CSC/ECE 506: Architecture of Parallel Computers
Sample Test 2 (refactored)

This was a 120-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. This question concerns the MSI, Dragon, and Firefly protocols.
(a) *(4 points)* In Firefly, when there is a write to a *shared* block with the SharedLine not asserted, the block changes state to *valid*. Why is its state not changed to *dirty*, given that the cache block has just been modified?

Answer: Because in the Firefly protocol, all changes to *shared* blocks are written through into main memory. Thus main memory is updated when the write occurs, so the block is consistent with main memory (i.e., not *dirty*). And because the SharedLine was not asserted, the block is the only copy in any cache. So the state would go to V, which is equivalent to E in other protocols.

(b) *(4 points)* Suppose we implemented a variant of the Firefly protocol without the SharedLine. When a read or write miss occurred, data would *always* be brought in in state *shared.* In the state diagram, we would remove the transition from state *S* to state *V* for a CPU write hit with SharedLine not asserted. What would the resulting cache-coherence policy be? (It is equivalent to another policy we have discussed in class.)

Answer: Pure write-through. For blocks in state *shared*, write-through is employed. With the changes we have made, *all* blocks would *always* be in the *shared* state. Therefore, write-through would always be employed.

(c) *(8 points)* Suppose that, in a four-processor system,

1. Processor 0 reads word 100.
2. Processor 1 writes word 100.
3. Processor 2 reads word 100.
4. Processor 3 writes word 100.

At the end of this sequence, what state will the block containing word 100 have in the four caches?

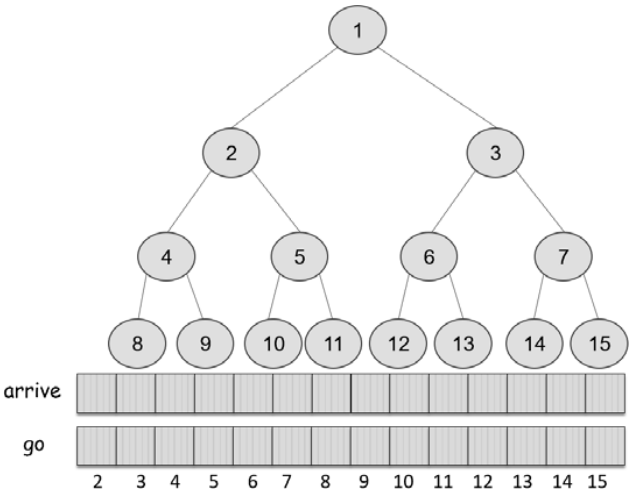
		Cache 0	Cache 1	Cache 2	Cache 3
(i)	the MSI protocol?	I	I	I	M
(ii)	the Dragon protocol?	Sc	Sc	Sc	Sm
(iii)	the Firefly protocol?	S	S	S	S

(*Answers* you were to fill in are underlined above.)

(d) *(4 points)* Based upon your answers to earlier parts of this question, describe a program that will perform better using the Dragon protocol than it would perform with the Firefly protocol.

Answer: Firefly writes through all writes to shared blocks. So when there are a lot of writes to shared blocks, a lot of memory traffic will be produced. In Dragon, these writes are only written to other caches, until another cache starts sharing the block, or the block is ejected from the cache.

Question 2. In a combining tree barrier, the processes are organized in a binary tree. Each node is "owned" by a predetermined process. Each process waits until its two children arrive, combines the results and passes them on to its parent. When the root learns that its two children have arrived, it tells its children that they can move on. This signal propagates down the tree until all the processes get the message.



(a) *(2 points)* The tree consists of a root node, interior nodes, and leaf nodes. Which nodes does an interior node *i* wait for before passing the result on to its parent?

Answer: 2*i* and 2*i*+1.

(b) *(3 points)* The code contains two arrays,

arrived[2...n]; //array that says which processes have arrived, initial values = 0

go[2...n]; //array for the signal to move-on, initial values = 0

Under what conditions will go[i] be set to 1?

Answer: When the parent of node i, i.e. i//2, has set its go array entry to 1.

(c) *(3 points per statement)* Fill in the blanks below to implement the tree-based barrier. Note that a blank may contain more than one statement.

