CS 433 Final Exam: Dec 15, 2021

Professor Sarita Adve

Time: 3 Hours

Please print your Name and NetID and circle your course section below.

Name:	Solutions	
NetID:		
Section:	T3 (Undergraduate)	T4 (Graduate)

Instructions

1. No books, papers, notes, or any other typed or written materials are allowed. No calculators or other electronic materials are allowed.
2. Please do not turn in loose scrap paper. Limit your answers to the space provided if possible. If this is not possible, please write on the back of the same sheet. You may use the back of each sheet for scratch work.
3. *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 even if your final solution is incorrect, provided you show the intermediate steps in reaching the final solution.*
4. 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.
5. This exam has **7 problems** and **15 pages** (including this one). **All students** should solve **problems 1 through 6**. Only **graduate students should solve problem 7**. Please budget your time appropriately. Good luck!

Problem	**1**	**2**	**3**	**4**	**5**	**6**	**Graduate Students 7**	**Total**
Points	15	6	10	6	12	6	10	55 (undergrads) 65 (graduates)
Score								

Problem 1 [15 points]

Consider the following code:

```
double A[256][256];
double x = 0.0;

for (int i = 0; i < 64; i++) {
      for (int 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 order.
- 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[i][j])`. This prefetches the entire cache line containing the word `array[i][j]` 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.
1 point for explanation.
No points for an answer without an explanation.

Part C [4 points]

Unroll the inner loop the number of times identified in Part B 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 (i.e., 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 (int i = 0; i < 64; i++) {
      for (int j = 0; j < 64; j += 8) {
            prefetch(A[2*i][2*(j+16)]);
            x += A[2*i][2*(j+0)];
            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.
1 point for unrolling the loop properly.

Take into account the answer to Part B for array offset and loop unrolling.
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? Consider both the inner and outer loops for this part. 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 for an answer 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. Consider both the inner and outer loops for this part. 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 (int i = 0; i < 64; i++) {
      prefetch(A[2*(i+1)][0]);
      prefetch(A[2*(i+1)][16]);
      for (int j = 0; j < 64; j += 8) {
            prefetch(A[2*i][2*(j+16)]);
            x += A[2*i][2*(j+0)];
            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 for each correct prefetch instruction in the outer loop (2 total).
2 points for a correct explanation.

Alternate responses may also be acceptable or receive partial credit.

Problem 2 [6 points]

Consider a system with a 4 GHz processor and a 667 MHz memory bus. Consider page mode memory with a row buffer size of 16 KB. There is a cache with 64 byte blocks. Assume the cache is large enough that there are no capacity or conflict misses. Use the following simplified memory timing model:

- Assume the time taken by a read is the time to send the address (including the memory access time for that address) plus the time to transfer back the data.
- Every read transfers 64 bytes of data. This data transfer takes 4 bus cycles.
- Sending an address that hits in the row buffer (it is to the same row as the previous read) takes 9 bus cycles.
- Sending an address on another row takes 18 bus cycles.
- Only one memory request can occur at a time. The next request may begin on the bus cycle after data from the last read is returned.

The program begins with the cache empty and no row buffer open in memory. Consider the following code, with array A stored in row-major order. Assume that only accesses to array locations generate memory accesses and that all scalar variables are allocated in registers.

```
int A[1024*1024]; /* 4-byte integers, begins on a row boundary */
int max = INT_MIN;

