CS/ECE 757: ADVANCED COMPUTER ARCHITECTURE II
COMPUTER SCIENCES DEPARTMENT
UNIVERSITY OF WISCONSIN—MADISON

Prof. Mark D. Hill

Midterm Examination I
In-Class
Wednesday, February 27, 2019
Weight: 25%

1:15 minutes.

CLOSED BOOK, etc., but one cheat sheet allowed (two-sided 8.5x11 page).

The exam is *two-sided* and has **EIGHT** pages, including two blank pages at the end.

Plan your time carefully, since some problems make take longer than others.

NAME: _____ KEY _____

ID# _____

Problem Number	Maximum Points	Actual Points
1	12	
2	12	
3	12	
4	12	
5	12	
Total	60	

Problem 1: Performance, etc. (12 points)

(a) Consider the statement, "Local operations on thread 1 of a message-passing program are not ordered with respect to local operations on thread 2 of the same program." Is this statement true or false? Why?

Strictly speaking, this statement is false as operations thread 1 from before a message send to thread 2 are order before thread's 2 operations after its receive of the message. I gave full credit even with you said "true" and then talked about how thread-local operation were unordered except due to messages.

(b) Consider multiplying an n-by-n matrix A by an n-element vector x to get a n-element vector y: y = Ax. How long will take to execute "y = Ax" on the PRAM model. Why?

Multiplying an n-by-n matrix A by an n-element vector x must do n inner products, one for each output element of vector y. These can be done in parallel. Moreover, each inner product can be done with n parallel multiplies followed by a log(n) tree summing the products. Final answer: O(log(n)) time. I gave good credit for O(n) if you thought the inner product takes O(n).

(c) Let time(X,N) be the time to execute a program with problem size X on N processors. Conventionally the speedup on this program with problem size X would be time(X,1)/ time(X,N). When is this ***not*** the best definition of speedup? Why? What might you do instead?

The above strong scaling metric is not appropriate when users of a larger machine will use it the do a larger problem on about the same amount of time as they formerly used a smaller machine to solve a smaller problem. This is called "weak scaling."

For some problem size X', let time(X,1)/ time(X',N) = 1. Here the speedup is the additional work possible on the larger machine: work(X',N)/ work(X,1)

Problem 2: Synchronization (12 points)

(a) Provide pseudo-code for a simple implementation of `Lock(L)` and `Unlock(L)` using regular instructions and the compare-and-swap or `CAS()` hardware primitive. Recall from Scott13:

```
boolean CAS(word *a, word old, word new):
    atomic { t := (*a = old); if (t) *a := new; return t }
```

```
/* Let 0 be unlocked and 1 locked */
Lock (L) {
    while (!CAS(L, 0, 1));    /* continue spinning if CAS fails to due to L not being 0 */
}

Unlock(L) {
    L = 0;    /* No CAS needed, as only one lock holder */
}
```

(b) What is deadlock? What are three ways it can be avoided?

Deadlock occurs when two or more threads cannot make for progress, because each is waiting for lock (or other resource) held by another thread with a cyclic dependence.

Deadlock can be avoiding using at least one of:

- Order locks in a partial order to be followed when multiple locks are acquired,
- Have only one lock, or
- Be willing to give up all locks held (pre-emptible) and try again.

(c) Scott13 Chapter 4 gives many implementations of a spin lock. Which one(s) would you choose if you wanted a lock to be acquired by threads in the same order that they requested it (also known as first-come-first-service (FCFS) or first-in-first-out (FIFO)? Why?

Ticket locks and linked-list locks like MCS and CLH maintain FIFO order.

I would use ticket lock to get FIFO service with low-contention and MCS or CLH for FIFO service with higher contention.

Problem 3: Snooping Coherence (12 points)

Regarding snooping coherence with some or all of MOESI states of *modified (M), owned (O), exclusive (E), shared (S),* and *invalid (I)*.

(a) Why is a coherence protocol supporting only M and I usually a bad idea?

Most multithreaded programs share substantial **read-only** data and instructions that are poorly served by an M-I protocol that allows value data for each block to reside in at most one processor's cache.

In contrast, adding the shared (S) state allows multiple caches to hold read-only data, greatly reducing communication.

(b) What is the purpose of the O state? Give an example of a block 100 at processor P1 transitioning ***into*** the O state.

O has two potential advantages:

1. A cache in state in O can respond to other GetS request, which is important if a cache is much faster than memory or the LLC.
2. A cache in state in M can respond to other GetS request and transition to O without updating memory/LLC, which is important if memory/LLC is slow or sending to two destinations is inefficient.

Example: P1 100 M sees P2 100 GetS and then sends block 100 to P2 resulting in P1 100 O & P2 100 S.

(c) Why might block 100 in the O state at processor P1 transition next to M? Alternatively, why from O to S? Alternatively, why from O to I?

O→M: store by P1

O→S: not common, but possible if P1 allowed to writeback dirty data and transition to S.

O→I: store and GetM by P2 or writeback by P1.

(d) With an aggressive bus implementation, a processor P1 with block 100 in state I can issue a GETM to seek to transition into state M. Why might it not transition ***immediately*** to state M? Might it never reach M? Why or why not?

