CS 433 Final Exam – Dec 14, 2020, **7pm to 10pm CST**

Professor Sarita Adve

Name	
NetID	
Grad/Undergrad	

Instructions

1. Connect to the exam zoom room before you download the exam from compass. Set your camera as instructed - with your hands, face, and the screen of the device you use to write your solutions visible on zoom - from before you download the exam to after you submit it on compass. If you repeatedly violate these instructions for the zoom setup, we will be forced to nullify your exam.
2. You are allowed two sheets of scratch paper of 8.5"x11" (or less) each. They should initially be blank on both sides and be visible on zoom. No books, notes, or any other typed or written materials or calculators or other electronic materials are allowed.
3. Download the exam from compass 2g and save a copy on your device. It is acceptable to convert the word file into a google doc. Type your answers on the saved file/google doc (start by typing your name, netid, and grad/undergrad status above). Try to keep your answer within the space provided if possible. After you finish editing the document with your solutions, please save it, convert it to pdf, and upload the pdf back to Compass.
4. *In all cases, show your work. No credit will be given if there is no indication of how the answer was derived. Partial credit will be given where possible even if your final solution is incorrect, provided you show the intermediate steps in reaching the final solution.*
5. If you believe a problem is incorrectly or incompletely specified, make a reasonable assumption and solve the problem. The assumption should not result in a trivial solution. In all cases, clearly state any assumptions that you make in your answers.
6. If you need to ask a question, please use the "raise hand" feature on zoom or send a private message on zoom chat to Antonio and we will take you to a breakout room. Note, however, that our typical response will be #5 above.
7. This exam has ***6 problems*** and **15 *pages*** (including this one). **Only graduate students should solve problem 6. All students should solve all the remaining problems.** Please budget your time appropriately. Good luck!

Problem	Maximum Points	Points Received
1	15	
2	13	
3	14	
4	6	
5	2	
6	6 **(Graduate students only)**	
Total	50 for undergraduates 56 for graduates	

Problem 1 [15 points]

Consider the following code:

```
int i, int j, double A[256][256]

for (i = 0; i < 64; i++)

      for (j = 0; j < 64; j++)

            x += A[2*i][2*j];
```

Assume the following:

- Array A contains double words.
- Only accesses to array locations generate loads to the data cache. The rest of the variables are allocated in registers.
- The array A is stored in row major form.
- The program is running on a machine with an L1 data cache that has 32 words per cache line.
- Memory is word addressable.
- Assume the array A starts at a cache line boundary.
- Assume an infinite data cache that is initially empty.

Part A [2 points]

How many L1 data cache misses occur for the above code? You must explain how you derived this number to receive any credit.

Solution:

Each cache line holds 16 elements of the matrix. Each inner loop invocation therefore results in 8 cache misses (since it must bring in 128 contiguous elements). There are a total of 64 invocations of the inner loop. Therefore there are a **total of 512 L1 misses**.

Grading: 1 point for explaining that there are 8 cache misses per inner loop invocation. 1 point for arriving at the correct final answer. No points for an answer without an explanation.

Part B [2 points]

Suppose our machine has a data prefetch instruction with the format prefetch(array[index1][index2]). This prefetches the entire cache line containing the word array[index1][index2] into the data cache. Assume it takes one cycle for the processor to execute this instruction and send it to the data cache. The processor can then execute subsequent instructions.

Consider inserting prefetch instructions for the inner loop of the above code (i.e., ignore the outer loop for this part). Explain why we may need to unroll the inner loop to insert prefetches. What is the minimum number of times you would need to unroll the loop for this purpose?

Solution:

If we attempt to insert prefetch instructions without unrolling the inner loop, we would be prefetching superfluously since each prefetch brings in an entire cache line. To take advantage of prefetching, we would like every iteration of the loop to access as much data as possible in the prefetched cache line. We are told that each cache line contains 32 words (i.e., 16 double words). However, the program uses a strided access pattern where only the even array entries are used, which translates to only half of a cache line being used. Thus, we should **unroll the loop 16/2 = 8 times**.

Grading: 1 point for the correct answer and 1 point for explanation. No points for an answer without an explanation.

Part C [4 points]

Unroll the inner loop for the number of times identified in the previous part and insert the minimum number of software prefetches to minimize execution time. The technique to insert prefetches is analogous to software pipelining. You do not need to worry about startup and cleanup code (epilog and prolog) and do not introduce any new loops. Assume that two iterations of your unrolled inner loop are sufficient to hide the full latency of an L1 cache miss. Again, for this part, you may ignore the outer loop.

