CSC/ECE 506: Architecture of Parallel Computers
Sample Final Examination

This was a 150-minute open-book test. You were to answer five of the six questions. Each question was worth 20 points. If you answered all six questions, your five highest scores counted.**Question 1.** The following code is taken from the MOESI PrRd and BusRdX implementations. State-transition counters have been removed.

(a) *(2 points each)* Fill in the blanks below (and then answer the two questions on the next page).

Hint: You may need to use `sharers_exclude()` *and* `c2c_supplier()`. Answers are in red below.

```
void MOESI::PrRd(ulong addr, int processor_number) {
      ……
      cache_line * line = find_line(addr);
      if (line == NULL || line->get_state() == I){
            read_misses++;
            cache_line *newline = allocate_line(addr);
            if (c2c_supplier(addr, processor_number) > 0){
                  cache2cache++;
            }else{
                  memory_transactions++;
            }
            if (sharers_exclude(addr, processor_number) > 0){
                  I2S++;
                  newline->set_state(S);
            }else{
                  I2E++;
                  newline->set_state(E);
            }
            bus_reads++;
            sendBusRd(addr, processor_number);
      }else{
            ….
      }
}
void MOESI::BusRdX(ulong addr) {
      cache_line * line=find_line(addr);
      if (line != NULL){
            cache_state state;
            state=line->get_state();
            if (state == S){
                  invalidations++;
                  line->set_state(I);
            }else if(state == O || state == M){
                  invalidations++;
                  flushes++;
                  line->set_state(I);
            }else if(state == E){
                  invalidations++;
                  line->set_state(I);
            }
      }
}
```

(b) *(3 points)* Why do we need to separate the counting of transaction type and state setting in `PrRd()`?

Answer: Because in MOESI, only cache blocks in state O or M can be the supplier. So if there is such a cache block, it will be a cache-to-cache transaction; otherwise, main memory needs to provide the data, even if there are other caches that hold the block.

(c) *(3 points)* What is the difference between `Flush` and `FlushOpt`? Give an example from the code on the previous page.

Answer: FlushOpt exists for performance enhancement while Flush is for the correctness of the MOESI protocol. One example would be the state == E in BusRdX(). Even though it has the same code with state == S, there will be a FlushOpt there if FlushOpt is counted.

Question 2. (a) Consider a 3-processor DSM with private write-back caches. It uses a full bit-vector implementation (FBV) of the directory-based MESI protocol. For simplicity's sake, assume that a cache contains only 1 word, and we are only concerned with a single line in the cache.

The table below uses one line to represent each operation. The table columns show

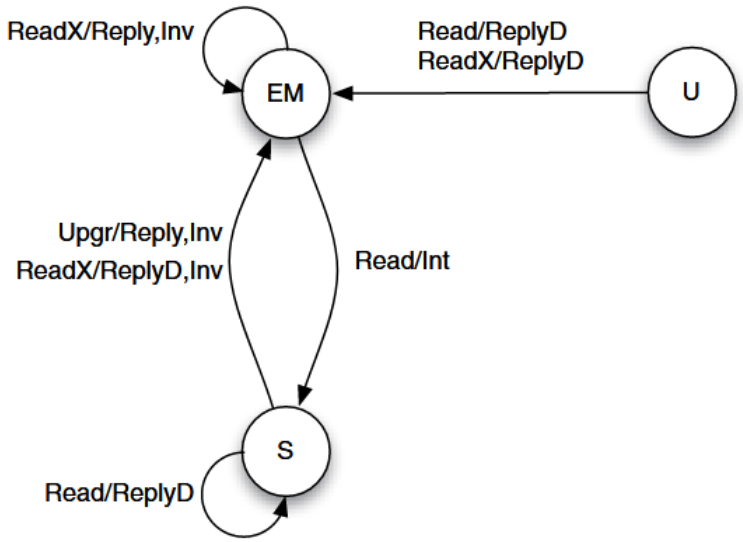
- for each cache, the state of its cache line, value in the cache line and the action induced by the protocol. A dash ("–") indicates that the line is empty.
- for the memory, the value in the corresponding memory cell, the directory state *after* the operation, and the action induced by the protocol.

Initially, the caches are empty and the corresponding memory cell contains 1.

Fill in the blanks in the table. Note that the number in the bracket refers to the order that messages are sent. For example, when processor 0 reads the location, it sends a message to the home node [1]. The home node sends back a ReplyD [2].

