ECE459: Programming for Performance Winter 2018

Lecture 8 — Atomics, Compiler Hints, Restrict

Patrick Lam and Jeff Zarnett

Atomics

What if we could find a way to get rid of locks and waiting altogether? That would avoid the lock convoy problem as well as any potential for deadlock, starvation, et cetera. In previous courses, you have learned about test-and-set operations and possibly compare-and-swap and those are atomic operations supported through hardware. They are uninterruptible and therefore will either completely succeed or not run at all. Is there a way that we could use those sorts of indivisible operations? Yes!

Atomics are a lower-overhead alternative to locks as long as you're doing suitable operations. Remember that what we wanted sometimes with locks and mutexes and all that is that operations are indivisible: an update to a variable doesn't get interfered with by another update. Remember the key idea is: an *atomic operation* is indivisible. Other threads see state before or after the operation; nothing in between.

We are only going to talk about atomics with sequential consistency. If you use the default `std::memory_order`, that's what you get. What do I mean by that? Well, in the header file `atomic` (C++11 here) there is an enumeration of memory orders and I am suggesting that using the default is pretty nice, compared to the alternative which may or may not be a Lovecraftian Horror to understand (or prove correctness). If you'd like to know about all the options, take a look at [C++15], but here's a quick summary from [Mil08a] (which is much more concise than the C++ Atomics listing):

Value	Explanation
`memory_order_acquire`	Subsequent loads are not moved before the current load or any preceding loads.
`memory_order_release`	Preceding stores are not moved past the current store or any subsequent stores.
`memory_order_acq_rel`	Combine the acquire and release guarantees
`memory_order_consume`	A potentially weaker form of memory_order_acquire that enforces ordering of the current load before other operations that are data-dependent on it (for instance, when a load of a pointer is marked memory_order_consume, subsequent operations that dereference this pointer won't be moved before it (yes, even that is not guaranteed on all platforms!).
`memory_order_relaxed`	All reordering are okay; only atomicity is required of this operation.
`memory_order_seq_cst`	Same as `memory_order_acq_rel`, plus a single total order exists in which all threads observe all modifications in the same order.

The C++11 standard includes both strong and weak atomics. The weak ones are the ones where you get to specify the the memory ordering of load and store operating in a way that is not sequentially consistent. But we care about the standard, sequentially consistent kind of operation. *Don't* use relaxed atomics unless you're an expert! Basically, a value that is seen from a memory load may come from the past or from the future (it's all relative, of course). If you want to dig into the details about an example, I recommend [Mil08b], which goes into the details of just how difficult it is to prove correctness. If that doesn't talk you out of it, I'm not sure what will.

Atomic Flags. The simplest form of C++11 atomic is the `atomic_flag`. Not surprisingly, this represents a boolean flag. You can clear the flag and test-and-set it.

```
#include <atomic>

atomic_flag f = ATOMIC_FLAG_INIT;
int foo() {
  f.clear();
```

```
  if (f.test_and_set()) {
    // was true
  }
}
```

This returns the previous value. There is no assignment (=) operator for `atomic_flag`s. Although I guess in C++ you could define one if you wanted. This is kind of a dangerous thing about C++. If in C you see a line of code like `z = x + y;` you can have a pretty good idea about what it does and you can infer that there's some sort of natural meaning to the + operator there, like addition or concatenation. In C++, however, this same line of code tells you nothing, unless you know (1) the type of `x`, (2) the type of `y`, and (3) how the `+` operator is defined on those two operands *in that order*. But I'm digressing.

More general C++ atomics. Boolean flags are nice, but we want more. C++11 supports arbitrary types as atomic. Here'a an example declaration:

```
#include <atomic>
atomic<int> x;
```

The C++11 library implements atomics using lock-free operations for small types and using mutexes for large types. The general types of operations that you can do with atomics are three: reads, writes, and RMW (read-modify-write) operations. C++ has syntax to make these all transparent.

```
// atomic reads and writes
#include <atomic>
#include <iostream>

std::atomic<int> ai;
int i;

int main() {
    ai = 4;
    i = ai;
    ai = i;
    std::cout << i;
}
```

If you want, you can also use `i = ai.load()` and `ai.store(i)`.

As for RMW operations, consider `ai++`. This is really

```
tmp = ai.read();
tmp++;
ai.write(tmp);
```

But, hardware can do that atomically. It can also do other RMWs: `+-, &=, etc, compare-and-swap`.

More info on C++11 atomics:
`http://preshing.com/20130618/atomic-vs-non-atomic-operations/`

We talked about C++11 atomics. Is there a pthread equivalent? Nope, not really.

OS X has atomics via OS calls:
`https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/Multithreading/ThreadSafety/ThreadSafety.html`

The Linux kernel provides a number of atomic operations (but that doesn't really make them portable). Reference:
`http://stackoverflow.com/questions/1130018/unix-portable-atomic-operations`

The Compiler and You

Making the compiler work for you is critical to programming for performance. We'll therefore see some compiler implementation details in this class. Understanding these details will help you reason about how your code gets translated into machine code and thus executed.