Note that if you obtained an incorrect answer for part B that results in a trivial solution to this part, you will not get any credit for this part. So be careful about part B. Again, the goal is to minimize the number of unnecessary prefetches (ignoring epilog/prolog issues and the outer loop).

Solution:

Using the answer from the previous part, the loop should be unrolled 8 times. Using a technique analogous to software pipelining, we should insert a prefetch instruction at the start of the loop body to prefetch the cache line being used two (new) iterations later. The reason for prefetching two iterations ahead is that we need two iterations to cover the memory latency. The correct solution is:

```
for (i = 0; i < 64; i++) {
        for (j = 0; j < 64; j += 8) {
                prefetch(A[2*i][2*(j+16)]);
                x += A[2*i][2*j];
                x += A[2*i][2*(j+1)];
                x += A[2*i][2*(j+2)];
                x += A[2*i][2*(j+3)];
                x += A[2*i][2*(j+4)];
```

```
            x += A[2*i][2*(j+5)];
            x += A[2*i][2*(j+6)];
            x += A[2*i][2*(j+7)];
        }
}
```

Grading: 1 point for properly inserting the prefetch instruction, 2 points for calculating the correct array offset for the prefetch instruction, taking answer to part (B) into account, and 1 point for unrolling the loop properly, taking the answer to part (B) into account. If an incorrect answer to part B results in a trivial solution here (e.g., no unrolling), no points are given.

Part D [3 points]

Consider your solution to part (C). How many of the data cache misses from the original code now have their latency fully hidden? For this part, consider both the inner and outer loops. You must explain how you derived this number to receive any credit. If an incorrect solution to part C trivializes this problem or makes it too difficult, no credit will be given.

Solution:

The first two iterations of the new inner loop will still result in cache misses because that data is not being prefetched. There are a total of 64 invocations of the inner loop. Therefore **there are a total of 128 L1 misses using prefetches.** Since there were originally 512 L1 misses, **384 of these L1 misses have their latency fully hidden due to the prefetches**.

Grading: 3 points for the correct answer with a reasonable explanation. No points without an explanation.

Part E [4 points]

Consider the data cache misses for which the latencies are not fully hidden in part (C). Suggest one enhancement to your code in part (C) that can hide some of the exposed latency. For this part, consider both the inner and outer loops. You may answer with a clear description in words or with code that contains the enhancement accompanied with some explanatory text. Again, solutions to previous parts that trivialize this part will not get any credit.

Solution:

The following code eliminates all but the first two L1 misses:

```
for (i = 0; i < 64; i++) {
        prefetch(A[2*(i+1)][0]);
        prefetch(A[2*(i+1)][16]);
```

```
        for (j = 0; j < 64; j+=8) {
                prefetch(A[2*i][2*(j+16)]);
                x += A[2*i][2*j];
                x += A[2*i][2*(j+1)];
                x += A[2*i][2*(j+2)];
                x += A[2*i][2*(j+3)];
                x += A[2*i][2*(j+4)];
                x += A[2*i][2*(j+5)];
                x += A[2*i][2*(j+6)];
                x += A[2*i][2*(j+7)];
        }
}
```

Since the data cache is infinite, this simply prefetches the data needed by the first two iterations of the inner loop during the previous iteration of the outer loop. Therefore, the only two remaining L1 misses take place during the first iteration of the outer loop for the first two iterations of the inner loop.

Grading: 1 point per correct prefetch insertion (2 points total), 2 points for correct explanation. Alternate responses may also be acceptable or receive partial credit.

Problem 2 [13 points]

Consider a virtual memory system with the following parameters.

- 64-bit virtual addresses
- 48-bit physical addresses
- 4KB pages
- 16B cache blocks
- Byte-addressing

Furthermore, main memory is interleaved on a word (32-bit) basis with four banks and a new bank access can be started every cycle. It takes 10 processor clock cycles to send an address from the processor to main memory; 50 cycles for memory to access a word; and 20 cycles to send one word of data back from memory to the processor. The memory bus width is 1 word.

Part A [3 points]
How many bits are needed for the page offset, virtual page number, and physical page number?

Solution:
2^12 byte page size → 12 bits for the page offset.
64 – 12 = 52 bits for virtual page number, 48 – 12 = 36 bits for physical page number.
Grading: 1 point for each set of bits.

