COL718 - Assignment 3

Bommakanti Venkata Naga Sai Aditya, 2019CS50471
Chapala Sriram Varma, 2019CS50426

Implementing Memory Models:

PLSC (fr, ws) will be guaranteed by the underlying memory model because stores and loads are atomic with respect to a single location. So, we just need to ensure program orders for different memory models.

Sequential Consistency:

Orders relaxed: rfi
Orders not relaxed: W-R, R-W, R-R. W-W

Ensuring W-R, R-W, R-R:
We modified LSQ such that these orders are not relaxed. In LSQ, we modified the method of forwarding. We forward a value from a store to a load only if they are consecutive (rfi is relaxed) i.e., load appears immediately after store. If they are not, we won't forward. Because of this, W-R, R-W, R-R orders are not relaxed.

Ensuring W-W:
We issue stores at commit time. But it can happen that the later committed store might have completed earlier because of routing in NoC. To avoid this, we need explicit acknowledgement from the memory system (this acts like a memory fence). We commit a store, then we wait for acknowledgement from the Memory system. Once we receive acknowledgement, we proceed further. Until then, no other memory instruction commits. We modified Cache.java to get the acknowledgement back from the memory system.

Changes made in Cache.java:

```
protected void cacheHit(long addr, RequestType requestType, CacheLine cl,
      AddressCarryingEvent event) {
   hits++;
//System.out.println("Yes the hits arwe"+hits);
   vmmHitCounter.inc(1);
   guestHitCounter.inc(1);
   noOfRequests++;
   noOfAccesses++;

   if (requestType == RequestType.Cache_Read) {
      sendAcknowledgement(event);
   } else if (requestType == RequestType.Cache_Write) {
      if(this.writePolicy == WritePolicy.WRITE_THROUGH) {
         sendRequestToNextLevel(addr, RequestType.Cache_Write);
      }

      if( (cl.getState() == MESI.SHARED || cl.getState() == MESI.EXCLUSIVE)  &&
            (mycoherence!=null)) {
         handleCleanToModified(addr, event);
      }
      AddressCarryingEvent ack = new AddressCarryingEvent(
            event.getEventQ(), 0, containingMemSys, this,
            RequestType.Cache_Write,
            event.getAddress());
      if(this.isTopLevelCache()){
         sendAcknowledgement(ack);
      }
   } else {
      misc.Error.showErrorAndExit("cache hit unknown event type\n" + event + "\ncache : " + this);
   }
}
```

On a cache hit for a cache_write event, if the cache is a top Level Cache (L1 cache) then we create an ‘AddressCarryingEvent’ for acknowledgement from this cache to the ContainingMemorySystem which forwards it to LSQ. Hence, we can send an ACK to LSQ

```
protected void processEventsInMSHR(long addr) {
   LinkedList<AddressCarryingEvent> missList = mshr.removeEventsFromMSHR(addr);
   AddressCarryingEvent writeEvent = null;

   for(AddressCarryingEvent event : missList) {
      switch(event.getRequestType()) {
         case Cache_Read: {
            sendAcknowledgement(event);
            break;
         }

         case Cache_Write: {
            CacheLine cl = accessValid(addr);

            if(cl!=null) {
               updateStateOfCacheLine(addr, MESI.MODIFIED);
               writeEvent = event;
            } else {
               misc.Error.showErrorAndExit("Cache write expects a line here : " + event);
            }
            AddressCarryingEvent ack = new AddressCarryingEvent(
               event.getEventQ(), 0, containingMemSys, this,
               RequestType.Cache_Write,
               event.getAddress());

            if(this.isTopLevelCache()){
               sendAcknowledgement(ack);
            }
            break;
         }
```

While processing events in MSHR, if the event is a cache write and if we are at top level cache, we send an acknowledgement to the Containing Memory System. We are sending Ack from only top level cache to eliminate duplicates.

```
public void sendAcknowledgement(AddressCarryingEvent event) {
    RequestType returnType = null;
    if(event.getRequestType()==RequestType.Cache_Read) {
        returnType = RequestType.Mem_Response;
    }
    else if(event.getRequestType()==RequestType.Cache_Write){
        returnType = RequestType.St_Mem_Response;
    }
    else {
        misc.Error.showErrorAndExit("sendAcknowledgement is meant for cache read operation only : " + event);
    }

    AddressCarryingEvent memResponseEvent = new AddressCarryingEvent(
            event.getEventQ(), 0, event.getProcessingElement(),
            event.getRequestingElement(), returnType,
            event.getAddress());

    sendEvent(memResponseEvent);
    noOfResponsesSent++;

    // if(ArchitecturalComponent.getCores()[1].getNoOfInstructionsExecuted()
    // > 6000000l)
    // System.out.println("sending mem response from " +
    // event.getProcessingElement() + " to " + event.getRequestingElement()
    // + " for addr : " + event.getAddress());
}
```

To send an acknowledgement of Cache write, we send a response message from the memory system with request type St_Mem_Response to the Containing Memory System.

