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 24, 2016
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: Message Passing (12 points)

(a) Message passing libraries, such as MPI, allow parallel programs to be written for parallel execution even on clusters. For best performance, programmers are often advised to avoid small messages. Why?

Message libraries, OSes, and networks have many overheads that are fixed per message, e.g., making a library call, system call, and network interface send. Sending fewer, larger messages incurs the overheads less often, effectively amortizing them better.

(b) What other techniques would you recommend for better message-passing performance? Why?

1. **Partition work to reduce the communication needed.**
2. **Replicate rarely updated data to reduce communication needed**
3. **Tolerate message latency by moving sends earlier and receives later.**
4. **Consider using asynchronous sends (see below).**

(c) Message passing libraries, such as MPI, often include alternative SEND routines. MPI_SEND, for example, always sends the value of its buffer argument at the time is was invoked even if the sending thread subsequently writes the buffer. On the other hand, MPI_ISEND permits the sending thread's subsequent buffer modifications to affect the contents to be sent. What are the advantages and disadvantages of these two types of SEND?

MPI_SEND cannot return until data is copied out of a user buffer. This can hurt performance by delaying the sending thread and/or adding an extra copy to the message send. This send is easier to use because, when it returns, the programmer can consider the data sent and immediately re-use the message buffer.

MPI_ISEND returns without copying data, potentially offering higher performance. Programmers must either not re-use the buffer for a while (e.g., via double buffer) or have to reason about message sends whose content may non-deterministically change.

Problem 2: Synchronization (12 points)

(a) Provide pseudo-code for a `test-and-test-and-set()` implementation of `Lock(L)` and `Unlock(L)` assuming only a `test-and-set()` hardware primitive. Assume that `test-and-set(L)` sets L to 1 and returns the previous value of L.

```
Lock(L) {
   while(test-and-set(L)==1) {
         while (L==1)} {} //spin
    }
}

Unlock(L) {
     L = 0
}
```

(b) What is a sense-reversed `Barrier(B)`? Why might it be better than a barrier implementation that does not use sense reversing?

When a barrier is reached by all necessary threads, a FLAG is often set to 1 to signal to all threads that they may proceed. If this simple barrier is to be reused again in a program, the FLAG must first be reset to 0. This requires some coordination so that the reset happens after all threads have observed the FLAG to be 1.

A sense-reversing barrier avoids this coordination have the first barrier completion set the FLAG to one, the second completion reset FLAG to 0, and alternating thereafter.

(c) Discuss at least one good option for implementing a barrier on a very large cache-coherence shared-memory machine.

On a very large shared memory machine, an implementation of a barrier with a single COUNT of the number of threads that have arrived can be a sequential bottleneck, as each thread has to update COUNT sequentially, often after obtaining a lock.

An option for very large shared memory machine is a tournament barrier that uses a tree of "sub-barriers". Each processors belongs to a group that accesses a leaf node of the tree. When the last of a group reaches the leaf, it proceeds up the tree and repeats. When all representatives have reached the tree root, all processors can be informed that the barrier is reached. For example, a two-level tree with fan-in 32 allows 1024 (32*32) processors to reach a barrier with at most 32processors trying to access a COUNT field at the same time.

Answers for other scalable barriers are also accepted if explained.

Problem 3: Consistency (12 points)

Consider the following example. Assume memory variables `data1`, `data2`, `flag1`, and `flag2` are initially `0` and `rij` means processor `Pi`'s register `j`. Assume also that all implementations ensure *write atomicity*, i.e., when a processor's write is seen by *any other* processor, it is seen by *all other* processors.

(a) Will this example behave the same with *total store order (TSO)* (or x86) as with sequential consistency? Why or why not? If not, what exactly should a programmer add to ensure a sequentially consistent execution even on TSO hardware. Why?

Processor P1	**Processor P2**	**Processor P3**
`data1 = 3;`	`L2: r21 = flag1;`	`L3: r31 = flag2;`
`data2 = 4;`	`if (r21==0)goto L2;`	`if (r31==0)goto L3;`
`flag1 = 1;`	`flag2 = 2;`	`r32 = data1;`
		`r33 = data2;`

Yes, this example will behave the same with TSO as with SC. TSO only relaxes the order of Stores → Loads. This example never depends on Stores → Loads so TSO behaves identically with SC. (P1 depends on Store → Store ordering, P2 depends on Load → Store ordering, and P3 depends on Load → Load ordering.)

