ECE459: Programming for Performance Winter 2019

Lecture 15 — Memory Consistency

Patrick Lam and Jeff Zarnett *2019-01-04*

Memory Consistency, Memory Barriers, and Reordering

Today we'll talk a bit more about memory consistency, memory barriers and reordering in general. We'll start with instruction reordering by the CPU and move on to reordering initiated by the compiler. I'll also touch on some CPU instructions for atomic operations.

Memory Consistency. In a sequential program, you expect things to happen in the order that you wrote them. So, consider this code, where variables are initialized to 0:

```
T1: x = 1; r1 = y;
T2: y = 1; r2 = x;
```

We would expect that we would always query the memory and get a state where some subset of these partially-ordered statements would have executed. This is the *sequentially consistent* memory model.

> "... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." — Leslie Lamport

What are the possible values for the variables?

Another view of sequential consistency:

- each thread induces an *execution trace*.
- always, the program has executed some prefix of each thread's trace.

It turns out that sequential consistency is too expensive to implement. (Think how much coordination is needed to get a few people to agree on where to go for lunch; now try to get a group of people to agree on what order things happened in. Right. Now imagine it's a disagreement between threads so they don't have the ability to negotiate.) So most systems actually implement weaker memory models, such that both `r1` and `r2` might end up unchanged. Recall the **flush** example from last time.

Reordering. Compilers and processors may reorder non-interfering memory operations within a thread. For instance, the two statements in `T1` appear to be independent, so it's OK to execute them—or, equivalently, to publish their results to other threads—in either order. Reordering is one of the major tools that compilers use to speed up code.

When is reordering a problem? Spin locks, as we'll see below.

Memory Consistency Models

Here are some flavours of memory consistency models:

- Sequential consistency: no reordering of loads/stores.
- Sequential consistency for datarace-free programs: if your program has no data races, then sequential consistency.
- Relaxed consistency (only some types of reorderings):
  - Loads can be reordered after loads/stores; and
  - Stores can be reordered after loads/stores.
- Weak consistency: any reordering is possible.

In any case, **reorderings** only allowed if they look safe in current context (i.e. they reorder independent memory addresses). That can still be problematic, though.

Compilers and reordering. When it can prove that a reordering is safe with respect to the programming language semantics, the **compiler** may reorder instructions (so it's not just the hardware).

Reordering example. Say we want thread 1 to print a value set in thread 2.

```
                          f = 0

/* thread 1 */                          /* thread 2 */
while (f == 0) /* spin */;              x = 42;
printf("%d", x);                        f = 1;
```

If thread 2 reorders its instructions, will we get our intended result? *No!*

Memory Barriers

We previously talked about OpenMP barriers: at a `#pragma omp barrier`, all threads pause, until all of the threads reach the barrier. Lots of OpenMP directives come with implicit barriers unless you add `nowait`.

A rather different type of barrier is a *memory barrier* or *fence*. This type of barrier prevents reordering, or, equivalently, ensures that memory operations become visible in the right order. A memory barrier ensures that no access occuring after the barrier becomes visible to the system, or takes effect, until after all accesses before the barrier become visible.

The x86 architecture defines the following types of memory barriers:

- `mfence`. All loads and stores before the barrier become visible before any loads and stores after the barrier become visible.
- `sfence`. All stores before the barrier become visible before all stores after the barrier become visible.
- `lfence`. All loads before the barrier become visible before all loads after the barrier become visible.

Note, however, that while an `sfence` makes the stores visible, another CPU will have to execute an `lfence` or `mfence` to read the stores in the right order.

Consider the example again:

```
                        f = 0

/* thread 1 */                      /* thread 2 */
while (f == 0) /* spin */;          x = 42;
// memory fence                     // memory fence
printf("%d", x);                    f = 1;
```

This now prevents reordering, and we get the expected result.

You can use the `mfence` instruction to implement *acquire barriers* and *release barriers*. An acquire barrier ensures that memory operations after a thread obtains the mutex doesn't become visible until after the thread actually obtains the mutex. The release barrier similarly ensures that accesses before the mutex release don't get reordered to after the mutex release. Note that it is safe to reorder accesses after the mutex release and put them before the release.

Preventing Memory Reordering in Programs: Compiler Barriers. First: Don't use volatile in C/C++ on variables [Inc08]. Remember that the volatile keyword is supposed to tell the compiler not to put the value in a register. That does NOT make it a synchronization construct. If you use the correct synchronization primitives, you will get the behaviour you want. However, you can prevent reordering using compiler-specific calls.