	Processor 0			Processor 1			Processor 2			Main Memory		
Op	State	Data	Action	State	Data	Action	State	Data	Action	Data	State,FBV	Action
R0	E	1	[1]Read→H	–	–	–	–	–	–	1	EM,100	[2]ReplyD→P0
R1	S	1	[3]Flush→H [3]Flush→P1	S	1	[1]Read→H	–	–	–	1	S,110	[2]Int→P0
R2	S	1	–	S	1	–	S	1	[1]Read→H	1	S,111	[2]ReplyD→P2
W1=4	I	-	[3]InvAck→P1	M	4	[1]Upgr→H	I	–	[3]InvAck→P1	1	EM,010	[2]Reply→P1 [2]Inv→P0` [2]Inv→P2
W2=3	I	–	–	I	–	[3]Flush→H [3]Flush→P2 [3]InvAck→P2	M	3	[1]ReadX→H	4	EM,001	[2]Reply→P2 [2]Inv→P1
R2	I	–	–	I	–	–	M	3	–	4	EM,001	–
W0=7	M	7	[1]ReadX→H	I	–	–	I	–	[3]Flush→H [3]Flush→P0 [3]InvAck→P0	3	EM,100	[2]Reply→P0 [2]Inv→P2

(b). There are several Read and ReadX processor transactions in the simplified directory-based MESI coherence protocol finite-state diagram below. There is no processor “Write” transactions to the main memory. Why?



Answer: This is a **write-back** protocol. So, processor write don’t write to the main memory. The writes

occur only on flushes when the directory is in EM state and there are any Read or ReadX requests or

when the line is written back to memory due to eviction (write-back transaction of the processor).

When the block is written back, these write transactions are generated by the cache controller and

they don’t involve the processor and so are not shown in the diagram.

(d). How can an OTB (Outstanding Transaction Buffer) help with protocol races caused by out-of-sync directory and protocol races caused by non-atomic messages?

In other words, what are the properties or end effects achieved by using an OTB when facing the two protocol races mentioned above?

Answer: OTB temporarily **stores the request messages** and waiting for home node’s **acknowledgement** of request completion. It is used to confirm the completion of certain requests.

For races caused by out-of-sync directory, the request being confirmed is **Flush**. The processor side will delay Read and ReadX requests to a block that is still being flushed. This is to ensure the block is clean when the directory receives a Read/ReadX request.

For races caused by non-atomic messages, all requests need to be confirmed. The processor delays requests to a block if the previous request to that block has not been handled. This is to ensure each request to a block is performed atomically.

Question 3.

(4 points each) For each code fragment below, put a check mark "✓ " below all the consistency models under which they are legal. For each one that is not legal, write "X". For partial credit, you must give a reason. **Notice there are 2 variables in some code fragments.**

(a)

P1:	*W*(*X*) 1	*W*(*Y*) 2			*R*(*X*) 1	*R*(*X*) 3
P2:			*R*(*Y*) 2	*W*(*X*) 3		
P3:				*R*(*X*) 1	*R*(*Y*) 2	*R*(*X*) 1

Sequential	Causal	Processor	PRAM

Answer:

Sequential	Causal	Processor	PRAM
X	X	✓	✓

P1-W(Y) 2 and P2-W(X) 3 are causally related. But this write order is violated in P3. So it is not causally consistent and therefore not sequentially consistent.

The only write order in the same processor is P1-W(X) 1 and P1-W(Y) 2. This is maintained in P3. Therefore it is PRAM consistent.

The write order to X observed is P1-W(X) 1, P2-W(X)3. This is maintained by both P1 and P3 (only sees 1 value). So it is coherent and therefore processor consistent.

(b)

P1:	*W*(*X*) 2				*R*(*X*) 2
P2:	*W*(*X*) 3	*R*(*X*) 2			
P3:		*R*(*X*) 3	*R*(*X*) 2		

Sequential	Causal	Processor	PRAM

Answer:

Sequential	Causal	Processor	PRAM
✓	✓	✓	✓

All processor see P2-W(X)3, P1-W(X)2 in that order. So it is sequentially consistent.

(c)

P1:	*W*(*X*) 1	*R*(*X*) 2	*W*(*X*) 3	
P2:			*R*(*X*) 2	*R*(*X*) 3
P3:	*W*(*X*) 2	*R*(*X*) 1		*R*(*X*)3

Sequential	Causal	Processor	PRAM

Answer:

Sequential	Causal	Processor	PRAM
X	✓	X	✓

It is not coherent because the first 2 W(X) are seen by P1 and P3 in different orders.

There are 2 write orders in question: P1-W(X) 1, P1-W(X) 3 and P3-W(X) 2, P1-W(X)3. The second pair is causally related.

P2 could observe P1-W(X) 1, P3-W(X) 2, P1-W(X) 3. P3 could observe P3-W(X) 2, P1-W(X) 1, P1-W(X) 3. 2 write orders are both observed in both sequences. So it is causally and PRAM consistent.