Changes in OutOrderCoreMemorySystem.java:

```
// Unified cache scenario
if(iCache==l1Cache)
{
    containingExecEngine.getFetcher().processCompletionOfMemRequest(address);
    lsqueue.handleMemResponse(address);
}

//if response comes from iCache, inform fetchunit
else if(memResponse.getRequestingElement() == iCache)
{
    containingExecEngine.getFetcher().processCompletionOfMemRequest(address);
}

//if response comes from l1Cache, inform memunit
else if(memResponse.getRequestingElement() == l1Cache)
{
    if(memResponse.getRequestType() == RequestType.Mem_Response){
        lsqueue.handleMemResponse(address);
    }
    else if(memResponse.getRequestType() == RequestType.St_Mem_Response){
        lsqueue.handleMemResponse1(address);
    }
}
```

Based on the response from the Memory System, we send the response to LSQ. If the request type is Mem_Response (response for a load) then we send it to handleMemResponse, if it is a response to store we send it to handleMemResponse1.

Note that before adding store to Instruction window, we haven't marked it as executed (IWEntry.java). A store/load will be marked as executed after we get an acknowledgement from the Memory System.

Changes in LSQ.java:

```
protected LSQSearchStatus loadResolve(int index, LSQEntry entry)
{
   if (!entry.isValid())
      misc.Error.showErrorAndExit(" 01 Invalid entry forwarded");

   int tmpIndex;

   if (entry.getIndexInQ() == head)
      return LSQSearchStatus.ShouldGoToL1;
   else
      tmpIndex = decrementQ(index);


   LSQEntry tmpEntry = lsqueue[tmpIndex];
   if (tmpEntry.getType() == LSQEntry.LSQEntryType.STORE)
   {
      if (tmpEntry.getAddr() == entry.getAddr())
      {
         if (tmpEntry.isValid())
         {
            // Successfully forwarded the value
            entry.setForwarded(true);
            NoOfForwards++;
            if (entry.getRobEntry() != null && !entry.getRobEntry().getExecuted())
               ((OutOrderCoreMemorySystem)containingMemSys).sendExecComplete(entry.getRobEntry());
            return LSQSearchStatus.Forwarded;
         }
         else
         {
            //found a store to the same address, but it is not valid yet
            return LSQSearchStatus.CanBeForwarded;
         }
      }
   }
```

In resolving a load, we just see the previous index and if it is a store to the same address then we forward the value to the load. If not, it should go to the memory at commit time.

```
protected void storeResolve(int index, LSQEntry entry)
{
   if (!entry.isValid())
      misc.Error.showErrorAndExit(" 02 Invalid entry forwarded");

   int sindex = incrementQ(index);


   LSQEntry tmpEntry = lsqueue[sindex];

   if (tmpEntry.getType() == LSQEntry.LSQEntryType.LOAD)
   {
      if(tmpEntry.getAddr() == entry.getAddr())
      {
         if (tmpEntry.isValid() && !tmpEntry.isForwarded())
         {
            tmpEntry.setForwarded(true);
            if (tmpEntry.getRobEntry() != null && !tmpEntry.getRobEntry().getExecuted())
               ((OutOrderCoreMemorySystem)containingMemSys).sendExecComplete(tmpEntry.getRobEntry());

            NoOfForwards++;
         }
      }
   }
}
```

For resolving a store, we just see the next index and if it is a load to the same address then we forward the value to the store.

```
public void handleMemResponse1(long address)
{
   LSQEntry lsqEntry = null;

   int index = head;
   for(int i = 0; i < curSize; i++)
   {
      lsqEntry = lsqueue[index];

      if(lsqEntry.getType() == LSQEntryType.STORE
            && (lsqEntry.getAddr() == address) && !lsqEntry.isRemoved())
      {
         if (!lsqEntry.isValid())
         {
            index = (index+1)%lsqSize;
            continue;
         }
         lsqEntry.getRobEntry().setExecuted(true);
         break;
      }

      index = (index+1)%lsqSize;
   }
}
```

Once we get an acknowledgement for a store, we mark the corresponding ROB entry as executed. Now, this will be removed from ROB.

