EECS 570 Midterm Exam
Winter 2018

Name: ___________________________        Uniqname: ____________________

Sign the honor code:

I have neither given nor received aid on this exam nor observed anyone else doing so.

_______________________________

Scores:

<table>
<thead>
<tr>
<th>#</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/ 25</td>
</tr>
<tr>
<td>2</td>
<td>/ 8</td>
</tr>
<tr>
<td>3</td>
<td>/ 20</td>
</tr>
<tr>
<td>4</td>
<td>/ 20</td>
</tr>
<tr>
<td>5</td>
<td>/ 12</td>
</tr>
<tr>
<td>Total</td>
<td>/ 85</td>
</tr>
</tbody>
</table>

NOTES:
- Closed book, closed notes.
- Calculators are allowed, but no PDAs, portables, cell phones, etc.
- Don’t spend too much time on any one problem.
- You have 85 minutes for the exam.
- There are 11 pages in the exam (including this one). Please ensure you have all pages.
- Be sure to show work and explain what you’ve done when asked to do so.
1) Short Answer [25 points]

a) Consider the following snooping-bus based multiprocessor system where each core has a private L1 cache and a private L2 cache. Compare and contrast the performance overhead and implementation complexity of snoop-based coherence protocol for the two cases:

(i) Inclusive caches, and
(ii) Exclusive caches

Exclusive caches increase complexity, because an address residing in the L1 cache does not necessarily reside in the L2 cache. This means, a BusInv packet on the bus will have to snoop all the cache hierarchies (L1 and L2, in this case) to find a matching address.

If the caches were inclusive, the BusInv packet will only need to snoop the L2 cache, because there cannot be an address that is present in the L1 cache, but not in the L2 cache.

On the other hand, exclusive cache have better cache utilization than inclusive caches. It is not necessary to check and invalidate a cache line in L1, when a block is evicted from L2.

b) Briefly explain the technique used in GPUs to avoid pipeline stalls due to long latency accesses to memory. Comment on whether a similar technique would be useful in a multi-core CPU to hide long memory access latencies and the trade-offs involved. [4 points]

GPUs map multiple warps and context switch between them to hide memory access latencies. One could map multiple threads to the same CPU but that would require larger registers and additional complexity, which can increase latency. Because CPUs optimize for latency, as opposed to GPUs (which optimize for throughput), this may be undesirable.
c) Consider the following program execution involving 2 threads.

\[ a = 0; \]

Thread 1

\[ a++; \]
\[ \text{while}(a < 2); \]

Thread 2

\[ a++; \]
\[ \text{while}(a < 2); \]

Which of the following statement(s) is/are correct? **Justify your answer. [3 points]**

i) Both threads will never finish execution

ii) Both threads will always finish execution

iii) At least one of the threads will always finish execution

iv) Either both threads will finish, or none of them will finish execution

iv) is correct. The threads may not finish execution because `a++` is not guaranteed to be atomic. It can be split into a read, a modify and a store operation. If both threads read `a = 0` and store `a = 1` just before the `while()` loop, both threads will be stuck executing the `while()`. On the other hand, the threads would finish execution if the `a++` statements get executed one after the order, after which "a" becomes 2, and then the `while()` statements get executed. Coherence would ensure that both threads finish in this case.

d) Under what circumstances might a test-and-test-and-set lock be preferable to an MCS lock? [2 points]

When the lock is hardly contended for, the TTS lock will have lower overhead for lock acquisition and release compared to the MCS lock.

e) Can you reduce work through parallelization? If yes, how? If not, why not? [2 points]

Work can not be reduced through parallelization. Computational work can be distributed between threads, but not eliminated.

f) Mention two advantages and two disadvantages of directory-based coherence over snooping bus-based coherence. [4 points]

**Advantages:**

1. Snooping coherence is not scalable due to the centralized bus. Directory coherence relieves this through point-to-point communication.
2. Snooping coherence incurs high snooping bandwidth due to tag lookup on every processor's cache when any processor sends a request. Directory coherence resolves this by communicating with only the nodes that care.
Disadvantages:
1. Directory protocols can increase coherence request/response latency.
2. A directory has to track the sharer’s list and the owner which adds storage overheads.