(b) Will this example (same as above) behave the same with *example relaxed consistency* (XC) (or similar model) as with sequential consistency? Why or why not? If not, what are the minimum changes a programmer should make to ensure a sequentially consistent execution even on weaker hardware? Why?

Processor P1	**Processor P2**	**Processor P3**
`data1 = 3;`	`L2: r21 = flag1;`	`L3: r31 = flag2;`
`data2 = 4;`	`if (r21==0)goto L2;`	`if (r31==0)goto L3;`
`flag1 = 1;`	`flag2 = 2;`	`r32 = data1;`
		`r33 = data2;`

No, this example will not behave the same under XC and SC, because XC does not enforce any of the four orders (L→L, L→S, S→L, and S→S) while that example depends on some of these. A programmer could make programs behave like SC by inserting a FENCE before:

- **P1's `flag1 = 1` (ensures data stored before flag1 stored),**
- **P2's `flag2 = 2` (ensures final flag1 load before flag2 store), and**
- **P3's `r32 = data1` (ensures final flag2 load before data loads).**

Problem 4: Snooping Coherence (12 points)

Consider implementing snooping coherence in a system with a single-level of write-back caches. This problem asks you to rank the MOESI states (*modified, owned, exclusive, shared,* and *invalid*) in order of their importance and *to justify your answer*. Assume that your protocol must have the invalid state (I).

(a) If you could only implement two states, which state would you choose to implement along with I. Why?

M (modified): required for correct operation, allows the processor to have exclusive copy of the cache block with both read and write permissions.

(b) If you could only implement three states, which state would you add to your answer to (a)? Why?

S (shared): allows to optimize the common case in which several processors cache the block with read-only permissions; in that case the cache block can be replicated across several caches without compromising coherence; reduces the number of invalidates compared to MI and saves bandwidth.

(c) If you could implement only four states which state would you add to your answer to (b)? Why?

E (exclusive): allows to optimize the common case in which a private cache block is first read and then written to; with E state, the processor who has the block in E state can transition to M state on a write without informing the other caches, therefore reduces bus traffic. E is chosen before O because most blocks are never shared and always private, and the scenario in which a private block is first read and then written is very common.

Okay to add O before E if justified.

(d) What factors would affect whether you would want to add the fifth MOESI state to your answer to (c)?

O (owned): Factors: (1) latency of cache-to-cache transfers relative to cache-to-memory transfers, (2) reduction of pressure on memory bandwidth due to reduce the number of write-backs, (3) power/energy saved on reduced memory accesses, (4) added complexity of O state, (5) application-dependent memory access patterns.

Problem 5: Miscellaneous (12 points)

(a) Zhang et al. [MICRO 2015] present COUP with a new *update* coherence state (U)? What is the goal of the U state? What operations can a core perform on a block B in state U? What happens when a core block B in state U seeks to read a word in block B?

COUP's goal is to efficiently support concurrent, "reduction" operations from multiple cores without ping-ponging an M block among the cores, provided no core read or writes the value being reduced.

COUP supports reductions with communicative, associative operations, such as add, multiply, maximum, minimum, AND, OR and XOR).

On a read, COUP gathers values from each U copy and merges them together with the value of memory (really LLC) using the appropriate reduction operations (e.g., at LLC).

(b) What is the data parallel programming model advocated by Hillis and Steele [CACM 1986]? How does it differ from the PRAM model?

Data parallel allows sequential reasoning about operations on data aggregates, which is simple to reason about than multiple threads using shared memory or message passing.

Data parallel allows sequential reasoning of operations on data aggregate in shared memory, as if there was one PC (SI) and lots of data (MD) and thus a programming model for SIMD.

PRAM is a theoretical MIMD shared-memory model where an infinite number of processors operate in lock-step, reducing/eliminated the need for synchronization and not charging for communication. If all PRAM processors do the same sequence of instruction (SI), it can operate similar to the data parallel model, but PRAM is more general, as processors can do any instructions (MI).

In summary, they both operate in lock-step, accesses shared memory, but the data parallel model follow SIMD while PRAM allows MIMD operations as well.

(c) In Hydra's thread-level speculation (Hammond et al.), say a write X=20 in iteration 2 of a loop is issued before a write X=10 in iteration 1. What is the final value in memory? How it this ensured?

The final version of memory should be X=20, because the second iteration logically occurs after the first. Hydra ensures by having iterations put writes in a writebuffer, which commits writes in iteration order, even if X-20 arrived first.

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 answers)