EECS 570 Final Exam - SOLUTIONS
Winter 2015

Name: ____________________________________    unique name: _______________

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>/ 21</td>
</tr>
<tr>
<td>2</td>
<td>/ 32</td>
</tr>
<tr>
<td>3</td>
<td>/ 12</td>
</tr>
<tr>
<td>4</td>
<td>/ 35</td>
</tr>
<tr>
<td>Total</td>
<td>/</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 about 90 minutes for the exam (avg. 22 minutes per problem).
- There are 12 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) Chip Multiprocessing/Multithreading [21 points]
(Thanks to Babak Falsafi for several of these questions).

a) Briefly describe each of the following. State how each form of multithreading improves performance. [9 points]

i) switch-on-miss multithreading

Upon long latency events, such as cache misses, the CPU switches to execute a different thread. Hence, the cache miss latency can be hidden by other useful work.

ii) fine-grain multithreading

Each cycle, the processor switches to a different thread, such that the various threads are interleaved over the pipeline stages. This approach can hide all true data dependencies between stages, avoiding pipeline bubbles, and hide long cache hit latencies.

iii) simultaneous multithreading

In an out-of-order processor, multiple independent instruction streams are fed to the out-of-order execution unit, which executes them concurrently. Each cycle the processor front end may fetch from one or more of the concurrently-executing threads. The threads compete for execution resources, and can improve functional unit utilization and hide one another's stalls.
b) Would OLTP throughput increase, decrease, or not change when using a simultaneously multithreaded core as compared to a superscalar core? Explain. [6 points]

OLTP workloads tend to have many stalls due to last-level cache misses. SMT will allow one thread to make progress on a core while other threads are stalled on misses. Hence, throughput is likely to increase, to a point, with SMT. This observation is precisely the motivation for Sun's Niagara processor design.

c) What is the key shortcoming in superscalar that speculatively-threaded designs (e.g., MultiScalar) address? Explain. [6 points]

Superscalar processors can find independent instructions only within small instruction windows, and are limited by their inability to speculate accurately across unpredictable branches. By speculating on control flow at coarser granularity, speculative threading allows greater ILP without being restricted by the unpredictable branch hammocks, which can be captured entirely within a thread and handled via conventional OoO mechanisms.
2) **Interconnection networks [32 points]**

a) Briefly state an advantage and a disadvantage of source routing. [6 points]

   *Advantage:* facilitates lookahead routing to eliminate RC state at each hop

   *Disadvantage:* cannot adapt to congestion

---

b) Briefly state an advantage and a disadvantage of the flattened butterfly topology as compared to a mesh with the same number of nodes. [6 points]

   *Advantage:* Fewer hops between nodes

   *Disadvantage:* requires higher radix routers
For parts (c) through (f), consider a 16-node 2D torus interconnect. Suppose there are 64-bit wide links in each direction between adjacent routers, as well as 64-bit incoming and outgoing links to each CPU. The interconnect is clocked at 1 GHz, and the link latency is a single cycle. All network messages include a 64-bit header and 64 bytes of message payload (e.g., a 64-byte cache block). The routers use the conventional 5-stage architecture (BW, RC, VA, SA, ST) discussed in class. The network uses credit-based flow control. Assume that credit updates are generated in the same cycle as switch traversal and take a cycle to process when received.

c) How many flits are there in each network message? [3 points]

9 - one header and 8 payload.

d) How many buffers are required for each virtual channel to avoid stalls due to a lack of flow control credits for flits that encounter no contention? Explain your reasoning. [5 points]

The credit count is decremented at the SA pipeline stage. A flit "holds" a credit through ST, LT, BW, RC, VA, SA, ST/CT, LT, CU, before it can be recycled by another flit. Hence, the required number of buffers is 9.

e) Suppose we wish to use a deterministic minimal routing algorithm on this topology. Briefly describe a strategy to make this routing algorithm deadlock free. [4 points]

Use a turn-based routing algorithm, such as west-first routing.
Consider a uniform random traffic pattern, that is, one where every node transmits a packet to every other node (including itself) with equal probability. What is the average number of hops a message must traverse? [4 points]

<table>
<thead>
<tr>
<th>Hops</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1/16</td>
</tr>
<tr>
<td>1</td>
<td>4/16</td>
</tr>
<tr>
<td>2</td>
<td>6/16</td>
</tr>
<tr>
<td>3</td>
<td>4/16</td>
</tr>
<tr>
<td>4</td>
<td>1/16</td>
</tr>
<tr>
<td>Avg</td>
<td>2 hops</td>
</tr>
</tbody>
</table>

Describe a scenario where non-minimal routing may be advantageous over minimal routing. [4 points]