g) Describe the sharing pattern in a program for which a write-update protocol would perform better than a write-invalidate based protocol. [2 points]

One producer and multiple consumers.

h) Calculate the work and depth of the following directed acyclic graph. Also, draw the critical path(s) in the system. [4 points]
Work = 40, depth = 19. There are two critical paths in this DAG, shown in red and blue below.
2) Synchronization Primitives [8 points]

Implement a barrier using the Message-Passing Interface. MPI functions that you can use are defined below. Assume that there are NO shared variables. Assume that a program always has a thread with the ID 0. A program can have one or more threads.

// send msg with contents value to thread dstThread
void blockingSend(int dstThread, int value);

// recv message from srcThread. Upon return, value is populated.
void blockingRecv(int srcThread, int *value);

int getThreadId(); // returns the id of the calling thread

int getNumThreads(); // returns the number of threads in the program

Complete the function below.

```c
void barrier_wait() {
    int myId = getThreadId();
    int N = getNumThreads();
    int i;
    // Assume thread ID 0 to be the root thread
    if (myId > 0) {
        // Send notification to root that this thread has reached the barrier
        blockingSend(0, 0);
        // Block awaiting notification from the root
        blockingRecv(0, 0);
    } else {
        // Block until this root thread receives notifications from N-1 other threads
        for(i = 1; i < N; ++i) {
            blockingRecv(i, NULL);
        }
        // All threads have arrived: signal everyone
        for(i = 1; i < N; ++i) {
            blockingSend(i, NULL);
        }
    }
}
```

3) Transactional Memory [20 points]

(a) Can transactions be used to replace barrier synchronizations? Justify. [2 points]

No. Transactions provide mutual exclusion. They serve a similar purpose as lock-unlock.
A barrier is used to ensure a set of threads reach a common point in a program execution.

(b) Provide an example where two transactions deadlock. [3 points]

\[
\begin{align*}
X &= Y = 0; \\
\begin{align*}
Y &= 1; \\
\text{while}(!(Y); \\
X &= 1; \\
\end{align*}
\end{align*}
\]

(c) Consider two versions of a program, one written using transactions, and another using locks. Assume that both are carefully optimized by expert programmers. Which of the two versions do you expect to perform better, and why? Assume that the runtime overhead of conflict check and versioning is similar to the overhead of lock operations. [3 points]

Lock version is likely to perform better.

Using fine-grained locks, a programmer should be able to expose all available parallelism. Transactions cannot expose more parallelism than a version with hand-crafted fine-grained locks. However, it could suffer from additional performance loss due to rollbacks when the runtime optimistically executes two conflicting transactions in parallel.

But well optimized fine-grained locks are hard to engineer in practice. Transactions can attain performance of fine-grained lock implementation, but they are as easy to use as a single coarse-grained lock.

(d) Two transactions are said to conflict if they execute memory accesses to the same location, and at least one of those accesses is a write. Two transactions are serializable if they appear to have executed one after the other.

Is it true that if two concurrent transactions conflict, then they are not serializable? Justify your answer. Use examples if necessary. Write your answer in the next page. [3 points]

Certain conflicting transactions are serializable. Here are a few examples (one example should be adequate).

\[
\begin{align*}
\begin{align*}
X &= 1 \\
\end{align*}
\end{align*}
\]

begin begin
    = X
    X = 1
end end

The following transactions are serializable as long as the statement I does not execute between the two statements in the second transaction.

begin begin
    I: = X
    X = 1

    X = 2
end end

(e) A hardware transactional memory (HTM) system can support eager conflict detection by extending the coherence protocol. Would a lazy conflict detection have any advantages in a HTM? Justify your answer. Use examples if necessary. [3 points]

Lazy conflict detection can help avoid rollbacks in instances where it is possible to determine that conflicting transactions are serializable.

