Send E to Owner / M^E	🟥	🟥
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	Copy data to memory / S	Stall	🟥	🟥
S^E	Add to Queue	Add to Queue	--E-Ack / Send Put-Ack to Req	Send Put-Ack to Req / Replace	🟥	--E-Ack / Send Put-Ack to Req	🟥	🟥	--E-Ack	Replace
M^E	Add to Queue	Add to Queue	Send Put-Ack to Req	Send Put-Ack to Req	Copy data to memory, Send Put-Ack to Owner / Replace	Send Put-Ack to Req	🟥	🟥	🟥	🟥

🟦 Original State Transition Entries 🟥 New State Transition Entries

Figure 5: Directory side state table



State/ Event	Load	Store	Replacement	Fwd-GetS	Fwd-GetM	Inv	Put-Ack	Data from Dir (ack=0)	Data from Dir (ack>0)	Data from owner	Inv-Ack	Last-Inv Ack	Evict (E signal)
I	Send GetS to Dir / IS^D	Send GetM to Dir / IM^{AD}											Ignore
IS^D	Stall	Stall	Stall			Stall		-/S		-/S			-/IS^{DE}
IM^{AD}	Stall	Stall	Stall	Stall	Stall			-/M	-/IM^A	-/M	Ack--		-/IM^{ADE}
IM^A	Stall	Stall	Stall	Stall	Stall						Ack--	-/M	-/IM^{AE}
S	Hit	Send GetM to Dir / SM^{AD}	Send PutS to Dir / SI^A			Send Inv-Ack to Req / I							Send E-Ack to Dir / I
SM^{AD}	Hit	Stall	Stall	Stall	Stall	Send Inv-Ack to Req / IM^{AD}		-/M	-/SM^A		Ack--		-/SM^{ADE}
SM^A	Hit	Stall	Stall	Stall	Stall						Ack--	-/M	-/SM^{AE}
M	Hit	Hit	Send PutM + data to Dir / MI^A	Send data to Req and Dir / S	Send data to Req / I								Send PutM + data to Dir / II^A
MI^A	Stall	Stall	Stall	Send data to Req and Dir / SI^A	Send data to Req / II^A		-/I						-/II^A
SI^A	Stall	Stall	Stall			Send Inv-Ack to Req / II^A	-/I						-Send E-Ack to Dir / II^A
II^A	Stall	Stall	Stall				-/I						Ignore
IS^{DE}	Stall	Stall	Stall			Stall		Send E-Ack to Dir / I		Send E-Ack to Dir / I			
IM^{ADE}	Stall	Stall	Stall	Stall	Stall			Perform Store & Send PutM + data to Dir / II^A	-/IM^{AE}	Perform Store & Send PutM + data to Dir / II^A	Ack--		
IM^{AE}	Stall	Stall	Stall	Stall	Stall						Ack--	Perform Store & Send PutM + data to Dir / II^A	
SM^{ADE}	Hit	Stall	Stall	Stall	Stall	Send Inv-Ack to Req / IM^{ADE}		Perform Store & Send PutM + data to Dir / II^A	-/SM^{AE}		Ack--		
SM^{AE}	Hit	Stall	Stall	Stall	Stall						Ack--	Perform Store & Send PutM + data to Dir / II^A	

Figure 6: Cache side state table