CS 5334 Spring 2021

Week 5 Homework – Memory Consistency and Cache Coherence

Due Sunday, February 21 at 11:59pm

10 points – Each problem is worth 2 points

You can find the code for problems 3, 4, and 5 at <https://github.com/mooresv/CS5334/tree/main/hw/week5>

1. Addresses are represented in binary. Initially all cache lines are invalid and all memory locations contain 0. Assume a cache hit takes one cycle (for both read and write operations), transferring a cache line (either fetch or write back) takes 8 cycles, and a request to use a cache line exclusively or upgrade to exclusive use takes 2 cycles. Fill in the table below with the state of the cache line on each core and the latency in cycles for each action.

All of them are at the same cache line

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Action | P0 | P1 | P2 | P3 | Latency |
| *R0*(00000000) | E | I | I | I | 11 cycles |
| *R1*(00000010) | S | S | I | I | 9 cycles |
| *R2*(00000011) | S | S | S | I | 9 cycles |
| *W3*(00000100) | I | I | I | M | 11 cycles |
| *W3*(00000101) | I | I | I | M | 11 cycles |
| *R0*(00000100) | S | I | I | S | 9 cycles |
| *W0*(00000101) | M | I | I | I | 11 cycles |

1. In the execution sequences shown below, Pn: W(x,v) means that processor/core n writes value v to memory location x, and Pn: R(x,v) means that processor n reads value v from x. Assume all memory locations are initially set to 0. For each execution, indicate if it is sequentially consistent and if so, give sequentially consistent total ordering of the operations. If it is not sequentially consistent, explain why not.
   1. P1: W(x,1);

P2: R(x,0); R(x,1)

SC allowed – P2: R(x,0); P1: W(x,1); P2: R(x,1); the zero value was there beforehand.

* 1. P1: W(x,1);

P2: R(x,1); R(x,0);

Non-SC allowed – P2: R(x,0); is outputs a read of 0 after a write of 1 and read of 1 occurred.

* 1. P1: W(x,1);

P2: W(x,2);

P3: R(x,1); R(x,2);

SC allowed, result complies with ordering P1: W(x,1); P3: R(x,1); P2: W(x,2); P3: R(x,2);

* 1. P1: W(x,1);

P2: W(x,2);

P3: R(x,2); R(x,1);

SC allowed, result complies with ordering P2: W(x,2); P3: R(x,2); P1: W(x,1); P3: R(x,1);

* 1. P1: W(x,1);

P2: W(x,2);

P3: R(x,2); R(x,1);

P4: R(x,1); R(x,2);

Non-SC allowed since after a writing a read is found to be the old value before the writing occurred.

* 1. P1: W(x,1); R(x,1); R(y,0);

P2: W(y,1); R(y,1); R(x,1);

P3: R(x,1); R(y,0);

P4: R(y,0); R(x,0);

SC allowed, total order P4: R(y,0); P4: R(x,0); P1: W(x,1); P1: R(x,1); P3: R(x,1); P3: R(y,0); P1: R(y,0); P2: W(y,1); P2: R(y,1); P2: R(x,1);

* 1. P1: W(x,1); R(x,1); R(y,0);

P2: W(y,1); R(y,1); R(x,1);

P3: R(y,1); R(x,0);

Non SC allowed x = 0 and y=0 are read after writes are done for X and Y to make them 1

1. Consider the Pthreads code in ordering.cpp. Compile the code on Bridges-2 with the following command line (use the default gcc 8.3.1):

gcc -o ordering ordering.cpp -lpthread

Run the code on Bridges-2 until you observe a few reorderings and then kill it with Ctrl-C (the code executes an infinite loop).

1. Explain why the reorderings are occurring.

As the threads execute the copies of the modified variables are stored in their own private buffers. When the execution ends, and the values are actually modified such values may sometimes actually reach to be evaluated as they should be in a sequentially consistent fashion, other times what happens is that the values are updated in such a way that the changes are never reflected to each other (since the code executes in multiple cores in parallel).

1. Modify the code to produce only sequentially consistent executions. Can reorderings still occur? Why or why not?

There are two possible modifications, the first is to re-arrange the use of the semaphore so that we either see the case of X=1 and Y=0 or Y=0 and X=1. The second is to modify the code by including fences so that we may actually force writes and then allow reads so that we always see the case of X=1 and Y=1.

Text

Description automatically generated

Since our condition for indicating a reordering is that both registers are set to 0 in none of the two cases we will see reordering.

1. Consider the OpenMP code in omp\_ordering.c. Compile the code on Bridges 2 using the following command line:

gcc -fopenmp -o omp\_ordering omp\_ordering.c

Run the code on Bridges-2 using three threads until you observe a few reorderings and then kill it with Ctrl-C (the code executes an infinite loop).

1. Explain why the reorderings are occurring.

Similar to problem 3 we have a set of two threads that are working on updating variables, due to the implementation, the buffer on which the modifications of variables are kept is not “flushed” (sent to cache so invalidations occur and memory updates also occur) in order for both c and d to have the new/modified value in the scope of the main thread’s memory.

1. Modify the code to produce only sequentially consistent executions. Can reorderings still occur? Why or why not?

The modification is to give a flush to the variables to synchronize the values of variables between threads, that of course costs heavily on performance but makes consistency to be present. So, the modification will look like this:

A picture containing text, black, plaque, screenshot

Description automatically generated

With these modifications to the original code there can’t be a reordering, if we were to reduce the flush lines used then there could be cases where reordering occurs but not right now.

1. A serial producer-consumer code is in prod\_cons.c and an OpenMP-parallelized version that uses two threads is in omp\_prod\_cons.c. Compile and run both codes on Bridges-2.
   1. Assume results from the serial code are correct. Is the parallel code guaranteed to always produce correct results on any multicore processor with a standard-conforming OpenMP implementation? Why or why not?

No, it is not guaranteed, because even though at the high level we set the flag to 1 after A is populated there may be some memory models that allow for a write before a read, hence we could allow a set of the flag before the finishing the call to fill\_rand(A), same logic is applicable to the while and sum sections.

* 1. If the parallel code is not correct, fix it so that it is guaranteed to run correctly on any platform with a standards-conforming implementation of OpenMP.

I believe the fix is to put a flush right after the call to fill\_rand so that no reordering of the operations can happen. And we should also place an OMP flush right after the while loop.

Text

Description automatically generated