(f) Describe at least three unique challenges that may arise in supporting HTM in a GPU architecture, when compared to a multi-core CPU. [6 points]

1. Scalable conflict detection is a challenge as a GPU executes thousands of threads at any given time.
2. A GPU may not have coherence support, which makes it challenging to support conflict detection.
3. Several warps are mapped to a core. Each warp has several threads. It is harder to track working set in a core's private cache. In a CPU where only one thread is mapped to a core, however, the working set of a transaction is easily maintained using one pair of read/write bits per cache line.
4. Register files are larger. Taking a full snapshot of entire register file at the beginning of a transaction may be expensive.
5. If a transaction in a thread that belongs a warp needs to be rolled back, but other threads do not have a conflict, then it can lead to additional control-flow divergences between threads in a warp.
6. Caches have generally poorer temporal locality in GPUs than CPUs, and GPU threads have a larger data footprint. Therefore, unbounded transactions (transactions whose working set does not fit within a cache) may be more common than in CPUs.
4) **Coherence Protocols [20 points]**

Xerox’s Dragon snoop-based coherence protocol has the following four states.

- **E (Exclusive):** block is present in only this processor’s cache, and not dirty.
- **M (Modified):** block is present in only this processor’s cache, and is dirty.
- **Sc (Shared-clean):** two or more caches hold this block, and this processor is not the last one to write the block.
- **Sm (Shared-modified):** two or more caches hold this block, and this processor is the last one to write the block. This processor is responsible to update main memory when the block is evicted.

Dragon is a **write-update** protocol, which means that when a cache block is modified, the copies of the same block in other caches are updated, instead of being invalidated.

a) Complete the following coherence state transition diagram for the Dragon protocol using **only the action/reaction symbols given below. [17 points]**

Three labelled transitions and several unlabelled state transitions have been completed for you. Label the remaining unlabelled edges. You may need to add more edges between states. **Superfluous edges will incur a -1 point penalty.**

Assume a **shared line (S)** (a hardware wire connected to all processors) that indicates whether a cache block, whose addressed is broadcasted on the bus, is cached in more than one processor. Use “S” to annotate a transition when there are sharers present, and “!S” for when there are no sharers present.

Do not introduce additional states. Ignore transient states and evictions.

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrRd</td>
<td>Processor Read</td>
</tr>
<tr>
<td>PrWr</td>
<td>Processor Write</td>
</tr>
<tr>
<td>PrRdMiss</td>
<td>Processor Read Miss</td>
</tr>
<tr>
<td>PrWrMiss</td>
<td>Processor Write Miss</td>
</tr>
<tr>
<td>BusRd</td>
<td>Bus Read</td>
</tr>
<tr>
<td>BusUpd</td>
<td>Bus Update</td>
</tr>
<tr>
<td>BusReply</td>
<td>Bus Reply</td>
</tr>
<tr>
<td>Update</td>
<td>Update own cache block</td>
</tr>
<tr>
<td>Flush</td>
<td>Place the block on the bus and write to memory</td>
</tr>
</tbody>
</table>
The solution is as follows:

We will accept solutions where a BusReply is used instead of a flush.
b) The Dragon protocol allows the cache blocks in the Sc state to be replaced silently without any bus activity. What optimization to the coherence protocol can be made if a broadcast was made to let other caches know that a Sc block is being replaced? [3 points]

Other caches could test the shared line and move to state E if there are no other sharers.
5) **Directory Coherence [12 points]**

Below are reproduced the state machines for the MOSI directory-based coherence protocol described in the Sorin, Hill, & Wood Synthesis lecture. Six table entries have been replaced with annotations (I) – (VI). You must fill in these blanks with an appropriate “action/state transition” pair that preserves correctness. Use the space below the tables to write your answers. [12 points]