(d)

P1:	*W*(*X*) 0	*W*(*X*) 1	*W*(*Y*) 2	
P2:			*R*(*X*) 1	
P3:			*R*(*Y*) 2	*R*(*X*) 0

Sequential	Causal	Processor	PRAM

Answer:

Sequential	Causal	Processor	PRAM
X	✓	X	X

It is causally consistent because there are no causal related operations. It is not PRAM because the write order from P1 is violated in P3.

(e)

P1:	*W*(*X*) 0	*W*(*X*) 1	*R*(*Y*) 0	
P2:	*W*(*Y*) 0	*W*(*Y*) 1	*R*(*X*) 0	
P3:			*R*(*X*) 0	*R*(*X*) 1

Sequential	Causal	Processor	PRAM

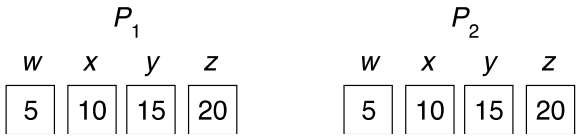
Answer:

Sequential	Causal	Processor	PRAM
X	✓	✓	✓

It is causally consistent because there are no causal related operations. It is PRAM consistent because all processor can observe 0, 1 order for both X and Y variable. It is coherent and therefore processor consistency for the same reason.

However, it is not sequential consistent because it will result in a loop if you try to order P1 and P2's operations in some sequential order. "...; each processor can order the other's write after its own read".

Question 4. This diagram shows the contents of a cache block being shared by processors P_1 and P_2, using the Dragon protocol



Given the sequence of reads and writes by P_1 and P_2 shown at the right, show the state of the cache blocks on P_1 and P_2.after each operation, indicating what processor and bus operations are performed for each operation. Assume that the cache block starts out in the *shared* state in both processors.

P_1: WRITE W = 60
P_2: READ Z
P_2: READ W
P_1: READ X
P_1: READ Z
P_2: WRITE Y = 70
P_2: WRITE X = 40
P_2: READ Y
P_1: WRITE Z = 50

Answer: The answers are underlined in the table below.

Action (using Dragon prot.)	P_1 state	P_2 state	Processor and bus operations	P_1 block after					P_2 block after			
				w	*x*	*y*	*z*		*w*	*x*	*y*	*z*
P_1: WRITE W = 60	Sm	Sc	PrWr, BusUpd(C)	60	10	15	20		60	10	15	20
P_2: READ Z	Sm	Sc	PrRd	60	10	15	20		60	10	15	20
P_2: READ W	Sm	Sc	PrRd	60	10	15	20		60	10	15	20
P_1: READ X	Sm	Sc	PrRd	60	10	15	20		60	10	15	20
P_1: READ Z	Sm	Sc	PrRd	60	10	15	20		60	10	15	20
P_2: WRITE Y = 70	Sc	Sm	PrWr, BusUpd(C)	60	10	70	20		60	10	70	20
P_2: WRITE X = 40	Sc	Sm	PrWr, BusUpd(C)	60	40	70	20		60	40	70	20
P_2: READ Y	Sc	Sm	PrRd	60	40	70	20		60	40	70	20
P_1: WRITE Z = 50	Sm	Sc	PrWr, BusUpd(C)	60	40	70	50		60	40	70	50

Question 5. In this sequence, assume that initially, **`flag0`** = **`flag1`** = 0, and **`A`** = **`B`** = **`u`** = **`v`** = **`w`** = 2. *Note that not all variables are initialized to 0!*

P0:	P1:	P2:
S0: A = 1; S1: B = 5; S2: u = B; S3: flag0 = 1;	S4: while (!flag0){}; S5: v = B − A; S6: w = A; S7: flag1 = 1;	S8: while (!flag1){}; S9: x = u × w + v;

(a) *(6 points)* What values could be produced for **`x`** under processor consistency? List *all* such values.

Answer: All writes from each processor must be seen in order by other processors. This means that **`P1`** cannot progress past S4 until it sees *all* changes written by **`P0`**. So the only possible value for **`v`** is 5-1 = 4. However, **`P2`** could see changes by **`P1`** before it sees changes by **`P0`**, meaning that u could be 2 or 5 when **`x = u * w + v`** is calculated. This would yield the value 6 or 9 for **`x`**.

(b) *(6 points)* Which writes *by different processors* are causally related in this sequence? Simply list pairs of statements (e.g., {S0, S4}) where the write in the second statement is causally related to the first. Just list the pairs that are directly related, don't write out the transitive closure. And don't write down any pairs that are executed by the same processor.