for (int row = 0; row < 1024; row++)
     for (int col = 0; col < 1024; col++)
          if (max < A[row + col*1024])
               max = A[row + col*1024];
```

How many read requests for the array A will be sent to main memory? How many bus cycles will these requests take, if they are issued in order with no extra delay between requests? What is the average number of CPU cycles per memory request?

Solution:

Each read returns $64 \div 4 = 16$ elements, so there are $1024 \times 1024 \div 16 = 1024 \times 64 = 64\text{K}$ read requests for array A sent to memory.

The row buffer size is 16 KiB and each array entry is 4 bytes, so the row buffer holds 4096 array entries or 4 contiguous rows of array A. Since A is being accessed column by column, every fourth memory access will be a row buffer miss; put another way, every fourth row of a column will miss in the row buffer.

Each read takes 4 cycles to return data, and 9 cycles for the column address. Every row buffer miss must send a new row address as well. In total, the memory accesses take $(9 + 4) \times 64\text{K} + 9 \times (64\text{K} \div 4) = 976\text{K} = 999,424$ memory bus cycles.

There are 64K requests in total, so the average memory bus cycles per request is $976\text{K} \div 64\text{K} = 15.25$. Each bus cycle takes $1e9 \div 667e6 = 1.5\text{ns}$; since at 4 GHz each CPU cycle is $1e9 \div 4e9 = 0.25\text{ns}$, this is $1.5\text{ns} \div 0.25\text{ns} = 6$ CPU cycles. So the average number of CPU cycles per request is $15.25 \times 6 = 91$ cycles.

Grading:

1 point for the number of read requests.
2 points for correct formulation of bus cycles for row buffer hits.
2 points for correct formulation for row buffer misses.
1 point for converting to average CPU cycles per memory request

Problem 3 [10 points]

Consider a system with the following characteristics:

- 20-bit physical address
- 32-bit virtual address
- 4 KB page size
- Fully-associative data TLB with
    - 4 entries
    - LRU replacement policy
- 4-way set-associative 16 KB cache with
    - 16-byte block size
    - Physically-indexed
    - Physically-tagged
    - Write-allocate, write-back policy
- Byte-addressable memory

The following lists part of the page table entries corresponding to a few virtual addresses, using hexadecimal notation. Protection bits of 01 imply read-only access and 11 implies read/write access. Dirty bit of 0 implies the page is not dirty. Assume the valid bits of all the following entries are set to 1. For your reference, this table is repeated on the next page as well.

Virtual Page Number	Physical Page Number	Protection Bits	Dirty Bits
FFCAC	FF	01	0
FFCAB	CA	11	0
FFCAA	1D	11	0
FFCAE	BA	01	0
FFCAD	CB	11	0

The table on the next page lists a stream of data loads and stores to virtual addresses by the processor. All addresses are in hexadecimal. Complete the blank entries in the table corresponding to these loads and stores using the above information. For the data TLB hit, data cache hit, and protection violation columns, specify “yes” or “no.” Assume that initially the data TLB and data cache are both empty.

	Operation	Virtual Address	Physical Address	Part of Physical Address used to Index Cache	TLB Hit?	Cache Hit?	Protection Violation?
1	Load	FFCACABC	FFABC	AB	No	No	No
2	Store	FFCABABC	CAABC	AB	No	No	No
3	Load	FFCACFBB	FFFBB	FB	Yes	No	No
4	Load	FFCABABD	CAABD	AB	Yes	Yes	No
5	Store	FFCAAAAD	1DAAD	AA	No	No	No
6	Load	FFCADABA	CBABA	AB	No	No	No
7	Load	FFCAECBA	BACBA	CB	No	No	No
8	Store	FFCACFBF	FFFBF	FB	No	Yes	Yes

Grading:

¼ point for each correct table entry.

The following table is repeated from the previous page for your reference.

Virtual Page Number	Physical Page Number	Protection Bits	Dirty Bits
FFCAC	FF	01	0
FFCAB	CA	11	0
FFCAA	1D	11	0
FFCAE	BA	01	0
FFCAD	CB	11	0

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 can begin.

Consider the following code segments. X, Y, Z, and Flag are shared variables. r1, r2, r3, and r4 are local registers. All variables are initialized to 0.

Processor 0	**Processor 1**
X = 1	while (Flag == 0) {;}
Y = 1	r3 = X
Flag = 1	r4 = Y
while (Flag == 1) {;}	Z = 1
r1 = Y	Flag = 0
r2 = 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 X, Y, and Z can return (i.e., write the possible combinations of final values for r1, r2, r3, and r4)?

Solution:

The final value of r1 is always 1, because Processor 0 must write Y before it can read Y—and there is no other write to Y. r2, r3, and r4 could have either 0 or 1 as their final values, since there is no ordering constraint between the writes to X, Y, and Z and their subsequent reads.

Grading:

1 point for determining the correct value of r1.
1 point for determining the correct set of values for r2, r3, and r4.

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:

Under sequential consistency, the only valid result is r1 = r2 = r3 = 1.

Grading:

2 points for determining the correct values of r1, r2, r3, and r4.

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 the code segments to limit the results on our very relaxed machine to only those possible under Sequential Consistency. For convenience, the code segment is duplicated below.

Processor 0	**Processor 1**
X = 1	while (Flag == 0) {;}
	mfence()
Y = 1	r3 = X
mfence()	
Flag = 1	r4 = Y
while (Flag == 1) {;}	Z = 1
mfence()	mfence()
r1 = Y	Flag = 0
r2 = 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.

Grading:

1 point if the above four fence instructions are inserted.
1 point if no extraneous fence instructions are inserted.

Problem 5 [12 points]

This problem concerns MESIF, an invalidation-based snooping cache coherence protocol for bus-based shared-memory multiprocessors with a single level of cache per processor. The MESIF protocol has five states. A block can be in one of the following states in cache C:

- **M** *Modified*: The block is present in a single cache C. The block is dirty or modified; i.e., the data in the block reflects a more recent version than the copy in memory.
- **E** *Exclusive*: The block is present in a single cache C. The block is clean; i.e., memory has an up-to-date copy of the block.
- **S** *Shared*: The block is present in cache C and possibly present in other caches. The block is clean.
- **I** *Invalid*: The block is not valid in cache C. Space for the block may or may not be currently allocated in this cache.
- **F** *Forward*: The block is present in cache C and possibly present in other caches. The block is clean. Additionally, cache C must service the requests of other caches to this block; memory will not respond to requests for this block. Only one cache can have the block in Forward state.