- Microsoft Visual Studio C++ Compiler:

  ```
  _ReadWriteBarrier()
  ```

- Intel Compiler:

  ```
      __memory_barrier()
  ```

- GNU Compiler:

  ```
  __asm__ __volatile__ ("" ::: "memory");
  ```

The compiler also shouldn't reorder across e.g. Pthreads mutex calls.

Aside: `gcc` Inline Assembly. Just as an aside, here's `gcc`'s inline assembly format

```
__asm__ ( assembler template
       : output operands                /* optional */
       : input operands                 /* optional */
       : list of clobbered registers    /* optional */
       );
```

Note that we've just seen **__volatile__** with __asm__. This isn't the same as the normal C volatile. It means:

- The compiler may not reorder this assembly code and put it somewhere else in the program.

Back to Memory Reordering in Programs. Fortunately, an OpenMP **flush** (or, better yet, mutexes) also preserve the order of variable accesses. That is, it prevents reordering from both the compiler and hardware. For GNU, flush is available as `__sync_synchronize();`

`volatile`**.** This qualifier ensures that the code does an actual read from a variable every time it asks for one (i.e. the compiler can't optimize away the read). It does not prevent re-ordering nor does it protect against races.

Note: proper use of memory fences makes `volatile` not very useful (again, `volatile` is not meant to help with threading, and will have a different behaviour for threading on different compilers/hardware).

Atomic Operations

We saw the **atomic** directive in OpenMP as well as C++11 atomics. Most OpenMP atomic expressions map to atomic hardware instructions. However, other atomic instructions exist, and we've seen the C++11 ones earlier as well.

Compare and Swap. This operation is also called **compare and exchange** (implemented by the `cmpxchg` instruction on x86). Here's some pseudocode for it.

```
int compare_and_swap (int* reg, int oldval, int newval)
{
  int old_reg_val = *reg;
  if (old_reg_val == oldval)
     *reg = newval;
  return old_reg_val;
}
```

Afterwards, you can check if the CAS returned `oldval`. If it did, you know you changed it.

Implementing a Spinlock. You can use compare-and-swap to implement spinlock:

```
void spinlock_init(int* lock) { *lock = 0; }

void spinlock_lock(int* lock) {
    while(compare_and_swap(lock, 0, 1) != 0) {}
    __asm__ ("mfence");
}

void spinlock_unlock(int* lock) {
    __asm__ ("mfence");
    *lock = 0;
}
```

You'll see **cmpxchg** quite frequently in the Linux kernel code.

ABA Problem

Sometimes you'll read a location twice. If the value is the same both times, nothing has changed, right?

No. This is an **ABA problem**.

The ABA problem is not any sort of acronym nor a reference to this [Abb74]. It's a value that is A, then changed to B, then changed back to A. The ABA problem is a big mess for the designer of lock-free Compare-And-Swap routines. This sequence will give some example of how this might happen [DPS10]:

1. P_1 reads A_i from location L_i.
2. P_k interrupts P_1; P_k stores the value B into L_i.
3. P_j stores the value A_i into L_i.
4. P_1 resumes; it executes a false positive CAS.

It's a "false positive" because P_1's compare-and-swap operation succeeds even though the value at L_i has been modified in the meantime. If this doesn't seem like a bad thing, consider this. If you have a data structure that will

be accessed by multiple threads, you might be controlling access to it by the compare-and-swap routine. What should happen is the algorithm should keep trying until the data structure in question has not been modified by any other thread in the meantime. But with a false positive we get the impression that things didn't change, even though they really did.

You can combat this by "tagging": modify value with a nonce upon each write. You can also keep the value separately from the nonce; double compare and swap atomically swaps both value and nonce.

C/C++11 Memory Model

We have talked about memory models in the context of OpenMP. Let's talk about the core languages now—that is, C and C++ [Lov14, BA08]—when not using OpenMP.

What outputs are possible from this example?

Thread 1:

```
foo = 7;
bar = 42;
```

Thread 2:

```
printf("%d\n", foo);
printf("%d\n", bar);
```

You might think "undefined", but actually it's worse than that. The C11 and C++11 language definitions don't even say what a thread is. Of course, there is Pthreads, but that is a library, not the language itself. So you can't even ask this question when talking about pre-C11/C++11 versions of C and C++. Well. Okay. You can ask, but the question makes no sense. Sort of like putting your hand up in class and saying "Where did you bury your oranges?". It's a syntactically valid question and it describes something that's possible, but it still makes no sense.