With an aggressive bus, regarding block 100 P1 issues a GETM transitions from I to IM^{AD} to await its transaction on the bus to then goes to IM^{D} and then receive data to go to M in the common case.

However, if P1 is in IM^{D} and sees an other GetM it transitions to $IM^{D}I$ and never to M. When is gets its data is will perform its store and transition to I. Other races are also possible. See Table 7.20 on the Primer.

Problem 4: Miscellaneous (12 points)

(a) In Arch2030, Ceze et al. identify five future trends. Describe at least two of them and why they might be important. To quote the paper:

1. We now have a clear *specialization gap*—a gap between off-the-shelf hardware trends and application needs. Some applications, like virtual reality and autonomous systems, cannot be implemented without specialized hardware, yet hardware design remains expensive and difficult.

2. *Cloud computing,* now truly ubiquitous, provides a clear "innovation abstraction;" the Cloud creates economies of scale that make ingenious, cross-layer optimizations cost-effective, yet offers these innovations, often transparently, to even the smallest of new ventures and startups.

3. *Going vertical* with 3D integration, both with die stacking and monolithic fabrication, is enabling silicon substrates to grow vertically, significantly reducing latency, increasing bandwidth, and delivering efficiencies in energy consumption.

4. *Getting closer to physics:* device and circuit researchers are exploring the use of innovative materials that can provide more efficient switching, denser arrangements, or new computing models, e.g., mixed-signal, carbon nanotubes, quantum- mechanical phenomena, and biopolymers.

5. And finally, *machine learning has emerged as a key workload;* in many respects, machine learning techniques, such as deep learning, caught system designers "by surprise" as an enabler for diverse applications, such as user preference prediction, computer vision, or autonomous navigation.

(b) Esmaeilzadeh et al. 2013 discuss "dark silicon." What is it? Why does it arise? Might it get worse? Why or why not?

With the end of Dennard Scaling, one can double the number of transistors in fixed area but using all of them will increase chip power. While this is sometimes okay, repeated transistor doublings can't repeatedly increase chip power. Thus, either chips get smaller or do not use all transistors at once, which leaves "dark silicon. Combinations are possible.

More transistors means more dark silicon, making the problem worse.

(c). Consider a system that supports "Sequential Consistency for Data Race Free (SC for DRF)". Must the hardware support SC? Why or why not?

The purpose of SC for DRF is to allow the programmers to reason with SC for a restricted set of programs (those without data races) even when the hardware supports only weaker models than SC, such as TSO and XC. Thus, hardware for "SC for DRF" does not need to implement SC.

Problem 5: Consistency (12 points)

Consider the following example with no other code executing. Assume that memory variables `foo` and `bar` are initially 0 and `rij` means processor `Pi`'s register `j`. A – F are statement labels. Assume *write atomicity*, i.e., when a processor's write is seen by *any other* processor, it is seen by *all other* processors.

(a) What executions does sequential consistency (SC) permit? Specify an execution by its final registers values: `(r11, r12, r21, r22) = (?, ?, ?, ?).`

Processor P1	**Processor P2**
`A: foo = 3;`	`D: r21 = bar;`
`B: r11 = foo;`	`E: bar = 5;`
`C: r12 = bar;`	`F: r22 = foo;`

Instruction B's r11 always gets 3 without a fence between A and B, regardless of whether bypassing is needed. Instruction D's r21 always gets 0 without a fence between D and E as it is before the only write to bar (E) in program order. Thus, (r11, r12, r21, r22) = (3, ?, 0, ?). There are three SC results:

- If A & C execute before E & F, we get (3, 0, 0, 3). /* old, new */
- If E & F execute before A & C, we get (3, 5, 0, 0). /* new, old */
- If A & E execute before C & F, we get (3, 5, 0, 3). /* new, new */

SC precludes (3, 0, 0, 0). /* old, old */ as program order has A→C and E→F and this outcome requires memory order to have C→E and F→A, resulting in a cycle: A→C→E→F→A.

(b) *Insert the necessary FENCEs to make a TSO implementation allow only SC executions. Why?*

Processor P1	**Processor P2**
`A: foo = 3;`	`D: r21 = bar;`
`B: r11 = foo;`	`E: bar = 5;`
`C: r12 = bar;`	`F: r22 = foo;`

Need one fence between instruction and A and C (either before or after B) to ensure A's write enters memory order after C's read. Otherwise TSO write buffer could delay A write past C's read. Need a second fence between E and F for the same reason.

(c). Insert the necessary FENCEs to make a relaxed XC implementation allow only SC executions. Why?

Processor P1	**Processor P2**
`A: foo = 3;`	`D: r21 = bar;`
`B: r11 = foo;`	`E: bar = 5;`
`C: r12 = bar;`	`F: r22 = foo;`

Need the same two fences as TSO, as only Store→Load ordering is in play and not L→L, L→S, or S→S.

Scratch Sheet 1 of 2 (in case you need additional space for some of your answers)

Scratch Sheet 2 of 2 (in case you need additional space for some of your answer