We modified the handleCommitsFromROB() function such that it commits only after receiving a response from the Memory System.

```
for(; ; i = (i+1)%lsqSize)
{
   LSQEntry tmpEntry = lsqueue[i];
   if(tmpEntry.isIssued() && !tmpEntry.getRobEntry().getExecuted()){
      break;
   }
   // if it is a store, send the request to the cache
   if(tmpEntry.getType() == LSQEntry.LSQEntryType.STORE)
   {
      if(tmpEntry.isValid() == false)
      {
         misc.Error.showErrorAndExit("store not ready to be committed");
      }

      boolean requestIssued =
            containingMemSys.issueRequestToL1Cache(RequestType.Cache_Write,
                  tmpEntry.getAddr());
      tmpEntry.setIssued(true);
      if(requestIssued == false)
      {
         event.addEventTime(1);
         event.getEventQ().addEvent(event);
         break; //removals must be in-order : if u can't commit the operation at the head, u can't commit the ones that follow it
      }
      else if(!tmpEntry.getRobEntry().getExecuted()){
         break;
      }
      else
      {
         if(head == tail)
         {
            head = tail = -1;
         }
         else
         {
            this.head = this.incrementQ(this.head);
         }
         this.curSize--;
         tmpEntry.setRemoved(true);
      }
   }
```

Here, if the head instruction is issued to the memory system but not executed (i.e., not received response) then we won't commit. If it is not issued, then the instruction will be issued and we will wait till it executes.

Total Store Order:

Orders relaxed: rfi, W-R
Orders not relaxed: R-W, R-R, W-W

We are relaxing W-R order. So, if there is a store and a load to the same address 'x' and if there are only interleaving 'stores' in between them to different addresses only then we can forward the value from the store to load. Otherwise we can't forward. Other orders R-W, R-R. W-W are ensured in a way similar to SC.

Changes to loadResolve()

```
if (entry.getIndexInQ() == head)
   return LSQSearchStatus.ShouldGoToL1;
else
   tmpIndex = decrementQ(index);

while(true)
{
   LSQEntry tmpEntry = lsqueue[tmpIndex];
   if (tmpEntry.getType() == LSQEntry.LSQEntryType.STORE)
   {
      if (tmpEntry.getAddr() == entry.getAddr())
      {
         if (tmpEntry.isValid())
         {
            // Successfully forwarded the value
            entry.setForwarded(true);
            NoOfForwards++;
            if (entry.getRobEntry() != null && !entry.getRobEntry().getExecuted())
               ((OutOrderCoreMemorySystem)containingMemSys).sendExecComplete(entry.getRobEntry());
            return LSQSearchStatus.Forwarded;
         }
         else
         {
            //found a store to the same address, but it is not valid yet
            return LSQSearchStatus.CanBeForwarded;
         }
      }
   }
   else{
      break;
   }
   if(tmpIndex == head)
      break;
   tmpIndex = decrementQ(tmpIndex);
}
return LSQSearchStatus.ShouldGoToL1;
```

Here, we added a break statement if we see a load while resolving a load. This will preserve R-R R-W ordering.

Changes to storeResolve()

```
protected void storeResolve(int index, LSQEntry entry)
{
   if (!entry.isValid())
      misc.Error.showErrorAndExit(" 02 Invalid entry forwarded");

   int sindex = incrementQ(index);

   while (true)
   {
      LSQEntry tmpEntry = lsqueue[sindex];

      if (tmpEntry.getType() == LSQEntry.LSQEntryType.LOAD)
      {
         if(tmpEntry.getAddr() == entry.getAddr())
         {
            if (tmpEntry.isValid() && !tmpEntry.isForwarded())
            {
               tmpEntry.setForwarded(true);
               if (tmpEntry.getRobEntry() != null && !tmpEntry.getRobEntry().getExecuted())
                  ((OutOrderCoreMemorySystem)containingMemSys).sendExecComplete(tmpEntry.getRobEntry());

               NoOfForwards++;
            }
         }
         else{
            break;
         }
      }
```

Here, we added a break statement if we see a store while resolving a store.