The older standards made no reference to any kind of CPU, memory architecture, cache strategy, or anything like that. Which on the one hand is nice and general, but on the other hand, leads to problems. The "abstract machine" that the C and C++ standards refer to is inherently single threaded, making it actually impossible to write a portable multithreaded C or C++ program [Lov14]. The part that's impossible is that word "portable" – people write multithreaded C and C++ programs all the time (in this class, even) but they have system specific code and implementation-defined behaviour. Open up a pthreads library or some equivalent and sure enough, you will find something architecture specific in there.

C++11 (and C11) have improved the situation, though. There is actually a memory model (based on an abstract machine) and threading primitives such as mutexes, atomics, and memory barriers—the concepts that we have seen in this course. Now there are rules! Yes. We like rules. Okay. I [JZ], at least, like rules. C++11 defines how a compiler can generate code that accesses memory even when there is concurrency. There are also standard mutex operations and atomics and barriers and all those lovely things.

Now, we can ask the question about the behaviour of the above example. It does have undefined behaviour, since there is contended access to the variables `foo` and `bar`. How can we fix that?

Atomics. A good exam question: if `foo` is atomic, what are the possible outputs?

Thread 1:

```
foo.store(7);
bar.store(42);
```

Thread 2:

```
printf("%d\n", foo.load());
printf("%d\n", bar.load());
```

Alright, we have some defined behaviour now. Honestly, it depends how these things are scheduled, but the answer is one of the following set: {0/0, 7/42, 7/0, 0/42}. The answer depends on how they are interleaved. But at least we get some certainty that the output will be one of those four things and there's no chance of garbage because the print takes place during an assignment operation.

We probably still don't like this because we don't have mutual exclusion here and we can get several different answers, some of which are probably "wrong" (for whatever definition of a correct answer is), but at least our set

of potential wrong answers is smaller. So that's a start. Compilers have to follow the new rules in generating code, so their output will behave as if the architecture followed the standard memory model. That's something.

Good C++ Practice

Lots of people use postfix (`i++`) out of habit, but prefix (`++i`) is better. In C, this isn't a problem. In some languages (like C++), it can be.

Why? Overloading. In C++, you can overload the `++` and `--` operators.

```
class X {
public:
  X& operator++();
  const X operator++(int);
};
```

```
X x;
++x; // x.operator++();
x++; // x.operator++(0);
```

Prefix is also known as **increment and fetch**, and might be implemented like this:

```
X& X::operator++() {
  *this += 1;
  return *this;
}
```

Postfix is also known as **fetch and increment**. Note that you have to make a copy of the old value:

```
const X X::operator++(int) {
  const X old = *this;
  ++(*this);
  return old;
}
```

So, if you're the least concerned about efficiency (and why else would you be taking programming for performance?), always use *prefix* increments/decrements instead of defaulting to postfix. This isn't really an issue if the operator is in a statement all on its own (e.g. a standalone line, or the last part of a for loop) because the compiler is (presumably) smart enough to know that this can be optimized as the return value is not assigned. Only use `postfix` when you really mean it, to be on the safe side.

Mind you, if you're doing something like `array[i++]` or something similarly "clever", you might want to think twice about this. There's a lot of potential for error or misunderstanding in a code review. Remember, clever is hard to grep for.

References

[Abb74] Abba. Waterloo, 1974. Online; accessed 14-December-2015. URL: `https://www.youtube.com/watch?v=Sj_9CiNkkn4`.

[BA08] Hans J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model, 2008. Online; accessed 16-December-2015. URL: `http://rsim.cs.illinois.edu/Pubs/08PLDI.pdf`.

[DPS10] Damian Dechev, Peter Pirkelbauer, and Bjarne Stoustroup. Understanding and effectively preventing the aba problem in descriptor-based lock-free designs, 2010. Online; accessed 14-December-2015. URL: `http://www.stroustrup.com/isorc2010.pdf`.

[Inc08] Stack Exchange Inc. Using C/Pthreads: do shared variables need to be volatile?, 2008. Online; accessed 14-December-2015. URL: `http://stackoverflow.com/questions/78172/using-c-pthreads-do-shared-variables-need-to-be-volatile`.

[Lov14] Robert Love. How are the threading and memory models different in c++ as compared to c?, 2014. Online; accessed 16-December-2015. URL: `http://www.quora.com/C++-programming-language/How-are-the-threading-and-memory-models-different-in-C++-as-compared-to-C`.