Part B [4 points]
What is the minimum size of a page table entry? Ignore bits to guide the replacement policy, but ensure you account for all other necessary bits. Assume a valid page can have every combination of read, write, and execute permissions encoded in the protection bits. Any extraneous bits mentioned will fetch negative points.

Solution:
The PTE needs to store 36 bits of physical address. There are also 3 permission (protection) bits, 1 valid bit, and 1 dirty bit for a minimum of 41 bits.
Grading: 1 point each for the correct number of physical address, permission, valid, and dirty bits. -1 for any extraneous bits. 4 points maximum. 0 points minimum.

Part C [2 points]
What is the minimum size of an entry in a fully-associative TLB? Again, ignore bits for the replacement policy. Any extraneous bits mentioned will fetch negative points.

Solution:
In a fully associative TLB, every entry needs the entire virtual page number as a tag, in addition to the information the PTE required. The size of a TLB entry therefore is 52 + 41 = 93 bits.
Grading: 1 point for the correct size of the tag and 1 point for the correct size of the "data" bits. -1 point for extraneous bits. Maximum 2 points. Minimum points is 0.

Part D [2 points]
If the processor did not have a TLB, how long would each address translation take? Assume that the complete page table resides in main memory and there are no page faults.

Solution:
A PTE needs to be fetched from the page table (which resides in main memory) to perform address translation. Each PTE is 41 bits, therefore two bus transfers are required to send the data back.
The time required would be 10 cycles to send the address + 50 cycles to access the word + 2*20 cycles to send the data back = 100 cycles.
Grading: 1 point for realizing that two bus transfers would be required to send a PTE from memory to the processor. 1 point for stating the correct memory access time formula.

Part E [2 points]
Now assume that the processor has a fully-associative TLB with a hit rate of 97% and a hit time of 1 cycle. How long do address translations take now, on average? What is the speedup over the system without a TLB? Again, assume that all pages reside in memory and there are no page faults.

Solution:
TLB access time = hit time + miss rate x miss penalty = 1 + 0.03*100 = 4 cycles.
Speedup = 100/4 = 25x.
Grading: 1 point for writing the correct formula for TLB access time/address translation time. 1 point for speedup.

Problem 3 [14 points]

This question concerns a snooping *update* (as opposed to invalidate) cache coherence protocol. Consider a system where the processors are connected by a bus, the caches are write-back and write-allocate and cache coherence is maintained through a snooping update protocol. In a snooping update protocol, when a cache modifies its data, it broadcasts the updated data on a bus using a *bus update* transaction, if necessary. Memory and all caches that have a copy of that data then update their own copies. This is in contrast to the invalidation protocol discussed in class where a cache invalidates its copy in response to another processor's write request to a block.

Our update protocol has three states – CE, CS and DE:

- CE (Clean Exclusive): The block is present *only in this cache (exclusively)* and memory also has the same (clean) copy.
- CS (Clean Shared): The block is present in *several caches (shared)* and memory and all those caches have the same (clean) copy.
- DE (Dirty Exclusive): The block is present *only in this cache (exclusively)* and the data in the cache is updated or dirty (i.e., a more recent version than the copy in memory).

All caches are write-allocate. A write-back policy is used if the line is in DE or CE state. For a line in CS state or a line not present in the cache, a write-through policy is used. The bus has a special line called Shared Line (SL) whose state is usually 0. *When cache i performs a bus transaction for a specific cache line*, all the caches that have the same line pull up the Shared Line (SL) to 1. If no other cache has the line, the Shared Line (SL) remains at 0. When cache i performs a bus transaction, it uses the state of the Shared Line (SL) to determine whether to change to an exclusive state or the shared state.

Assume that if a request is made to a block for which memory has a clean copy, memory will service that request. If the memory does not have a clean copy, the cache with the updated block will service the request and memory will also get updated.

For the question below, consider the following bus transactions:

- BR: Bus Read – Request to get the cache line (on a cache miss).
- BU: Bus update – Request to update copies of the cache line in memory and other caches with the new value of a word in the block.
- BRU: Bus read and update – A combination of BR and BU.

Note: you are not required to consider Bus Writeback, which may take place on a replacement.

Part A [8 points]

Fill out the following state transition table for processor *i* performing a memory instruction. Show the next state for a block in the cache of processor *i* and any bus transaction performed by processor *i*. Each entry should be filled out as:

Next State/Bus Transaction (e.g. *CS/BR*),

Where

Next State = CS, CE, DE or NIC (Not in Cache; i.e., a cache miss)
Bus Transaction = BR, BU, BRU, or NT (No transaction)

Note: If an entry is not possible (i.e., the system cannot be in such a state) write "Not Possible" in that entry.

