ECE M116C - CS M151B Computer Architecture Systems - UCLA

Homework 6

Deadline: Wednesday, Nov. 30, 1:55 PM

Fall 2022

(Upload it to Gradescope.)

Q1. Assume that we have a multi-core system with the following instruction sequences:

Initial Values: `x1=1, x2=0, x3=0, M[A]=0, M[B]=0, flag =0`

THREAD 1	THREAD 2	THREAD 3
`M[B] = x1;`	`if(flag) {`	`while (M[B] == 0);`
`flag = 1;`	`x2 = M[B];}`	`x3 = M[A];`
`x2 = M[A];`	`M[A] = 4;`	`M[A] = 3;`

a) If the memory model is TSO, what are the possible values for `x2` and `x3`? (show the sequence by denoting numbers as T1.1, T1.2, etc. where Tx.y means the Yth instruction in Thread X, i.e., T1.1 is `M[B] = x1`).

(For each unique pair of `(x2,x3`, show only one possible sequence that would create that value. You don't need to show all possible sequences for each pair.)

SOLUTION

With total store order (TSO), the order of stores is maintained per thread when committing their own stores. We have license to move loads before stores and stores before loads, within a thread, as long as we don't violate any hazards (such as RAW).

flag is shared across these three threads. So writes to flag behave as stores. This means that T1.2 (flag = 1) can't be moved before T1.1 (M[B] = x1).

Each thread has their own copy of x1, x2, and x3. For this problem, we want to find unique (x2, x3) pairs for each thread and find a sequence of execution that causes these unique pairs.

Within Thread 1:
T1.1 < T1.2 < T1.3, T1.1 < T1.3 < T1.2, T1.3 < T1.1 < T1.2

x2 can equal 0, 3, or 4. x3 always = 0
Note: TSO defines that T1.1 has to occur before T1.2

Within Thread 2:
If flag == 1:
T2.1 < T2.2 < T2.3, T2.3 < T2.1 < T2.2
x2 can equal 1 (but not 0 as TSO prevents this!). x3 always = 0
If flag == 0:
T2.1 < T2.3, T2.3 < T2.1
x2 = 0. x3 always = 0

Within Thread 3:
T3.1 < T3.2 < T3.3
x2 always = 0. x3 can equal 0 or 4

Overall:
T1.1 < T 3.1

yellow: unique (x2, x3) pair for T1
blue: unique (x2, x3) pair for T2
purple: unique (x2, x3) pair for T3

T1		T2		T3		Global Memory Order
x2	x3	x2	x3	x2	x3	
0	0	1	0	0	4	T1.1 < T1.2 < T1.3 < T2.1 < T2.2 < T2.3 < T3.1 < T3.2 < T3.3
3	0	1	0	0	4	T1.1 < T1.2 < T2.1 < T2.2 <T2.3 < T3.1 < T3.2 < T3.3 < T1.3
4	0	1	0	0	0	T1.1 < T1.2 < T3.1 < T3.2 < T3.3 < T2.1 < T2.2 <T2.3 < T1.3
3	0	0	0	0	4	T2.1 < T2.2 <T2.3 < T1.1 < T3.1 < T3.2 < T3.3 < T1.2 < T1.3
0	0	1	0	0	4	T1.1 < T1.2 < T1.3 < T2.1 < T2.2 < T2.3 < T3.1

						< T3.2 < T3.3
4	0	1	0	0	0	T1.1 < T1.2 < T3.1 < T3.2 < T3.3 < T2.1 < T2.2 <T2.3 < T1.3
0	0	1	0	0	4	T1.1 < T1.2 < T1.3 < T2.1 < T2.2 < T2.3 < T3.1 < T3.2 < T3.3

b) Under a "release consistency" memory model, add `barriers` and `acquire(lock)`, `release (lock)` to guarantee `x2 = 0` and `x3 = 0` at the end of running this code?

SOLUTION

THREAD 1	THREAD 2	THREAD 3
`barrier(b1);`	`if(flag) {`	`barrier(b1);`
`M[B] = x1;`	`x2 = M[B];}`	`while (M[B] == 0);`
`flag = 1;`	`barrier(b1);`	`x3 = M[A];`
`x2 = M[A];`	`barrier(b2);`	`barrier(b2);`
`barrier(b2);`	`M[A] = 4;`	`M[A] = 3;`

Barriers are used to synchronize between threads. Essentially, barriers stipulate that different threads must all reach a specified point in their execution before each thread can continue on.

To ensure that x2 = 0 and x3 = 0 for every thread at the end of execution, we can first use barrier 1 (b1) in all threads to allow Thread 2's x2 to be set to zero (M[B] = 0) before Thread 1 sets M[B] = x1. We can then use barrier 2 (b2) to allow Thread 1's x2 to be set to zero (M[A] = 0) and Thread 3's x3 to be set to zero (M[A] = 0) before Thread 2 sets M[A] = 4 and Thread 3 sets M[A] = 3. Thread 1's x3 will always = 0, Thread 2's x3 will always = 0, and Thread 3's x2 will always = 0.

This is just one solution, and there are many more, even using locks. While locks could be used here, locks don't guarantee anything about execution order, which barriers provide. To make x2 and x3=0 at the end of execution, it made sense to use barriers here to provide a flow of execution that would end in this result.

Q2. Assume we have a MOESI coherency protocol. Assume we have 4 cores and all cores are sharing the memory line `M[Z]`. Assume that the initial state for all of them are (I)nvalid.

Complete the following table (show the writebacks with *, and if one cache forwards a value to another cache show it by ^):

Access	P1	P2	P3	P4
R1				
W1				
R2				
W3				
R3				
R4				
R1				
W1				
R1				
W2				

SOLUTION

Access	P1	P2	P3	P4
R1	E	I	I	I
W1	M	I	I	I
R2	O^	S	I	I
W3	I*	I	M	I
R3	I	I	M	I
R4	I	I	O^	S
R1	S	I	O^	S
W1	M	I	I*	I
R1	M	I	I	I
W2	I*	M	I	I

Access
R1: P1 is exclusive, as no other cores have accessed this address
W1: P1 is modified, as P1 has written to this address
R2: As P2 reads, P1 becomes the “owner” and forwards the data to P2. P2 enters the shared state
W3: As P3 writes, P3 enters the modified state, and P1 and P2 are invalidated. P1 writes back it’s data first
R3: P3 stays in the modified state
R4: P3 becomes the “owner” and forwards the data to P4. P4 enters the shared state

R1: P3 forwards the data to P1, which enters the shared state
W1: P1 writes and becomes modified, invalidating P3 and P4
R1: P1 stays in the modified state
W2: P2 enters the modified state, invalidating P1. P1 writes back