Three Address Code. Compiler analyses are much easier to perform on simple expressions which have two operands and a result—hence three addresses—rather than full expression trees. Any good compiler will therefore convert a program's abstract syntax tree into an intermediate, portable, three-address code before going to a machine-specific backend.

Each statement represents one fundamental operation; we'll consider these operations to be atomic. A typical statement looks like this:

$$\text{result} := \text{operand}_1 \text{ operator } \text{operand}_2$$

Three-address code is useful for reasoning about data races. It is also easier to read than assembly, as it separates out memory reads and writes.

GIMPLE: `gcc`'s three-address code. To see the GIMPLE representation of your code, pass `gcc` the `-fdump-tree-gimple` flag. You can also see all of the three address code generated by the compiler; use `-fdump-tree-all`. You'll probably just be interested in the optimized version.

I suggest using GIMPLE to reason about your code at a low level without having to read assembly. Let's take a few minutes to look at a few examples, focusing on some code we have already created.

The `restrict` qualifier

The `restrict` qualifier on pointer `p` tells the compiler [Act06] that it may assume that, in the scope of `p`, the program will not use any other pointer `q` to access the data at `*p`.

The `restrict` qualifier is a feature introduced in C99: "The restrict type qualifier allows programs to be written so that translators can produce significantly faster executables."

- To request C99 in `gcc`, use the `-std=c99` flag.

`restrict` means: you are promising the compiler that the pointer will never alias (another pointer will not point to the same data) for the lifetime of the pointer. Hence, two pointers declared `restrict` must never point to the same data.

In fact [Act06] includes a contract that goes with the use of restrict:

> I, [insert your name], a PROFESSIONAL or AMATEUR [circle one] programmer recognize that there are limits to what a compiler can do. I certify that, to the best of my knowledge, there are no magic elves or monkeys in the compiler which through the forces of fairy dust can always make code faster. I understand that there are some problems for which there is not enough information to solve. I hereby declare that given the opportunity to provide the compiler with sufficient information, perhaps through some key word, I will gladly use said keyword and not bitch and moan about how "the compiler should be doing this for me."
>
> In this case, I promise that the pointer declared along with the restrict qualifier is not aliased. I certify that writes through this pointer will not effect the values read through any other pointer available in the same context which is also declared as restricted.
>
> * Your agreement to this contract is implied by use of the restrict keyword ;)

Of course, I highly recommend that you have your personal legal expert review this contract before you sign it. As I would for any contract. Contracts are serious business.

An example from Wikipedia:

```
  void updatePtrs(int* ptrA, int* ptrB, int* val) {
    *ptrA += *val;
    *ptrB += *val;
  }
```

Would declaring all these pointers as `restrict` generate better code?

Well, let's look at the GIMPLE.

```
void updatePtrs(int* ptrA, int* ptrB, int* val) {
 D.1609 = *ptrA;
 D.1610 = *val;
 D.1611 = D.1609 + D.1610;
 *ptrA = D.1611;
 D.1612 = *ptrB;
 D.1610 = *val;
 D.1613 = D.1612 + D.1610;
 *ptrB = D.1613;
}
```

Now we can answer the question: "Could any operation be left out if all the pointers didn't overlap?"

- If `ptrA` and `val` are not equal, you don't have to reload the data on **line 7**.
- Otherwise, you would: there might be a call, somewhere:
  `updatePtrs(&x, &y, &x);`

Hence, this set of annotations allows optimization:

```
    void updatePtrs(int* restrict ptrA,
                    int* restrict ptrB,
                    int* restrict val)
```

Note: you can get the optimization by just declaring `ptrA` and `val` as `restrict`; `ptrB` isn't needed for this optimization

Summary of `restrict`. Use `restrict` whenever you know the pointer will not alias another pointer (also declared `restrict`).

It's hard for the compiler to infer pointer aliasing information; it's easier for you to specify it. If the compiler has this information, it can better optimize your code; in the body of a critical loop, that can result in better performance.

A caveat: don't lie to the compiler, or you will get undefined behaviour.

Aside: `restrict` is not the same as `const`. `const` data can still be changed through an alias.

References

[Act06] Mike Acton. Demystifying the restrict keyword, 2006. Online; accessed 7-December-2015. URL: `http://cellperformance.beyond3d.com/articles/2006/05/demystifying-the-restrict-keyword.html`.

[C++15] C++ Reference. std::memory_order, 2015. Online; accessed 6-December-2015. URL: `http://en.cppreference.com/w/cpp/atomic/memory\_order`.

[Mil08a] Bartosz Milewski. C++ atomics and memory ordering, 2008. Online; accessed 6-December-2015. URL: http://bartoszmilewski.com/2008/12/01/c-atomics-and-memory-ordering/.

[Mil08b] Bartosz Milewski. The inscrutable c++ memory model, 2008. Online; accessed 6-December-2015. URL: http://bartoszmilewski.com/2008/12/23/the-inscrutable-c-memory-model/.