Lab Assignment 6: Coherence

Freddy Chen 997363124 Anthony Alayo 997487401

Prelab Work

1. Why are transient states necessary?
Transient states are intermediate states that facilitate the transitions between two stable states. They are required because while state transitions happen instantaneously on paper, they do not happen atomically in real life. Transient states help cache coherence implementations avoid violating coherence protocols, such as the Single-Writer, Multiple-Reader (SWMR) invariant.

2. Why does a coherence protocol use stalls?
Coherence protocols use stalls for blocks in transient states because the block pending coherence requests does not have write or read permissions. If the coherence protocol didn't stall, it would violate its invariants. (eg. one core tries to write to an invalid block, while another core is writing to the same block that is modified)

3. What is deadlock and how can we avoid it?
Deadlock is when no forward progress is made in a system because of two inter-dependent events.

In the case of an MSI directory protocol, a deadlock can happen like below in a two-core system if only 1 queue is used for message passing per core:

- Block A modified in Core 2's cache
- Core 1 issues load request for block A, goes into transient state
- Core 1 issues store request to block A, stalls (because it's in transient state) and store is at the head of queue for Core 1
- Core 2 sends data block to Core 1, which gets added to the tail of the queue for Core 1, behind the store

At this point, the reply from Core 2 never gets seen as Core 1 is waiting at store to continue, resulting in a deadlock.

This can be avoided using Virtual Networks to separate logical networks / queues. For the MSI protocols, three virtual networks are enough to avoid deadlocks:

- Cache-Initiated Requests
- Directory-Initiated or Forwarded Coherence Requests
- Cache Responses

This separates the type of messages for each queue and avoids the scenario above.

4. What is the functionality of Put-Ack (i.e., WB-Ack) messages? They are used as a response to which message types?
Put-Ack messages are used by the directory to tell a cache that a write-back has successfully executed after a modified block has been evicted from that cache. Put-Ack messages are triggered by PutM messages sent from a cache. *This should tell the cache block that it is safe to update its state (ie. to shared).*

5. What determines who is the sender of a Data reply? Which are the possible options?
The owner of the Data determines who the sender of a Data reply is. The owner can be the directory (or one of the cache controllers?) depending on the scenario.

6. What is the difference between a Put-Ack and an Inv-Ack message?
Put-Ack messages are described in question 4.

Inv-Ack messages are used by caches to tell other caches that they are giving up their shared Data and marking it as invalid. This happens when a cache block requests modify access. More specifically:

- When a cache wants to modify Data, the cache controller sends a GetM (Get Modified) message to the Directory
- When the Directory receives a GetM message, it does the following actions:
    - Sends back the most up-to-date Data with an AckCount to the cache controller requester. The Directory will wait until it has ownership of the Data before sending anything back.
    - Send an Inv message to all caches that currently share Data
- All caches that share Data will send an Inv-Ack message back to the cache controller requesting to modify Data. As soon as the cache controller has AckCount number of Inv-Ack messages, it can proceed to change the state of the Data to Modified.

Methodology Questions

1. How does the FSM know it has received the last Inv Ack reply for a TBE entry?
The FSM uses the AckCount it receives from the directory to know how many Inv Ack replies to expect for a TBE entry.

2. How is it possible to receive a PUTM request from a NonOwner core? Please provide a set of events that will lead to this scenario.
Original state: Block A owner is C2, Block A is in state M on C2's cache and Dir
Here is a set of events that will lead to a PutM request from a NonOwner core:

- C1's cache controller does the following:
    - Sends a GetM request to Dir to get write access for Block A
    - Changes Block A's state from I to IM^{AD}
- C2's cache controller does the following:

    - Sends PutM to Dir to evict Block A (there are no silent evictions)
    - Changes Block A's state from M to MI^A
- C1's cache controller's request reaches Dir first. Dir sees that Block A is in state M, and does the following:
    - Sends Fwd-GetM to Block A's current owner, C2
    - Changes Block A's owner from C2 to C1
- C2's cache controller's request reaches Dir. It's a PutM request from a NonOwner core

3. Why do we need to differentiate between a PUTS and a PUTS-Last request?
We need to differentiate between PUTS and PUTS-Last requests because when all copies of a block are no longer in cache, the directory controller needs to retrieve it from memory upon the next request.

