ECE459: Programming for Performance Winter 2018

Lecture 9 — C++ Atomics, Compiler Hints, Restrict

Patrick Lam and Jeff Zarnett

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 an example.

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`.