ECE552: Computer Architecture

University of Toronto
Faculty of Applied Science and Engineering
Lab 5

Prepared By	Danja Papajani (1002136217) Kathryn Tremblay (1003286639)

1.0 Prelab Questions

1. Transient states are required to maintain the single writer/multiple readers (SWMR) invariant in implementations of cache coherence protocols. For example, when a core must perform a set of actions to issue a read request, these cannot be done atomically so there is a window of time between when this request is issued and when it is serviced. If the core that issues the read request transitions immediately to the shared state, but before the required actions are able to complete, another core could issue a write request that could be serviced by the directory. Since a write request is being serviced in the middle of a read request, this violates the SWMR invariant that states a memory location can be read by one or multiple cores, or it can be written by a single core.

2. Stalls are used to prevent local read/write requests to blocks that are in transient states because they are waiting on coherence requests to be serviced. If a block in a core's cache is initially invalid and that core issues a read request, it must go to a transient state while it waits for the data from the directory, and the appropriate read permissions. While in this transient state, any subsequent read or write requests for that block on that core must be stalled because the core has not yet received the required read or write permissions.

3. A deadlock occurs when no progress between cores can be made because they are both waiting on resources or actions from the other core. Deadlocks can be avoided by using Virtual Networks, which are logically separate and have their own queues and buffers. If only one queue is used for all requests, a deadlock may arise in the case where one core is waiting on data from a remote core, but then issues a store for that same block. Since the core is in a transient state, the store request remains in the buffer, but when the remote cache forwards the data, it gets buffered after the store request. This means the store must complete first (it is at the head of the queue), but it requires the data from the remote cache, which is behind it in the queue, so no progress can be made.

4. Put-Ack (WB-Ack) messages are sent by the directory controller to the core that made the request, specifically PUT messages, to indicate that the directory controller has performed the required actions for a write-back. Once the cache controller that issued the PUT has received the corresponding Put-Ack, it is free to move from a transition state to

a stable state since the directory controller has indicated the completion of a request. For example, when a core is the sole owner of a block (so it is in Modified state), and that block is being replaced, the core must write-back the data to the directory so the directory can write-back to memory. Before the block in the core's cache can become invalid, it must receive confirmation from the directory controller that it has in fact written back to memory to ensure that the only up-to-date copy of the block is not overwritten before it is stored elsewhere.

5. The sender of a data reply is indicated in the ResponseMsg structure that has a sender field which holds the MachineID of the node that sent the data. Upon a core issuing a read request, it can receive data back from either the directory, main memory, or a remote core that owns the data.

6. Put-Ack and Inv-Ack differ in which directory sends them and in response to different messages. Specifically, Put-Acks are sent by the directory controller to the requestor cache controller to indicate successful completion of a write-back. Whereas Inv-Acks are sent by cache controllers to the requestor cache controller to indicate that they have invalidated the block that is being modified by the requestor core.

2.0 Methodology

1. Since the Directory keeps track of all sharers of a certain block and sends an AckCount equivalent to this amount, the Cache Controller is able to get the total number of sharers and how many InvAcks to expect. From this, the FSM can know if it has received the last InvAck reply for a TBE.

2. The following events lead to a PutM request from a NonOwner core:
    a. Initially, Core 1 has a block in Modified state.
    b. Another core makes a request to read the block so it sends a GetS request to Directory.
    c. The Directory does not have the block so it forwards the request (Fwd-GetS) to the owner, Core 1. The Directory also adds the previous owner and the current requestor to the list of sharers, and then clears the owner. The Directory transitions from M to MS_D as it waits for the data from Core 1.
    d. Meanwhile, Core 1 receives a Replacement Request for the block, so it issues a write-back (PutM+Data) to the Directory.
    e. Core 1 then receives the Fwd-GetS request while it is in MI_A, and sends the requested block to the Directory and the core that sent the request.
    f. The Fwd-GetS reply from Core 1 arrives, so now the Directory can complete the transition from MS_D to Shared.

g. The Directory then receives the PutM+Data message, but at this point Core 1 is no longer the owner.