If cache C has a block A in the Modified, Exclusive, or Forward state, then cache C responds to any requests for block A from other processors. If the request is a read, then the state of block A in cache C transitions to the Shared state, while the new copy of block A in the requester's cache assumes the Forward state. If cache C had block A in the Modified state, then memory updates its copy as well.

On replacement of a block in the Modified, Exclusive, or Forward state, memory resumes responsibility for servicing subsequent requests for that block.

For this problem, assume a system with three processors—P1, P2, and P3—each with a single level cache that stores only a single block and starts empty.

Complete the table on the next page for the MSI protocol studied in class and for the MESIF protocol described above. The first column gives memory accesses (Read or Write) initiated by P1, P2, or P3 to block A. For each protocol, you are to fill the entry for the states of block A in the caches of P1 and P2 after the access in the first column completes. You are also to fill the entry for the actions on the bus initiated by the processors or Memory (Mem) required to complete the access in column 1.

The possible states are: M, E, S, I, and F.

The possible bus actions are X / Y, where:

- X is the initiator of the action and can be P1, P2, P3, Mem, or None
- Y is the action initiated by X and can be: Get Block A, Send Block A, Invalidate, or None

For each entry, there can be multiple actions, possibly by multiple initiators.

Assume that the accesses occur in the order in the table and that no other memory accesses / traffic occur. The first row is filled out for you.

	MSI			MESIF		
Access on block A	**P1 State**	**P2 State**	**Bus Actions**	**P1 State**	**P2 State**	**Bus Actions**
P1 Read	S	I	P1 / Get Block A Mem / Send Block A	E	I	P1 / Get Block A Mem / Send Block A
P2 Read	S	S	P2 / Get Block A Mem / Send Block A	S	F	P2 / Get Block A P1 / Send Block A
P3 Read	S	S	P3 / Get Block A Mem / Send Block A	S	S	P3 / Get Block A P2 / Send Block A
P2 Write	I	M	P2 / Invalidate	I	M	P2 / Invalidate
P2 Write	I	M	None	I	M	None
P1 Write	M	I	P1 / Get Block A, Invalidate P2 / Send Block A	M	I	P1 / Get Block A, Invalidate P2 / Send Block A
P2 Read	S	S	P2 / Get Block A P1 / Send Block A	S	F	P2 / Get Block A P1 / Send Block A

Grading:

¼ point for each correct state entry.
½ point for each correct bus action entry, for all actions together.
We aim to not penalize cascading errors, unless the cascading errors fundamentally impact the intended learning from the problem.

Problem 6 [6 points]

This question concerns the mini-project presentations in class. Circle the most appropriate choices for each question below. Some questions have multiple correct choices. Points will be given for a question only if **all appropriate** choices for that question are circled and **no incorrect** choice is circled.

Part A [1 point]

Which of the following branch predictors are used in AMD Zen 3?

a) Simple neural network (Perceptron)
b) Static, always-taken
c) A combination of BERT and GPT transformer neural network
d) TAGE predictor

Part B [1 point]

Which of the following features are available in ARM Cortex A76?

a) Thousand way Simultaneous Multithreading (SMT) or Hyperthreading
b) Single issue, in-order, non-pipelined execution with no branch prediction
c) On-chip wireless/RF cache invalidation broadcasts
d) Support for heterogeneous processors via DynamIQ big.LITTLE

Part C [1 point]

What is true of the memory hierarchy in ARM Cortex X1?

a) Multiple error detection and correction modes for both cache and memory
b) Caches are built out of non-volatile memory technology
c) Main memory uses 6D stacked architecture
d) L1 cache miss latency is 1 cycle

Part D [1 point]

Which of the following features is unique to the Graphcore IPU?

a) All processing uses homomorphic encryption
b) Major focus on supporting a large amount of on-die SRAM memory
c) On-die SRAM is configured as local memory or scratchpads (vs. caches and shared memory)
d) Designed primarily for ML workloads

Part E [1 point]

Which of these is true about Intel Ice Lake?

a) Shadow Register Alias Table (RAT) helps track speculative state
b) Supports alternative RISC architecture
c) It works at 0 Kelvin, hence the name Ice Lake
d) It breaks instructions into micro ops which can be cached after decoding

Part F [1 point]

What is true about the Exynos M5?

a) It is designed by Samsung for Galaxy phones
b) It generates a new hash for each execution context to prevent microarchitectural state leakage (e.g., to prevent Spectre like security attacks)
c) It achieves low power by being a simple single-issue, in order processor
d) It's major competitor is NVIDIA's Ampere GPU

ONLY GRADUATE STUDENTS SHOULD SOLVE PROBLEM 7

Problem 7 [10 points]

A common primitive to implement atomicity in cache-coherent systems is the pair of instructions *Load-Linked* and *Store-Conditional* (usually abbreviated LL and SC, respectively). The Load- Linked instruction returns the value at the requested memory location, as usual. A subsequent Store-Conditional instruction will store a new value *only* if no updates by any other processor to that location can have occurred since the LL instruction. Assume that the SC instruction returns **1** if the store succeeded (i.e., there were no intervening writes to the memory location since LL), **0** if it failed.

Rather than actually guarantee atomicity, the LL-SC primitive instead informs the program when atomicity might have been violated. For the curious: LL/SC can be implemented (in the MIPS R4000 for example) by loading the line into cache normally, but also recording the address in a special register, and causing the store to fail if the line has been invalidated in the interim.

Use the following mnemonics for LL and SC:

`LL Rx, addr`	loads the value at address `addr` into register `Rx`
`SC Rx, addr`	stores the value in `Rx` at `addr` and then puts `1` in `Rx` for success and `0` for failure

Part A [5 points]

Implement the functionality of a compare-and-swap (CAS) primitive using the Load-Linked and Store-Conditional primitives, using any normal pseudo-assembly instructions you require.

Recall the semantics of CAS: CAS(`Addr`, `Rold`, `Rnew`) compares the values at memory location `Addr` to `Rold`. If they are equal, it exchanges the value of `Addr` with `Rnew`; i.e., `Addr` gets `Rnew` and `Rnew` gets the old value at `Addr`. CAS is performed as one atomic operation.

Solution:

