ECE459: Programming for Performance Winter 2015

Lecture 10 — January 26, 2015

Patrick Lam *version 1*

C++ atomics

We've talked about locks. Atomics are a lower-overhead alternative to locks as long as you're doing suitable operations.

We are only going to talk about atomics with sequential consistency. If you use the default `std::memory_order`, that's what you get.

Don't use relaxed atomics unless you're an expert![1]

Key idea. An *atomic operation* is indivisible. Other threads see state before or after the operation, nothing in between.

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
  }
}
```

`test_and_set` atomically sets the flag to true, and returns the previous value. There is no assignment (=) operator for `atomic_flag`s.

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.

[1] `http://stackoverflow.com/questions/9553591/c-stdatomic-what-is-stdmemory-order-and-how-to-use-them`

```
// 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/`

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.

Branch Prediction

We've mentioned before that modern CPUs must rely on branch prediction to get the performance we're used to. We also had a live coding demo which demonstrated the impact of mis-prediction. Here's a bit more information about providing the compiler with branch prediction hints.

The right thing to do most of the time is to not call it. Use profile-guided optimization whenever possible if you do want to use branch prediction. Nevertheless, providing branch prediction hints can be useful for error cases on slow processors.

`gcc` provides a builtin function:

```
long __builtin_expect (long exp, long c)
```

When you call it, you are indicating that the expected result is that `exp` equals `c`. In response, the compiler will pass the prediction on to the CPU and reorder the code properly to take advantage of the (hopefully correct) hint.

The `restrict` qualifier

The `restrict` qualifier on pointer `p` tells the compiler[2] 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.

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.

```
1  void updatePtrs(int* ptrA, int* ptrB, int* val) {
2   D.1609 = *ptrA;
3   D.1610 = *val;
4   D.1611 = D.1609 + D.1610;
5   *ptrA = D.1611;
6   D.1612 = *ptrB;
7   D.1610 = *val;
8   D.1613 = D.1612 + D.1610;
9   *ptrB = D.1613;
10 }
```

[2] `http://cellperformance.beyond3d.com/articles/2006/05/demystifying-the-restrict-keyword.html`

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.