CS 6810 - Assignment 10 *

Anirudh Narasimhamurthy(u0941400)

April 26, 2015

1 Consistency Models

We are given a multi-threaded program containing two threads T1 and T2 and are asked to report the possible values of the variable A during the execution. The threads contain the following instructions:

Thread 1:
a: B=8;
b: A=4;

Thread 2:
A: if (A+B¿ 10)
B: then A=A+B;
C: print A

We are also given the information, that before these threads start executing. A and B are both initialized to zero.

Since we are in the context of sequential consistency models, we have the following assumptions:
1. Instructions are executed atomically
2. Program order is followed for instuctions within a single thread.
3. Instructions from different threads can be interleaved arbitrarily.

With the above assumptions and the model, the total no of possible/ valid combinations of instructions execution in our multi-threaded program would be $\binom{4}{2} = 6$. This is because Thread 1 has two instructions and I am assuming that the If and Then statements in Thread 2 would always occur together as it would make more logical sense.

*CS 6810 ; Spring 2015

Based on the different combinations, the values of "print A" which we obtain are tabulated below:

Combination	Value of A	Explanation
abABC	12	A+B =12 > 10, A=A+B
ABCab	0	A+B=0 < 10, A=0
ABabC	4	A+B=0 < 10 , A=4 gets executed and then print statement executes
ABaCb	0	A+B=0 < 10, B=4
aABbC	4	A+B=8 < 12, A=4 gets executed and then print statement executes
aABCb	0	A+B=8 < 12, print gets executed and then A=4 gets executed

Table 1: Valid outputs of A

Thus the valid outputs of A for sequentially consistent execution of the program is **{0, 4, 12}**

Coarse-Grained Lock Scenario

When the code is encapsulated within lock and unlock statements as shown below, the valid values of A would change.

Thread 1:
lock(L1)
a: B=8;
b: A=4;
unlock(L1)

Thread 2:
lock(L1)
A: if (A+B¿ 10)
B: then A=A+B;
C: print A
unlock(L1)

The code shwon above makes use of coarse grained lock and in this scenario only one of the two threads would be able to obtain a lock and proceed with its execution. When a thread finishes its execution, it releases the lock and the other thread obtains the lock and begins its execution.

Given the situation, this leads to two possible scenarios in our program.

Case 1: Thread 1 first acquires lock and then Thread 2 acquires the lock

In this case, the instructions a and b get executed first and then when Thread 2 starts executing, the if condition in A becomes true as A+B=12 > 10 and so A=A+B, which would give us the value of 12.

Case 2: Thread 2 first acquires lock and then Thread 1 acquires the lock

In this case instructions ABC get executed first and then instructions ab get executed. When Instruction A executes, A=0 and B=0 and so the if condition evaluates to False. So the statement print A will print the latest value of A which is 0.

Thus the valid outputs for A when the code is encapsulated within lock and unlock statements is **0 and 12**

2 Consistency Models

How are modern processors able to provide sequentially consistent results for multi-threaded programs ?

Modern processors provide sequential consistency by implementing or making sure the following assumptions are working right.

1. Instructions are executed **atomically.** This ensures that before one instruction completes its execution another instruction cannot change or modify the same variable which the previous instruction was still working on and allow the previous instruction to commit. Atomicity ensures instructions are executed either in complete or rolled or aborted back.

2. The **program order** is followed while executing instructions within a single thread. However instructions from threads can be interleaved arbitrarily and this combined with the atomicity property ensures **write serialization** happens correctly. That ensures everyone has seen an update before the value for that variable is read. This ensures consistency.

3. The cache coherence protocol implements this sequential consistency for modern processors. The process of making sure we receive Acknowledgements whenever we send out invalidate signals to other processors or threads makes sure that the correct value is propagated and sequential consistency is maintained across the multi threaded execution of the program.

This is how modern processors are able to provide sequentially consistent results for multi-threaded programs.

How are modern processors able to provide high performance for multi-threaded programs ?

Sequential consistency model can be disrupted by hardware innovations such as out of order executions which generally make our execution faster and saves us time. So in a way sequential consistency model slows down the execution.

But modern processors have the possibility of implementing few hardware optimizations to make sure it provides high perfomance even in this setting. Two such methods are mentioned below:

Issue Loads speculatively
We could still take advantage of faster execution offered by reorder buffer by this method. Since load instructions generally finish faster than store instructions which requires to obtain permissions to peform writes, we can allow the loads to be issued speculatively and proceed with the value.

After the store gets committed and load becomes the oldest instruction in the LSQ, re-issue the Load again. If the new value given by the Load matches the previous value, then our speculation was correct and we would provide better performance. If the new value was different from the speculated value, then we will have to re-issue the Load agin and execute all its dependent instructions to maintain sequential consistency.

Even in this case, it is much better than having to wait for the Store to commit and then start the Load instruction by which time we would have lost a lot of time and performance might be slow.

Acquire permissions speculatively
Store takes a long time to complete once it gets to the top of the re-order buffer because it has to acquire permissions to write the block and them performs the write. But if we could obtain the permissions earlier, then that would improve the performance. Also speculating on permissions is safe as we would not be changing any result. Once we have the permisisons then it would take very less time to make updates as we would have it in our cache and we can modify the locaaly cached copy much faster.

Modern processors either implement the above hardware optimizations or sometimes even prefer to work in a relaxed consistency model as opposed to sequential consistency model to provide higher performance.

3 Transactional Memory

How can a lazy-lazy transcational memory cause starvation in some programs ?

Consider the following example:

Transaction T1:
Begin Transaction
{
Read B
....
....
....
....
....
....
}
End Transaction

Transaction T2:
While
{
Begin Transaction
{
Write B
}
End Transaction
}

Assuming the transaction in T1 is extremely long which takes long time to execute and assuming transaction T2 is very short and finishes once it perfoms a Write B operation. So it acquires the token earlier than T1 and then sends the broadcast message to T1 asking it to invalidate its copy of B. It then commits and then starts the while loop again.

SInce it is a smaller transaction every time it gets executed it acquires the token and sends broadcast to T1 asking it to invalidate its copy of B and this results in T1 being continously aborted. This could continue forever and ever until the smaller transaction releases the token or stops its execution leading to a starvation for transaction T1 as it would not be able to complete its execution for a long time.

Thus because it performs lazy conflict detection or in other words deferring the conflict detection until atleast one of the transaction commits and since it performs a lazy versioning, the lazy-lazy transactional memory causes starvation in this case. The process of acquiring token to complete or abort a transcation makes it kind of lock like feature which we knew had its drawbacks of slow performance or leading to deadlocks.

Solution to starvation problem in lazy-lazy transactional memory

The solution in this case would be for Transaction T1 to realize its being aborted frequently and so it can send a request to central arbiter for the Commit Token say after it has been aborted 'M' number of times. And when it recieves one, it ensures that no other transaction can commit before T1. So the transaction T1 executes all of its instructions and goes to the very end. T2 in this case would have to wait for T1 to finish and can then proceed. T1 then releases the commit token and then T2 can acquire it from the central arbiter and proceed with it.

Again this degenerates into lock based model where only one transaction can do its work at a time, thereby not allwoing parallel processing to take place. But this is still a much better and neater version of handling overflows than what we would have had in lock based model and hence this could help avoid starvation in lazy-lazy transactional memory implementation.