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.
Here is a set of events that will lead to a PutS-Last request for a block in modified state in the directory:

- C1's cache controller does the following:
    - Sends a PutS request to Dir to evict Block A (there are no silent evictions)
    - Changes Block A's state from S to SI^A.
- C2's cache controller does the following:
    - Sends a GetM request to Dir to get write access for Block A
    - Changes Block A's state from S to SM^{AD}
- C2's cache controller's request reaches Dir first. Dir does the following:
    - Responds to C2 with Data and AckCount equal to number of sharers in Block A's sharer list (including C1)
    - Sends Inv messages to each core in Block A's sharer list (including C1)
    - Changes Block A's state from S to M
- C1's cache controller's request reaches Dir. Block A in Dir will receive a PutS-Last request in the M state if C1 is the last core in Block A's sharer list

5. Why is it not possible to get an Invalidation request for a cache block in a Modified state? Please explain.
This is not possible because at any given time in the MSI protocol, one and only one of the following conditions must be true:

- A block is in the S state for all caches
- A block is in the I state for all caches
- A block is in the I state for all but one of the caches. The remaining cache must have that block in the M state.

As a direct result of the above conditions, there will never be a case where a block is in both the S and M states across different caches. Thus, when the Dir receives a GetM request, it will never send out an Inv request to a cache block in the M 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.

This is not possible because at any given time in the MSI protocol, one and only one of the following conditions must be true:

- A block is in the S state for all caches
- A block is in the I state for all caches
- A block is in the I state for all but one of the caches. The remaining cache must have that block in the M state.

When the directory controller receives a GetS request for a modified block, it will send a Fwd-GetS request to the core with the modified block. There must be only one cache with the modified block according to the conditions above. In other words, no cache has a shared copy of the block that we're interested in. Therefore, it is impossible for a cache controller to get a Fwd-GetS request for a block in the SI^A state.

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)?

Our verification testing was not exhaustive. We were able to pass some core configurations (ie. 1, 2, 4, 10, 16) with default loads and default cache sizes. However if we increased the load count to 1 million, we failed in some instances. It's also difficult to ensure that the random tester has exercised all possible transitions, but changing the random seed or changing the sizes of the L1 data and/or instruction cache would help to test them all. We had limited time to test exhaustively, so that is why we were not able to debug our failing cases.

Protocol Modifications

This section details the modifications we made to Tables 8.1 and 8.2. Table 8.2 is shown in detail because there are significant changes.

Below are our modifications with respect to Table 8.1. There are only slight changes so we have only shown the new transitions below.

New Transitions for Table 8.1

	Data_from_Dir_Ack_Cnt_Last
IM_AD	Write data to cache, update AckCount, deallocate TBE entry, inform local core that store has completed/M
SM_AD	Write data to cache, update AckCount, deallocate TBE entry, inform local core that store has completed/M

Below are our modifications with respect to Table 8.2. The transient states are mainly to facilitate the time to read and write data from memory. In the table, **_M means that we're waiting on the memory, while **_D means that we're waiting on data from a cache.

Modified Table 8.2

	GetS	GetM	PutS	PutS-Last	PutM+Data (owner)	PutM+data (nonowner)	Data	Mem Data	Mem Ack
I	Read from memory, add Req to Sharers/IS_M	Read from memory, set Owner to Req/IM_M		Send Put-Ack to req	Send Put-Ack to req	Send Put-Ack to req			
IS_M	Stall	Stall						Send Data to Req, copy Data to Dir/S	
IM_M	Stall	Stall						Send Data to Req, copy Data to Dir/M	
S	Add Req to Sharers, send Data to Req	Send Inv to Sharers, send AckCount to Req, send Data to Req, 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	-		
M	Send Fwd_GetS	Send Fwd_Get	Send Put-Ac	Send Put-Ac	Write to memory,	Send Put-Ack to Req	-		

	to Owner, add Owner to Sharers, clear Owner/MS_D	M to Owner, set Owner to Req	k to Req	k to Req	clear Owner/MI_M				
MS_D	Stall	Stall	Remove Req from Sharers, send Put-Ack to Req	Remove Req from Sharers, send Put-Ack to Req		Write Data to memory/MS_M	Write Data to memory/MS_M		
MS_M	Stall	Stall	Remove Req from Sharers, send Put-Ack to Req	Remove Req from Sharers, send Put-Ack to Req		-	-		-/S
MI_M	Stall	Stall	Send Put-Ack to Req	Send Put-Ack to Req		Send Put-Ack to Req			Send Put-Ack to Req/I

Work Completed by Each Partner:
Anthony: wrote the cache controller FSM, wrote part of the document
Freddy: wrote the dir controller FSM, wrote part of the document