**MOSI Directory Protocol—Cache Controller:**

<table>
<thead>
<tr>
<th></th>
<th>Send/Store</th>
<th>Replacement</th>
<th>Ack From/To Owner</th>
<th>Ack Count From Dir</th>
<th>Inv-Ack</th>
<th>Last-Ack</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>send GetS to Dir/IM</td>
<td>send GetM to Dir/IM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IM</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>inv-ack</td>
</tr>
<tr>
<td>IM</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>inv-ack</td>
</tr>
<tr>
<td>S</td>
<td>hit</td>
<td>send GetM to Dir/SM</td>
<td>send PutS to Dir/S</td>
<td>send Inv-Ack to Req</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SM</td>
<td>hit</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>inv-ack</td>
</tr>
<tr>
<td>M</td>
<td>hit</td>
<td>hit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>stall</td>
<td>stall</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>hit</td>
<td>send GetM to Dir/OM</td>
<td>send data to Req</td>
<td>send data to Req</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OM</td>
<td>hit</td>
<td>stall</td>
<td>send data to Req</td>
<td>send data to Req</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OM</td>
<td>hit</td>
<td>stall</td>
<td></td>
<td>send data to Req</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OI</td>
<td>stall</td>
<td>stall</td>
<td>send data to Req</td>
<td>send data to Req</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SI</td>
<td>stall</td>
<td>stall</td>
<td>send data to Req</td>
<td>send Inv-Ack to Req</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PI</td>
<td>stall</td>
<td>stall</td>
<td>send data to Req</td>
<td>send Inv-Ack to Req</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MOSI Directory Protocol—Directory Controller:

<table>
<thead>
<tr>
<th>GetS</th>
<th>GetM from</th>
<th>GetM from</th>
<th>PutS-</th>
<th>PutS-</th>
<th>PutM+data</th>
<th>PutM+data</th>
<th>PutO+data</th>
<th>PutO+data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Owner</td>
<td>NonOwner</td>
<td>NotLast</td>
<td>Last</td>
<td>from Owner</td>
<td>from NonOwner</td>
<td>from Owner</td>
<td>from NonOwner</td>
</tr>
<tr>
<td>I</td>
<td>send Data to Req, add Req to Sharers/S</td>
<td>send Data to Req, set Owner to Req/M</td>
<td>send Put-Ack to Req</td>
<td>send Put-Ack to Req</td>
<td>send Put-Ack to Req</td>
<td>send Put-Ack to Req</td>
<td>send Put-Ack to Req</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>send Data to Req, add Req to Sharers</td>
<td>send Data to Req, send Inv to Sharers, set Owner to Req, clear Sharers/M</td>
<td>remove Req from Sharers, send Put-Ack to Req/I</td>
<td>remove Req from Sharers, send Put-Ack to Req</td>
<td>remove Req from Sharers, send Put-Ack to Req</td>
<td>remove Req from Sharers, send Put-Ack to Req</td>
<td>remove Req from Sharers, send Put-Ack to Req</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>forward GetS to Owner, add Req to Sharers</td>
<td>send Ack-COUNT to Req, send Inv to Sharers, clear Sharers/M</td>
<td>forward GetM to Owner, send Inv to Sharers, set Owner to Req, clear Sharers, send Ack-COUNT to Req/M</td>
<td>remove Req from Sharers, send Put-Ack to Req</td>
<td>remove Req from Sharers, send Put-Ack to Req</td>
<td>remove Req from Sharers, copy data to mem, send Put-Ack to Req, clear Owner/S</td>
<td>remove Req from Sharers, copy data to mem, send Put-Ack to Req, clear Owner/S</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>forward GetS to Owner, add Req to Sharers/O</td>
<td>forward GetM to Owner, set Owner to Req</td>
<td>send Put-Ack to Req</td>
<td>send Put-Ack to Req</td>
<td>copy data to mem, send Put-Ack to Req, clear Owner/O</td>
<td>send Put-Ack to Req</td>
<td>send Put-Ack to Req</td>
<td></td>
</tr>
</tbody>
</table>

Write your answers below:

(I)  
Send Inv-Ack to Req / IM<sup>ab</sup>

(II)  
Send PutM + Data to Dir / MI<sup>a</sup>

(III)  
Send Data to Req / O

(IV)  
Send Data to Req / OI<sup>a</sup>

(V)  
Send PutO + Data to Dir / OI<sup>a</sup>

(VI)  
Send Data to Req / IM<sup>ab</sup>