	SL is 0 **if** proc *i* does a bus transaction		SL is 1 **if** proc *i* does a bus transaction	
Current State in processor i	**Read by proc *i***	**Write by proc *i***	**Read by proc *i***	**Write by proc *i***
CE	CE/NT	DE/NT	CE/NT	DE/NT
CS	CS/NT	CE/BU	CS/NT	CS/BU
DE	DE/NT	DE/NT	DE/NT	DE/NT
NIC	CE/BR	CE/BRU	CS/BR	CS/BRU

Grading: ¼ point for each Next State or Bus Transaction value that is correct. That is, each entry is worth ½ point – ¼ point for the Next State value and ¼ point for the Bus Transaction value.

Part B [6 points]
Fill out the following state transition table for the cache of processor *i*. Show the next state for a block in the cache of processor *i* and any action(s) taken by the cache when a bus transaction is initiated by another processor *j*. Each entry should be filled as:

Next State/Action (e.g. *CS/UPDL*)

Where

Next State = CS, CE, DE or NIC (Not in Cache; i.e., a cache miss)
Action = PULLSL1: Pull SL to 1
UPDL: Update block in cache *i* (i.e., one's own cache)
PROVL: Provide block in response to BR or BRU (main memory is also updated)
NA: No Action

Note: If an entry is not possible (i.e., the system cannot be in such a state) write "Not Possible" in that entry.

State in proc *i*	BR by proc *j*	BU by proc *j*	BRU by proc *j*
CE	CS/PULLSL1	Not possible	CS/PULLSL1, UPDL
CS	CS/PULLSL1	CS/UPDL, PULLSL1	CS/UPDL, PULLSL1
DE	CS/PROVL, PULLSL1	Not possible	CS/PROVL, UPDL, PULLSL1
NIC	NIC/NA	NIC/NA	NIC/NA

Grading: ¼ point for each correct Next State, 1/4 point for each correct component of each Action. Each "Not Possible" carries ½ point.

Problem 4 [6 points]

Systems with a relaxed consistency model usually offer a *memory fence* instruction to allow the programmer to enforce correct behavior. A full fence restricts reordering memory operations, requiring all previous (by program order) reads and writes to complete before subsequent (by program order) accesses begin.

Consider the code segments below. X, Y, Z, and Flag are shared variables. Reg1, reg2, reg3, and reg4 are local registers. All variables are initially 0.

Processor 0:	**Processor 1:**
X = 1	while (Flag == 0) {;}
Y = 1	reg3 = X
Flag = 1	reg4 = Y
while (Flag == 1) {;}	Z = 1
reg1 = Y	Flag = 0
reg2 = Z	

Part A [2 points]

Consider a machine with a very relaxed memory model, where any program ordered pair of accesses to different variables may be reordered (except as restricted by a fence instruction). Program order between accesses to the same variable, however, must be enforced.

What are the different combinations of values that the four reads on A, B, and C can return (i.e., write the possible combinations of final values of reg1, reg2, reg3, and reg4)?

Solution:

reg 1's final value is always 1 because Processor 0's write of Y must occur before its read of Y and there is no other write to Y. reg2, reg3, and reg4 could have either 0 or 1 as their final values since there is no ordering constraint between the writes of X, Y, and Z and the reads for those register values.

1 point for reg1 and 1 point for the rest.

Part B [2 points]

Now assume our machine obeys sequential consistency. What are the possible combinations of the final values for the four registers?

Solution:

(2 points) With sequential consistency, the only allowed result is: reg1 = reg2 = reg3 = reg4 = 1.

Part C [2 points]

Now reconsider the machine from Part A with the very relaxed memory model. Insert the minimal number of fence instructions in our code fragment to limit the results on our very relaxed machine to only those possible under Sequential Consistency. For convenience, the code fragment is repeated below.

Processor 0:	**Processor 1:**
X = 1	while (Flag == 0) {;}
Y = 1	reg3 = X
Flag = 1	reg4 = Y
while (Flag == 1) {;}	Z = 1
reg1 = Y	Flag = 0
reg2 = Z	

Solution:

To achieve this result on our relaxed machine, we need a fence before each write to flag and after each read of flag (these are the synchronizing or racing instructions).

1 point if the above fence instructions are inserted. 2 points if no other fence instructions are inserted.

