Lab5 Report

Section6 questions

1. Why are transient states necessary?

In real hardware the state transitions along with their associated actions do not happen instantaneously and atomically due to interconnection network layers. If the protocol has no transient states, it might violate the single-writer, multiple-reader invariant and the data-value invariant. For example, say core 1 executes a read request for block B and the cache is in invalid state. To transition from invalid to shared state, a set of actions needs to be taken. During the time window, a write request from core 2 might have arrived at the directory and been serviced. If Core 1 had transitioned to S state immediately, then the system would have violated the single write/multiple readers invariant. To resolve this issue, transient states are introduced. The block stays in a transient state until all the associated actions of transition are completed and can safely transition to a stable state.

2. Why does a coherence protocol use stall?

Stalls are used to make sure the read/write requests are handled correctly by the system. In most cases, read/write requests cannot be processed if there are pending coherence requests (i.e., the cache block is in transient state). If a cache block is in a transient state, it might not have enough information to respond to read/write requests as the state is not stable. For example, if a load is issued to a block that is transitioning from invalid to shared state, the cache controller can only respond to it once the data has been received in the cache. So, it needs to stall the load instruction.

3. What is a deadlock and how can we avoid it?

Deadlock is the phenomenon where no forward progress is made in a system because of two inter-dependent events. For example, if an event A can only occur after event B completes, and event B can only complete after event A occurs, then none of these two events will ever take place. This cyclic dependence between A and B indefinitely deadlocks the system. The system avoids deadlocks by having separate resources for different message types. For an MSI protocol, different virtual networks are used to make sure that different types of requests are served in different queues. (request message, response message and forward message)

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 is used to acknowledge cache eviction (i.e., replacement). They are used as a response to messages of type **request**. Specifically, Put-Ack is used to respond to PutS-NotLast, PutS-Last, PutM from Owner, and PutM from Non-owner such that the caches can complete their transitions from M or S to I.

5. What determines who is the sender of a data reply to the requesting core? Which are the possible options?

The directory state determines the sender of the data. If the directory is in invalid state, the directory will request the data from the memory and send it to the requesting core. If the directory state is shared, the directory's copy is up to date. So, the directory will forward the copy. If the directory is in modified state, the directory will forward the get message to the owner of the cache block. The owner will send the data.

6. What is the difference between a Put-Ack and an Inv-Ack message?

Put-ack is issued by the directory controller to acknowledge a transition from M or S to I state (i.e., cache eviction). Inv-ack is issued by cache controller to notify the requestor that the sharers have invalidated their copies and have transitioned to invalid state. After receiving all Inv-Ack messages and the data, the requester can proceed to upgrade to M state safely.

<u>Section 5 questions</u>

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

Whenever a cache controller issues an upgrade request (transition from S to M), the directory controller (if it is in the shared state) will respond to it with the data along with the **ackCount** that is equal to the number of shares. Whenever a sharer sends an Inv-Ack, it updates the number of **pendingAcks** in the TBE entry. If the **ackCount** equals the number of **pendingAcks**, the FSM knows that it has received the last Inv Ack and it will send an **Inv_Ack_all** event to the requestors noting that all the sharers have invalidated their copies.

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.

Assume that there are 2 processors and processor 1 (P1) is the owner of block A and the directory is in the modified state. Let's say the processor 2 (P2) issues a read to block A and P1 receives a replacement request and issues a PUTM request. P1 is in MI_A state. Let's say that the request from P2 gets processed first, the directory controller sends the Fwd-GetS request to P1 and adds P1 to the sharers list. P1 receives the Fwd_GetS request and sends the data to the directory and transitions to SI_A state. The directory is now in Shared state. Let's say the PUTM request from P1 is now processed by the directory. Since P1 is the sharer of the data, it is not the owner of the cache block. The event is a PUTM-NonOwner event.

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

Because there is a potential state transition for the directory if it receives a **PUTS** request in the shared state. If there is only one reader, the directory will need to transit into invalid state after acknowledging the **PUTS-Last** request. However, if there are more than one reader, the directory will stay in the shared state after acknowledging a **PUTS** 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.

Assume that there are 2 processors and processor 1 has block A in the shared state and is the only sharer of the block and the directory is also in the shared state for block A. Let's say the processor 1 receives a replacement event and issues a PUTS request and the processor 2 issues a GETM request at the same time. For some reason, the GETM request from processor2 is received by the directory first. The directory controller performs the following actions: it sends data to the processor2, sends invalidate request to processor1 and sets the owner to the requestor and transitions to the **modified state**. After the set of actions has been completed by the directory controller, it receives the **PUTS Last request** from processor1.

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

MSI protocol enforces single writer multiple reader invariant, therefore at any given time for any memory block, there are 3 possible scenarios:

- All caches are in invalid state
- Some caches are in shared state while others are in invalid state
- One cache is in the modified state while others are in invalid state

An invalidation request is initiated only when the directory for a cache block is in the shared state. When the directory is in the shared state, there cannot be any writers for the cache block. Therefore, it is not possible to get an invalidation request for a cache block in a 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 request is generated by a directory controller when it receives a GetS request and the directory is in the modified state. The Fwd-GetS request is only sent to the owner of the block. If the cache block is initially in the shared state and is transitioning to the invalid state, the cache block is not the owner of the cache block. Therefore, it is not possible for the cache controller to get a Fwd-GetS request from a block in 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 as it is time consuming to check if all transitions and states pairs happen. However, we were able to pass configuration with 64 cores and load count of 1 million. In the future, if time permits, we will write a script to parse the output file of the tester and check whether all possible transitions have been exercised.

Protocol modifications

The MSI directory protocol for the cache controller is almost identical to table 8.1 in the textbook. The only difference is that we included the I-fetch event which is equivalent to the load event. We added new transient states for the directory protocol as the textbook did not account for the fact that the data needs to be read from and written back to the memory. We added IS_MD state that waits for the data from the memory before it completes its transition to S state. IM_MD state is added such that the memory data is received before it finishes its transition to M state. We also added MS_D state to make sure the data from the owner of the cache block is received by the directory. After the cache data is received, the directory controller sends a write back request and transitions to MS_A state. After the memory_ack is received, the directory can then transition to S state. We also added the MI_A state to make sure the memory receives the data written back from the owner of the cache block.

The modified state transition table for the directory controller is shown below:

Directory protocol

	GETM	**GETS**	**PUTM**	**PUTM Owner**	**PUTM NotOwner**	**PUTS NotLast**	**PUTS Last**	**PUT Ack**	**Memory Data**	**Memory Ack**	**Data**
I	**v** e qf i / IM MD	**v** ars2 qf i / IS MD			a i	a i	a i				
M	f e i	f ars2 aos **v** i / MS D		lq pl **v** i / MI A	a i	a i	a i				
S	ds2 fwm cs e i / M	ars2 ds i			**k** a i	**k** a i	**k** a i / I				
IS MD	z	z			z	z	z		w d qm / S		
IM MD	z	z		z	z	z	z		w d qm / M		
MS D	z	z		**k** a i	**k** a i	**k** a i	z				lqi pl2 pr / MS A
MS A	z	z		**k** a i	**k** a i	**k** a i	z			w c qm / S	
MI A	z	z			**k** a i	**k** a i	z			la w c qm / I	

The abbreviation for each action can be found in the handout.

Work distribution

Tianyi Xu: implemented lab5, wrote report

Hao Wang: Debug