Answer: {S0, S5}, {S0, S6}, {S1, S5}, {S2, S9}, {S3, S4}, {S5, S9}, {S6, S9}, {S7, S8}

(c) *(4 points)* What values could be produced for **`x`** under causal consistency? Explain how you have determined this.

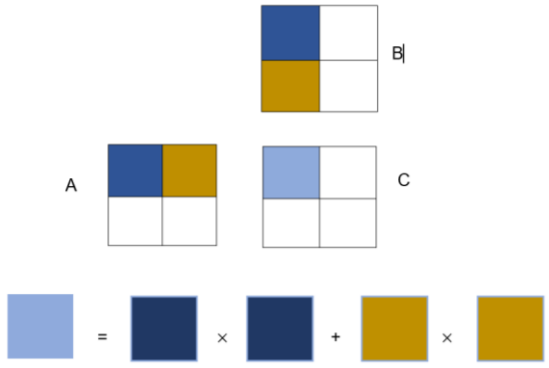
Answer: Only 9. We have a sequence of causally related writes, S0 → S1 → S3 → S4 → S7 → S8 → S9. These writes must be kept in order, assuring that S9 sees the up-to-date version of all the variables.

(d) *(4 points)* If weak ordering is in use, what is the minimum set of variables that should be treated as synchronization variables in order to assure the same results as under sequential consistency? Explain why fewer synchronization variables will not suffice.

Answer: There need to be two synchronization variables, which are **`flag0`** and **`flag1`**. If there are less than two synchronization variables, then the writes by one processor are not seen by the other processors in time so that the most recent updated values cannot be used by the other processors even when they have the updated values in their memory location. This would lead to inappropriate results.

In the above sequence, if we don't have the proper synchronization variables then the final result of **`x`** could have the values like **6, 13, 10, 14, 11, 7, 12, 6, 3, 4, 8, 5** etc., instead of the original value, which has to be **9.**

Question 6. The following CUDA code is for matrix multiplication using shared memory (this is called local memory tiling). We decompose matrices A and B into non-overlapping submatrices of size BLOCK_SIZE× BLOCK_SIZE. The thread blocks are also BLOCK_SIZE × BLOCK_SIZE. Each thread in a thread block computes a portion of the sum.



When each thread has computed this sum, we can load the next BLOCK_SIZE × BLOCK_SIZE submatrices from A and B, and continue adding the term-by-term products to our result in C. After all submatrices have been processed, we will have computed our result matrix C. The kernel code for this portion of the program is shown below. **Fill in the blanks** in the code *(2 points each)*, **place barriers** (_syncthreads()) **at the 2 places** (out of the 6 places circled) where they're needed *(2 points each)*, and **explain your reasoning** for each barrier placement *(3 points each)*.

Answer:

```
_global_ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
        // Block row and column
        int blockRow = blockIdx.y, blockCol = blockIdx.x;
```

```
(1)  // Each thread block computes one sub-matrix Csub of C
     Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

     // Each thread computes 1 element of Csub, accumulating results into Cvalue
     Cvalue = 0.0;
(2)  // Thread row and column within Csub
     int row = threadIdx.y, col = threadIdx.x;

     // Loop over all the sub-matrices of A and B required to compute
     Csub
     for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
(3)          // Get sub-matrices Asub of A and Bsub of B
             Matrix Asub = GetSubMatrix(A, blockRow, m);
             Matrix Bsub = GetSubMatrix(B, m, blockCol);

             // Shared memory used to store Asub and Bsub respectively
             _shared_ float As[BLOCK_SIZE][BLOCK_SIZE];
             _shared_ float Bs[BLOCK_SIZE][BLOCK_SIZE];

             // Load Asub and Bsub from device memory to shared memory
             // Each thread loads one element of each sub-matrix
             As[row][col] = GetElement(Asub, row, col); Bs[row][col] = GetElement(Bsub, row, col);
(4)          // Multiply Asub and Bsub together
             for (int e = 0; e < BLOCK_SIZE; ++e)
                  Cvalue += As[row][e] * Bs[e][col];
(5)  }
     // Each thread writes one element of Csub to memory
     SetElement(Csub, row, col, Cvalue);
}
```

You may identify the two places synchronization is needed and write your reasons here, or you may circle the places and write your reasoning on the previous page.

Where does the first _syncthread() go?

Answer: (4)

Why does it go here?

Answer: With the first call to _syncthreads() we insure that every entry of the submatrices of A and B have been loaded into shared memory before any thread begins its computations based on those values.

Where does the second _syncthread() go?

Answer: (5)

Why does it go here?

Answer: The second call to _syncthreads() ensures that every element of the submatrix of C has been processed before we begin loading the next submatrix of A or B into shared memory.