Pseudo-code for tree-based barrier:

```
if(i == 1){ //root
await(arrived[2] = 1); arrived[2] = 0;
await(arrived[3] = 1); arrived[3] = 0;
go[2] = 1; go[3] = 1;
}

else if(i <= (n-1)/2){ // interior node
await(arrived[2*i] = 1); arrived[2*i] = 0;
// wait for the 2i+1 node and update the arrived array
await(arrived[2*i+1] = 1); arrived[2*i+1] = 0;
arrived[i] = 1;
await(go[i] = 1); go[i] = 0;
go[2*i] = 1; go[2*i+1] = 1;
}
else{  // leaf
arrived[i] =1;
await(go[i] = 1); go[i] = 0;
}
```

Question 3. Four types of misses were described in Lecture 15: cold, capacity, conflict, and coherence. Coherence misses can be classified as either true sharing or false sharing. Consider the following system.

Direct-mapped cache organization

- Cache block size = 4 words
- P_1 and P_2 each contain 2 cache lines
- MESI coherence protocol

B_1 and B_2 are two memory blocks that *map to the same cache line*. They contain the data items *P*, *Q*, *R*, *S*; and *W*, *X*, *Y*, *Z*, respectively as shown below. Each data item is one word. Each write writes one word.

B_1: B_2:

P	*Q*	*R*	*S*

W	*X*	*Y*	*Z*

Work through the following trace of memory accesses from two processors, *P*1 are *P*2. Assume that the accesses occur sequentially in the order shown below. Fill in the state of each block in each cache, and whether it is a hit, and if not, the miss type. If a block is not in a particular cache, or has been replaced from the cache, write a dash ("—").

Action	State of B_1 in P_1's cache	State of B_2 in P_1's cache	State of B_1 in P_2's cache	State of B_2 in P_2's cache	Hit, or type of miss
P_1: Write *P*	M	—	—	—	Cold
P_1: Write *Q*	M	—	—	—	Hit
P_1: Read *W*	—	E	—	—	Cold
P_2: Read *Y*	—	S	—	S	Cold
P_1: Read *R*	E	—	—	S	Conflict
P_2: Write *P*	I	—	M	—	Cold
P_2: Read *X*	I	—	—	E	Conflict
P_1: *Read P*	E	—	—	E	False sharing
P_1: Write *Z*	—	M	—	I	Conflict
P_1: Read *W*	—	M	—	I	Hit
P_2: Read *W*	—	S	—	S	False sharing
P_1: Read *Z*	—	S	—	S	Hit

A common mistake was to think that there were capacity misses in the trace. This can't be, since there are only two blocks in use, and there are two lines in the cache. All of those misses are therefore conflict misses, caused by the fact that the two blocks map to the same line.

Question 4. This question concerns inclusion policies. Suppose we have a cache that has an equal number of L1 and L2 sets. The L1 cache is 2-way, and the L2 is 4-way associative. Let's focus first on Set 0 in both the L1 and the L2.

(a) *(8 points)* Assuming that this is an exclusive cache, fill in the missing block numbers (A–G) in the table below. If a line is empty, write a dash ("—").

Ref. #	Block refer-enced	L1 cache (Set 4)		Exclusive L2 cache (Set 4)			
		MRU line	LRU line	MRU line	…		LRU line
1	A	A	—	—	—	—	—
2	B	B	A	—	—	—	—
3	D	D	B	A	—	—	—
4	C	C	D	B	A	—	—
5	A	A	C	D	B	—	—
6	D	D	A	C	B	—	—
7	B	B	D	A	C	—	—
8	G	G	B	D	A	C	—
9	F	F	G	B	D	A	C
10	C	C	F	G	B	D	A
11	H	H	E	F	G	B	D

(b) *(2 points each)* What is the number of the first reference (Ref. #) where—

- the exclusive cache would eject a block from the L2? *Answer:* Exclusive would eject when the 7th block from set 4 Is referenced. That is H, at reference 11.
- an *inclusive* cache would eject a block from the L2? *Answer:* Inclusive would eject when the 5th block from set 4 Is referenced. That is G, at reference 8.
- a *NINE* cache would eject a block from the L2? *Answer:* At reference 8 i.e G, NINE would eject a block.
- the contents of the L2 differ between an exclusive cache and a NINE cache? *Answer:* At reference 5, when A is re-referenced, it stays in the L2 with NINE, but not with exclusive.
- one of the caches (inclusive, exclusive, or NINE) would have an L2 miss while another has an L2 hit? *Answer:* Reference 8, where B is referenced. This will be an L2 hit unless the policy is inclusive.

