ECE552 Lab 5: Cache Coherence

Yi Fan Zhang 1000029284, Jianwei Sun 1000009821

I. Prelab Questions

1. **Why are transient states necessary?**
   When a block transitions between MSI states, it will encounter delays while waiting for responses. During this time, another event can transition between states, violating the atomicity requirement. Transient states are used as intermediate states to facilitate these transitions.

2. **Why does a coherence protocol use stalls?**
   To maintain coherence, any read/write requests to the block cannot happen when it's in transition state. Thus, cores that send requests to blocks in transition states must be stalled until the blocks arrive at a stable state and enable its read/write permissions.

3. **What is deadlock and how can we avoid it?**
   A deadlock is when two inter-dependent events exist, and no forward progress is made. For example, if A depends on the results of B, while B depends on the results of A, then both will wait for each other and never progress. Virtual Networks is a solution to deadlocks. It logically separates networks with their own set of I/O queues and buffers using tags.

4. **What is the functionality of Put-Ack (i.e., WB-Ack) messages? They are used as a response to which message types?**
   When the Cache evicts a block, its controller sends a PutM or PutS message to the Directory controller depending on its state. The directory controller then sends a Put-Ack response to tell the cache controller that the eviction has been completed. Put-Ack is used as forward/directory initiated requests in response to “PUT” requests.

5. **What determines who is the sender of a data reply to the requesting core? Which are the possible options?**
   The directory keeps track of the owners of each block. If the directory owns the block in a shared state, it'll respond with the data, and change the block state in the cache controller to S. If the data is owned by a local cache in M state, then the directory controller forwards data requests to its owner. The owner then responds with the data and transitions to S, thus making the directory the owner. If block is in none of the local caches, then directory will fetch its data from memory.

6. **What is the difference between a Put-Ack and an Inv-Ack message?**
   When cache issues a GetM request, all the other sharers of the block must invalidate their respective blocks and each send an acknowledgement, Inv-Ack, to the requester. The transition is complete only when the cache receives Inv-Acks from all the sharers.
   Put-Ack is a single forward request message from the directory to the requester as a response to the requester's PutM/PutS. Put-Ack is a coherence request message, whereas Inv-Ack is a coherence response message.

II. Methodology Questions

1. **How does the FSM know it has received the last Inv Ack reply for a TBE entry?**

The Directory sends an AckCount that tells the Cache Controller how many Inv-Acks it needs to expect. Each Inv-Ack decrements the “ack” field in the TBE; once an Inv-Ack decreases “ack” to 0, the FSM would know that a Last_Inv_Act has been sent.

2. **How is it possible to receive a PUTM request from a Non-Owner core? Please provide a set of events that will lead to this scenario.**

As PutM and Fwd-GetM is sent through two different virtual channels, it is possible for the two signals to arrive out-of-order. Consider the following: C1 sends a GetM request to the directory, then C2, the owner of the block, sends a PutM to the directory. The directory receives the GetM, forwards the GetM to C2, and changes the ownership to requester C1. Finally, C2's PutM arrives at the directory. However, the directory has already changes the owner of the block, thus a PutM has been received from a non-owner core.

3. **Why do we need to differentiate between a PUTS and a PUTS-Last request?**

PutS-Last request is sent when the last core with a copy of the block evicts it from its cache. The directory needs to set its block to Invalid so that the next time this block is requested, the directory will know to retrieve it from memory. Regular PutS would not change the state of the directory, as the block will still be shared among other cores.

4. **How is it possible to receive a PUTS-Last request for a block in modified state in the directory? Please provide a set of events that will make this possible.**

Consider C1 and C2 in state S. C2 sends a GetM request to the directory. The directory would then send the data to C2 and a forward invalidation request to C1, and then transition to M. However, during this time, C1 sends a PutS request to the directory. (this is possible as the forwarded requests and initiated requests are on two different virtual channels). Thus, when the Directory receives C1's PutS, it will have received a PutS-Last request while in M.

5. **Why is it not possible to get an Invalidation request for a cache block in a Modified state? Please explain.**

When a cache block (C1) is in Modified state, it means that it is the owner, and is not shared with other caches. When another block (C2) sends a GetM to the directory, the directory would forward the GetM to C1 for its data rather than issuing an invalidation request. Invalidation requests are only sent when the block is in S state, in which case no cores are in Modified state.

6. **Why is it not possible for the cache controller to get a Fwd-GetS request for a block in SI_A state? Please explain.**

Fwd-GetS is only sent when the block is owned by a single core in M state; this transitions it to the S state. The moment this block becomes a shared state, the block will be owned by the directory which provides the data to any future blocks requesting GetS. Thus, in SI_A state, a block wont get a Fwd-GetS, as the directory can send the data without requesting it from any particular local cache.

7. **Was your verification testing exhaustive? How can you ensure that the random tester has exercised all possible transitions (i.e., all {State, Event} pairs)**

The verification was tested up to all 16 cores, and 1 million load counts. Random combination of cores and load counts were also tested to ensure success against a random testers. During this process, to

verify that all possible FSM transitions were taken into account, we also used print statements for more detailed analysis and debugging, as well as generated a table to compare against our conceptualization.

III. Protocol Modifications

Most of the modifications were for Table 8.2 to accommodate the latency between directory and memory. Table 8.1 had very little modifications

Modifications to Table 8.1

Modification	Description
Added IFetch event	L1 is used for both instruction fetches and read requests, so IFetch was grouped with Loads in our protocol
Misc. changes for implementation	various Cache/TBE allocation and deallocation actions were added to the events to set up data transfer

Modifications to Table 8.2

New Transition states (rows) with Memory interface events (columns) are added to the original table. Detailed modifications are show in bold below.

	GetS	GetM	PutS - NotLast	PutS-Last	PutM+data from Owner	PutM+data from NonOwner	Data	**Data From Mem**	**Ack From Mem**
I	Request Data from Mem / Add Req to Sharers /IS_D	Request Data from Mem Send Data to Req / IM_D	Send Put-Ack to Req	Send Put-Ack to Req		Send Put-Ack to Req			
IS_D	**stall**	**stall**	**Send Put-Ack to Req**	**Send Put-Ack to Req**				**Send Data to Req / S**	
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		remove Req from Sharers, send PutAck to Req			
IM_D	**stall**	**stall**						**Set Req to owner / M**	
M	Send Fwd Get S to owner Add req and owner to sharers Clear to owner /MS_D	Send Fwd Get M to owner Set Owner to requester	send Put-Ack to Req	send Put-Ack to Req	**Write data to directory Copy data to memory, /MI_A**	Send Put-Ack to Req			
MS_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 PutAck to Req	**Write data to directory Copy data to memory /MS_A**		
MS_A	**stall**	**stall**							**-/S**
MI_A	**stall**	**stall**							**clear Owner, send Put-Ack to Req /I**

IV. Division of Work

The implementation design and conceptualization was performed equally by both members, this includes modifications to the tables, discussion of the entire protocol, the coding and answers to the lab questions. In particular, Jianwei typed out most of the code, and Yi Fan typed out the report.