Problem 5 [2 points]

This question concerns the mini-project presentations in class. You should answer **only 2 out of the 5** questions below (if you answer more than two, we will pick the first two). To answer a question, highlight in yellow the most appropriate choices. Some questions have multiple correct choices. Points will be given for a question only if all appropriate choices for that question are highlighted and no incorrect choice is highlighted.

Part A [1 point]

Which **one** of the following processors uses DynamIQ cluster technology **and** supports big and little cores in a big.LITTLE configuration?

a) Intel Skylake
b) ARM Cortex A78
c) NVIDIA Ampere GA102
d) AMD Zen

Solution: (b).

Part B [1 point]

Which **one** of the following processors uses a simple neural network (Perceptron) for branch prediction as reported in the class presentations?

a) Intel Skylake
b) ARM Cortex A78
c) NVIDIA Ampere GA102
d) AMD Zen

Solution: (d).

Part C [1 point]

Which of the following features are supported by the Intel Skylake processor?

a) Three levels of on-chip cache
b) Simultaneous Multithreading or Hyperthreading
c) Multiple cores
d) A picture by Rembrandt of the sky and a lake in his hometown on the processor die.

Solution: (a), (b), (c).

Part D [1 point]

Which of the following applications is the ARM Neoverse N1 CPU especially suited for?

a) Mobile (smartphones/tablets)
b) Cloud and infrastructure
c) Ultra-high reliability in extreme environments, especially space vehicles
d) Gaming

Solution: (b).

Part E [1 point]

Which **one** of the following processors supports CUDA cores, tensor cores, and ray tracing (RT) cores?

a) Intel Skylake
b) ARM Cortex A78
c) NVIDIA Ampere GA102
d) AMD Zen

Solution: (c).

ONLY GRADUATE STUDENTS SHOULD SOLVE THIS PROBLEM

Problem 6 [6 points]

A ticket lock is yet another scheme for multiprocessor synchronization. The ticket lock is modeled after the way customers are often served in a deli, where the customers all take a uniquely numbered ticket as they arrive, and there is a display showing the "now serving number." When your number appears, you are served.

In a multiprocessor system, a ticket lock is implemented with two counters - one for the "take a unique number" counter and one for the "now serving" counter. As each processor arrives at the lock point, it reads the first counter to get its unique number and increments the counter. It then waits (spins) until its number appears on the 2nd counter. It then proceeds into its critical section. At the end of the critical section, it increments the 2nd counter so another processor can proceed.

Assume the system implements a fetch&increment as discussed in class. Assume a sequentially consistent memory model.

Part A [1 point]

Would you use the fetch&increment instruction to implement the first counter, "take a unique number," or would you use ordinary loads/stores? Explain your answer.

Solution:

The fetch&increment instruction is best. This location must return unique values and is updated by every processor, so simply using one load and one store will not work.

Grading: No partial credit.

Part B [1 point]

Would you use the fetch&increment instruction to implement the second counter, the "now serving number," or would you use ordinary loads/stores? Explain your answer.

Solution:

This location is only written to by the processor (the lock holder) in the critical section. The value written is one plus the value the lock holder saw when it read the "take a number" counter. The lock holder can therefore use a simple store to increment the "now serving" counter.

Grading: No partial credit.

Part C [4 points]

For the system described, write assembly code to implement the ticket lock sequence of taking a unique number, waiting for your turn, and updating the counters as needed. Do not worry about the initial value of either counter or about overflow. Indicate where the critical section is executed with a comment line such as /*Critical Section*/. Use names such as "Count1" for the first counter and "Count2" for the 2nd counter, make sure it is obvious what you are referring to. For the fetch&increment instruction, use the format "FETCH&INC Rn, ADDR", where ADDR is the memory address that is fetched and incremented and Rn is the register that stores the result of the fetch&increment.

Solution:

```
        FETCH&INC R1, Count1      /* take your number and increment */
WAIT:   LD  R2, Count2            /* poll now serving counter */
        SUB R2, R2, R1            /* test for match, spin if not equal */
        BNEQZ R2, WAIT

        /* Critical Section */

        ADD R1, R1, #1            /* increment now serving count */
        ST R1, Count2             /* and update the memory location */
```

Grading:

1 point for proper atomic Fetch&Inc to get the ticket number – no points if not Fetch&Inc.

1 point for proper poll and spin on compare to second counter.

1 point for proper location of critical section.

1 point for proper update of Count2 – ½ point if answered B as yes and did a proper atomic update here (if implementation is consistent with answer of Part B).