Adaptive non-minimal routing may be able to route around local contention that a minimal route must cross.
3) GPUs [12 points]

a) Briefly describe how the strategy for dealing with long-latency memory accesses differs in a GPU from an out-of-order CPU. [6 points]

OoO cores hide long-latency memory accesses through out-of-order execution and memory level parallelism, by issuing multiple instructions and accesses from a single thread while waiting for a long latency miss. GPUs instead switch execution to a different warp while memory accesses from the first warp are outstanding.

b) Why is memory management currently more difficult in applications that use a GPU to accelerate computational kernels? [6 points]

The programmer must explicitly copy data structures to/from GPU memory and arrange them across the multiple memory spaces of the GPU.
4) Memory Consistency [35 points]

a) List two optimizations (hardware or software) enabled by the DRF0 programming model that are disallowed under (non-speculative implementations of) SC. [4 points]

- coalescing post-retirement store buffer
- any compiler optimization that reorders stores across loads, e.g., loop-invariant code motion

b) Briefly define store atomicity. [3 points]

- there is a total order of all stores performed in a system and all processors observe stores in this total order.

c) Propose a mechanism to achieve store atomicity in a processor with an update-based coherence protocol and an unordered point-to-point interconnect (i.e., one in which messages may be arbitrarily reordered). [4 points]

Achieving store atomicity in an update-based coherence protocol is challenging. A correct mechanism must assure that the new values of all stores becomes visible to all processors at the same moment with respect to other loads and stores. One way to accomplish this is using a two-phase update. The writer sends an update message to all readers, who record the new value, but may not yet supply it to any load; loads to the address stall. The readers all acknowledge the update. Once the last acknowledgement is received, the writer commits the update via a second message.
c) Assume the following program segments are executed on three processors of a multiprocessor machine. Initially before execution, all variables are equal to 0.
(Thanks to Prof. Narayanasamy for this question). [4 points]

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=1</td>
<td>u = A</td>
<td>v = B</td>
</tr>
<tr>
<td>B = 1</td>
<td></td>
<td>w = A</td>
</tr>
</tbody>
</table>

i) What are the possible final states for u, v and w under a sequentially consistent model?

- u = 0, v = 0, w = 0
- u = 0, v = 0, w = 1
- u = 0, v = 1, w = 0
- u = 0, v = 1, w = 1
- u = 1, v = 0, w = 0
- u = 1, v = 0, w = 1
- u = 1, v = 1, w = 1

ii) What are the final states that are NOT possible under SC?

- u = 1, v = 1, w = 0
d) Conventional high-performance out-of-order processors, such as the MIPS R10K, allow load instructions in the re-order buffer to execute speculatively out of order with respect to one another and with respect to store instructions. (i) How do out-of-order processors like the R10K ensure sequentially consistent execution? (ii) What hardware structures are required? (iii) Briefly describe a scenario where a misspeculation occurs. (Be sure your answer addresses all three parts (i)-(iii)). [10 points]

The R10K allows load operations to speculate past other loads and past incomplete stores. All load instruction addresses are tracked in a load queue, and incoming invalidation requests snoop this load queue. If a snoop matches a completed load, then a memory order violation occurred, and the offending load is rolled-back and re-executed using the ROB’s exception recovery mechanism. An example misspeculation scenario: Store to A is incomplete and blocked at the head of the ROB. A subsequent load to B completes via an L1 cache miss. Then, a store to B by another node generates an invalidation. Upon snooping the load queue, the invalidation finds an match, and the load to B (and all subsequent instructions) must be squashed and re-executed.
e) Rather than using complex hardware to detect memory ordering misspeculations, one simple way to enforce sequential consistency with high performance is to execute each load instruction twice: first, execute loads out-of-order as their addresses are calculated (as in your answer to part (c)), and then a second time using the naive method (as in your answer to part (a)). If both load operations return the same value, then sequential consistency has been maintained (i.e., the early out-of-order load did not race with a conflicting store). This approach is called "value-based memory ordering."

However, the downside of value-based memory ordering is that, because all loads execute twice, the L1 cache access bandwidth is doubled. Fortunately, several filtering mechanisms can identify loads that do not need to be re-executed, because they could not possibly have caused a memory ordering violation; these filtered loads are executed only once.

**Propose a design for such a filtering method.** In your answer, be sure to include: [10 points]

- A description of which loads your filtering mechanism will not re-execute
- An explanation of why the filtered loads do not need to be re-executed (i.e., how do you know they couldn't have exposed a memory ordering violation.)
- An argument that your mechanisms filters out a substantial number of load re-executions (you don't need to filter a majority, but you should argue you are capturing some kind of common case).
- A diagram and description of any new hardware structures you add and/or description of new software or compiler support you require