(c) *(2 points)* Suppose that the line size is 1024 bytes and the L1 has 2^6 sets. What is the *minimum* possible distance between the first address in A and the first address in B? *Answer:* Since A and B both map to Set 4, their 6-bit index field must be the 0. The first address in each block has a 10-bit offset field of 0. So the least significant 16 bits of both addresses are all 0. Thus, A and B must be at least 216 bytes apart.

Question 5. For each of the following systems, tell whether they would need a cache-coherence protocol, a memory-consistency model, neither, or both. For partial credit, you may explain your ans

(a) A uniprocessor system that has a 2-level cache *Answer:* Neither

(b) A single-chip multicore processor with caches where memory is connected to the processors via a shared bus *Answer:* Cache coherence, but not consistency.

(c) A multiprocessor that has no caches *Answer:* Memory consistency, but not both

(d) A NUMA multiprocessor with single-level caches *Answer:* Both

(e) A data-parallel processor without caches where all communication between processors occurs through registers. *Answer:* Neither

Question 6. [*Note:* This is on material not covered on Test 2 this year.] *(5 points each)* In this question, you are asked to compute the speedup and efficiency of different kinds of parallelization. In each case,

(i) Identify the kind of parallelism for the parallelized version on the right.

(ii) compare the speedup of the parallelized version of the code with the serial version;

(iii) compute the efficiency, assuming that $N = 100$ processors are available, and

(iv) compute the efficiency, assuming that no more processors are available than can effectively be used by the parallelization. That is, if the parallelization can use only two processors, then base your efficiency calculation on two processors being available.

In all parts, you may assume that $N = 100$, and $T_{S1} = T_{S2} = \ldots = T_{S4} = 1$. Remember to include your steps for partial credit.

(a)
```
for (i = 1; i <= N; i++) {
  S1: x[i] = z - B[i]*A[i];
  S2: y[i] = a[i] + c[i];

}
```
⇒
```
for_all (i = 1; i <= N; i++) {
  S1: x[i] = z - B[i]*A[i];
  S2: y[i] = a[i] + c[i];
}
```

Answer: $T_{serial} = 100 \times (1+1) = 200$ $T_{parallel} = 1+1 = 2$

(i) DOALL (ii) Speedup = 100 (iii) Efficiency = 100/100 = 1 (iv) Efficiency = 100/100 = 1

(b)
```
for (i=1; i<=N; i++) {
  S1: a[i] = c[i] - d[i];
  S2: b[i] = b[i-1]+a[i]*d[i];
}
```
⇒
```
post(0);
for_all (i=1; i<=N; i++) {
  S1: a[i] = c[i] - d[i];
  S2: temp = a[i] * d[i];
  wait(i-1);
  S3: b[i] = b[i-1] + temp;
  post(i);
}
```

Answer: $T_{serial} = 100 \times (1+1) = 200$ $T_{parallel} = 1 + 1 + 100 \times 1 = 102$

(i) DOACROSS (ii) Speedup = 200 / 102 ≈1.96 (iii) Efficiency ≈1.96/100 ≈0.0196 (iv) Efficiency≈1.96/100

(c)
```
for (i = 1; i <= N; i++) {
  S1: a[i+1] = a[i]+d[i];
  S2: c[i+2] = x - b[i]/c[i];
}
```
⇒
```
for (i = 1; i <= N; i+=1) {
  S1: a[i+2] = a[i]+d[i]; }
for (i = 1; i <= N; i+=2) {
  S2: c[i+2] = x - b[i]/c[i]; }
for (i = 2; i <= N; i+=2) {
  S2: c[i+2] = x - b[i]/c[i]; }
```

Answer: T_{serial} = 100 × (1+1) = 200 $T_{parallel}$ = max(100,50,50) = 100

(i) Function (?) (ii) Speedup = 200/100 ≈ 2 (ii) Efficiency ≈ 2/100 = .02 (iii) Efficiency ≈ 2/3

(d)
```
for (i=1; i<=N; i++) {
  S1: d[i+1] = a[i] / d[i];
  S2: c[i+1] = c[i] * d[i] * x;
}
```
⇒
```
for (i=1; i<=N; i++) {
  d[i+1] = a[i] / d[i];
  post(i);
}
for (i=1; i<=N; i++) {
  wait(i);
  c[i+1] = c[i] * d[i] * x;
}
```

Answer: T_{serial} = 100 × (1+1) = 200 $T_{parallel}$ = 1 + 100 × 1 = 100

(i) DOPIPE (ii) Speedup = 200/100 = 2 (iii) Efficiency ≈ 2/100 = 0.02 (iv) Efficiency ≈ 2/2 = 1