```
CAS:  MOVE  R1,   Rnew
      LL    R2,   Addr
      BNE   R2,   Rold, DONE
      SC    R1,   Addr
      BEQZ  R1,   DONE
      MOVE  Rnew, R2
DONE:
```

One issue with this solution is that if there is an intervening write between `LL` and `SC` that does not change the value of `Addr`, then depending on the implementation, `SC` might still fail. If you considered this and wrote a solution where the branch after the `SC` loops back to `CAS` to retry after failure, you will also get full credit.

Grading:

1 point for correct use of load-linked.
1 point for correctly checking if the value is equal to `Rold` and branching.
1 point for correct use of store-conditional.
1 point for correctly checking whether LL/SC fails and branching.
1 point for returning the correct value in `Rnew`, depending on whether the CAS succeeds or fails.

Part B [5 points]

Implement the functionality of a Fetch-and-Add (FAA) primitive using the Load-Linked and Store-Conditional primitives, using any normal pseudo-assembly instructions you require.

Use the following semantics of FAA: FAA(`Addr`, `Rincr`) atomically increments the value at memory location `Addr` by an amount equal to `Rincr`. (FAA doesn't return until the memory location has been successfully updated.) It returns the *old* value of location `Addr` (not the newly updated value) in `Rincr`. FAA is performed as one atomic operation.

Solution:

```
FAA:  LL    R1,   Addr
      DADD  R2,   R1,   Rinc
      SC    R2,   Addr
      BEQZ  R2,   FAA
      MOVE  Rinc, R1
```

Grading:

1 point for correct use of load-linked.
1 point for the addition of the increment.
1 point for correct use of store-conditional.
1 point for correctly retrying if LL/SC fails.
1 point for returning the correct (i.e., old) value in `Rinc`.