Implementing OO Languages under a Weak Memory Order

William Pugh[*], Univ. of Maryland, College Park, pugh@cs.umd.edu

Keywords: Data race, weak memory model, thread, synchronization, Java, OO implementation

Introduction

Many shared memory multiprocessors implement a weak memory model. These weak memory models might allow higher performance, but also produce surprising results when multithreaded programs are not properly synchronized.

Although these problems can arise in many programming languages, they are particular severe in object oriented languages and in languages that make safety guarantees. The difficulty for object oriented programs is due to the number of "hidden" data structures manipulated by the runtime system (e.g., the virtual function table) and the richer mental model programmers have of objects (and their surprise when these models are violated by improperly synchronized programs).

An active discussion is occurring on these issues; this is a snapshot of the discussion.

Weak memory models

A memory model describes how different threads/processors can see their memory actions interleave with those of other processors. A strong memory model, such as sequential consistency, imposes very strict constraints.

There are many weak memory models. They all tend to support and/or need explicit memory barriers or acquire/release operations. A possible implementation/intuition is shown in Figure 1. Each processor reads and writes to a local cache. Updated memory locations may be flushed from the cache to main memory at any time, and the cache can fill a memory location from main memory at any time. When a processor does an acquire/lock operation, it must reload the cache from main memory, and when a processor does a release/unlock operation, it must flush all modified memory locations from the cache to main memory. Multiprocessors based on the DEC Alpha and the Intel Merced chip provide this kind of weak memory order.

[*]This work was supported National Science Foundation grants ACI9720199 and CCR9619808.

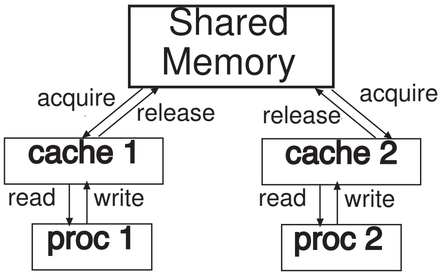

Figure 1: Implementation of a Weak Memory Model

```
Initially:
class Foo {
  final int x;
  int f(int y) { return x+y; }
  Foo(int i) { x = i; }
  static Foo bar;
  }
Foo.bar = new Foo(1);
```

On Processor 1:	On Processor 2:
`Foo.bar = new Foo(2);`	`int z = Foo.bar.f(3);`

Figure 2: Unsynchronized access

What can go wrong?

Consider execution of Figure 2 on a multiprocessor with a weak memory model; the processors have unsynchronized access to `Foo.bar`. The initial code is executed first and all processors see it. Next, it happens that processor 1 executes its code first. Furthermore, the cache happens to flush the modification of Foo.bar back to main memory, but none of the other writes. Then, when processor 2 executes its code, it loads its cache only with the new value of `Foo.bar` from main memory (finding old values in the cache for all other memory locations).

What unexpected things could happen? When processor 2 executes `Foo::f` and reads the `x` field of the object allocated by processor 1, it won't see the value that the `x` field was initialized to by processor 1. It could read garbage: an arbitrary value. Since `x` is only an integer value, this is only moderately bad. If `x` were a reference/pointer, then seeing a garbage value would violate type safety and make any kind of security/safety guarantee impossible.

We can allocate objects out of memory that all processors agree has been zeroed. Essentially, you would zero memory during garbage collection, and then have all processors perform a memory barrier/acquire before restarting after a garbage collection. This would ensure that if processor 2 sees a stale value for the `x` field of the object allocated by processor 1, it will see zero/null. For references/pointers, this will ensure type safety.

We could require that a processor perform a release/flush operation between creating an object and publishing a reference to that object (by publishing, I mean store a reference to the object in a place where it might be read by other threads). A compiler could easily figure out where such flush operations are required (e.g., after object initialization) and the cost of doing such flushes would likely be small.

However, it isn't enough. Under a weak memory model, processor 2 must also do a barrier/acquire operation to see all of the writes sent to main memory by processor 1. The problem is that there isn't anything in the code executed by processor 2 to suggest that we might be reading a reference to an object created by another processor.

Perhaps we should just decide that seeing a zero/null value for a field is OK. In Java, that is the default value for a field, and if you allow an object to escape before it is properly initialized, that it what you will see anyway. (Some people are horrified by this idea, but let's run with it for the moment).

However, in an OO environment, we also have to consider the object header fields. For example, when processor 2 reads the `vtbl` (virtual method table) entry from the object referenced by `Foo.bar`, it might see null. Dispatching the `Foo::f()` method could result in a `SIGSEGV` fault crashing your virtual machine; this clearly should *not* be considered acceptable.

Other information must be considered suspect in a multithreaded environment. In Java, the length of an array might be seen as zero. In C++, the pointer to a virtual base class might be null.

There are even worse problems if you consider dynamic class loading. Consider what happens if processor 1, rather than just creating a instance of a class that processor 2 already knows about, loads an entirely new class Faz (a subclass of Foo that overrides `f()`), compiles native code for `Faz::f()` from bytecode, creates an instance of Faz, and then stores a reference to that instance in `Foo.bar`). There is still nothing in the code being executed by processor 2 to indicate that it will need to synchronize. However, any of the memory locations read by processor 2 might be null. Even if the reference to the virtual function table isn't null, an entry of the `vtbl` could be null. Processor 2 might not see the native code generated by processor 1.

What makes this particular difficult is that just *one* of the memory locations read by processor 2 could be stale, even though all the others see properly updated values. Just checking to see if you got a valid pointer to a `vtbl` won't suffice.

The semantics of data races

Few programming languages have defined the semantics of programs that contain unsynchronized access to shared data. Ada and Modula3 define a multithreaded semantics, but simply say that it is erroneous to have unsynchronized access to shared data.

The Java Language Specification provided [GJS96, Chap. 17] a semantics for threads, locks and memory. Unfortunately, that specification has proven to be fatally flawed [Pug99]. An intended feature (memory coherence) turned out to have the unintended consequence of prohibiting standard compiler optimizations essential for good performance and done by virtually every JVM. It also unintentionally prohibits many bytecode to bytecode transformations and memory reorderings and is very difficult to correctly handle in a compiler/JIT.

We are discussing a number of safety guarantees that can't be taken for granted in unsynchronized code. An example is *initialization safety*: if an object isn't made visible outside the constructor until after the constructor terminates, then no code, even unsynchronized code in another thread, can see that object without seeing all of the effects of the constructor for that object. However, it may be difficult or impossible to efficiently implement this safety guarantee on multiprocessors with weak memory models.

More information

There is a mailing list for discussion of a specification of the multithreaded semantics of Java and the issues involved with implementing OO runtime systems on multiprocessors with weak memory models. Information at:

http://www.cs.umd.edu/∼pugh/java/memoryModel

Thanks to the many people who are participating in the discussions, far too numerous to list in the space available.

References

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[Pug99] William Pugh. Fixing the Java Memory Model. In *ACM Java Grande Conference*, June 1999.