3. PUTS is sent when the Cache Controller is replacing a cache block in S state with more than one sharer, while PUTS-Last is sent when there is only one sharer or none at all. When there is more than one sharer and the Directory receives a PUTS, it must keep its cache block because other sharers still exist. When there is only one sharer, the Directory must invalidate it's cache block. Therefore, we need to differentiate between PUTS and PUTS-Last so the Directory can know when to invalidate it's cache block.

4. The following events lead to a PUTS-Last requests for a block in M state:
   a. Block A in M state of Cache 1 issues a GETM and goes to transient state
   b. Block A in S state of Cache 2 sends PUTS-Last and goes to transient state
   c. Directory receives GETM first, goes to M state, and sends Invalidations
   d. PUTS-Last request finally arrives at Directory (M)

5. There can only be one cache that contains a block in M state at any given time. An invalidation request can only be sent by a cache if it is modifying a block, but for this to happen all caches with that block must have first been in S state. Since the block is already in a M state, every other cache's state for this block must be in the I state. Therefore, this negates the possibility for an invalidation to that block to happen, because it would require other processors being in S state.

6. In order to be in SI_A state, the cache block must have been in either S or M state previously and relinquished it's ownership of the block. When the Directory receives a GETS request, it will send a Fwd-GetS to the owner of the block. Since the said block in the Cache Controller is in SI_A state, it is not the owner and therefore it will be impossible for it to receive a Fwd-GetS.

7. Although the system would signal when an unexpected transition occurs, it does not signal whether all transitions have been entered at some point. Therefore, it is not possible to do an "exhaustive" testing of our system, but rather an "extensive" one. To ensure the random tester had exercised an extensive amount of transition states, we would test progressively by increasing the number of cores, increasing number of instructions (loads), and decreasing cache size. As errors occurred, we would review and handle that transition. Increasing the number of cores and instructions increases the different types and frequency of transitions in the system. More cores will have to speak to each other, along with higher traffic due to the increase of instructions, which increases the probability of revealing transitions that wouldn't have otherwise occurred including their

associated errors. This provided an extensive testing of our system, although not entirely exhaustive.

3.0 Table Documenting Protocol Modifications

Description of additional directory controller transient states:

- *IS_MD*: Upon moving from Invalid to Shared, must wait for data from memory.
- *IM_MD*: Upon moving from Invalid to Modified, must wait for data from memory.
- *MI_A*: Upon moving from Modified to Invalid, must wait for ack from memory indicating write-back has completed.
- *MS_A*: Upon moving from MS_D to Shared, must wait for ack from memory indicating write-back has completed.

Table 1. Additional Directory Controller transition states and corresponding actions.

	Memory Data	Memory Ack
IS_MD	Send data retrieved from memory to remote cache, add requester to sharers / S	
IM_MD	Send data retrieved from memory to remote cache / M	
MI_A		Send Put-Ack to Req / I
MS_A		/ S

Note 1: The entry points of the transient state are not shown in the table but are mentioned here.

- State IS_MD is entered when in the Invalid state and GetS is received.
- State IM_MD is entered when in the Invalid state and GetM is received.
- State MI_A is entered when in the Modified state and PutM_Owner is received.
- State MS_A is entered when in MS_D and Data is received.

Note 2: Messages (columns of the table) that result in a stall or are not possible regardless of the current state, have been removed to present a more concise representation of the added transient states.

The states IS_MD, IM_MD, and MS_A stall upon receiving GetS, GetM, PutS-NotLast, PutS-Last, and PutM+data from Non-Owner.
The state MI_A stalls upon receiving GetS, GetM, PutS-NotLast, PutS-Last, PutM+data from Owner, and PutM+data from Non-Owner.

Brief Statement of Work

KC Tremblay & Danja Papajani: worked on all deliverables (½ of workload each)