Weak Ordering:

Orders relaxed: rfi, W-R, R-W, R-R, W-W
Orders not relaxed: rfe

Results:
SC:
BodyTrack Benchmark:
Overall Stats:

Total Cycles taken = 73134

Total IPC = 2.4209 in terms of micro-ops
Total IPC = 2.4116 in terms of CISC instructions

Per Core Stats:

core = 1
Pipeline: outOfOrder
instructions executed = 8
cycles taken= 73134 cycles
IPC = 0.0001 in terms of micro-ops
IPC = 0.0295 in terms of CISC instructions
core frequency = 3200 MHz
time taken = 22.8544 microseconds

number of branches = 0
number of mispredicted branches = 0
branch predictor accuracy = NaN %

predictor type = TAGE
PC bits = 8
BHR size = 16
Saturating bits = 2

core = 2

Pipeline: outOfOrder
instructions executed = 50452
cycles taken= 73134 cycles
IPC = 0.6899 in terms of micro-ops
IPC = 0.7133 in terms of CISC instructions
core frequency = 3200 MHz
time taken = 22.8544 microseconds

number of branches = 2407
number of mispredicted branches = 10
branch predictor accuracy = 99.5845 %

predictor type = TAGE
PC bits = 8
BHR size = 16
Saturating bits = 2

core = 3
Pipeline: outOfOrder
instructions executed = 51829
cycles taken= 73134 cycles
IPC = 0.7087 in terms of micro-ops
IPC = 0.7257 in terms of CISC instructions
core frequency = 3200 MHz
time taken = 22.8544 microseconds

number of branches = 2472
number of mispredicted branches = 10
branch predictor accuracy = 99.5955 %

predictor type = TAGE
PC bits = 8
BHR size = 16
Saturating bits = 2

core = 4
Pipeline: outOfOrder

instructions executed = 49583
cycles taken= 73134 cycles
IPC = 0.6780 in terms of micro-ops
IPC = 0.6989 in terms of CISC instructions
core frequency = 3200 MHz
time taken = 22.8544 microseconds

number of branches = 2365
number of mispredicted branches = 10
branch predictor accuracy = 99.5772 %

predictor type = TAGE
PC bits = 8
BHR size = 16
Saturating bits = 2

core = 5
Pipeline: outOfOrder
instructions executed = 2380
cycles taken= 73134 cycles
IPC = 0.0325 in terms of micro-ops
IPC = 0.0619 in terms of CISC instructions
core frequency = 3200 MHz
time taken = 22.8544 microseconds

number of branches = 113
number of mispredicted branches = 0
branch predictor accuracy = 100.0000 %

predictor type = TAGE
PC bits = 8
BHR size = 16
Saturating bits = 2

core = 6
Pipeline: outOfOrder
instructions executed = 7375

cycles taken= 73134 cycles
IPC = 0.1008 in terms of micro-ops
IPC = 0.1301 in terms of CISC instructions
core frequency = 3200 MHz
time taken = 22.8544 microseconds

number of branches = 352
number of mispredicted branches = 3
branch predictor accuracy = 99.1477 %

predictor type = TAGE
PC bits = 8
BHR size = 16
Saturating bits = 2

core = 7
Pipeline: outOfOrder
instructions executed = 27
cycles taken= 73134 cycles
IPC = 0.0004 in terms of micro-ops
IPC = 0.0252 in terms of CISC instructions
core frequency = 3200 MHz
time taken = 22.8544 microseconds

number of branches = 2
number of mispredicted branches = 0
branch predictor accuracy = 100.0000 %

predictor type = TAGE
PC bits = 8
BHR size = 16
Saturating bits = 2

Memory Stats:
[Per core statistics]

core = 0

Memory Requests = 97
Loads = 65
Stores = 32
LSQ forwardings = 1

core = 1
Memory Requests = 18346
Loads = 16515
Stores = 1831
LSQ forwardings = 0

core = 2
Memory Requests = 18514
Loads = 16666
Stores = 1848
LSQ forwardings = 0

core = 3
Memory Requests = 18335
Loads = 16505
Stores = 1830
LSQ forwardings = 0
.
core = 4
Memory Requests = 1472
Loads = 1324
Stores = 148
LSQ forwardings = 0

core = 5
Memory Requests = 18487
Loads = 16640
Stores = 1847
LSQ forwardings = 0

core = 6
Memory Requests = 2302

Loads = 2071
Stores = 231
LSQ forwardings = 0

core = 7
Memory Requests = 48
Loads = 38
Stores = 10
LSQ forwardings = 0