ECE459: Programming for Performance Winter 2014

Lecture 7 — January 28, 2013

Patrick Lam *version 2*

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.

The `volatile` qualifier

We'll continue by discussing C language features and how they affect the compiler. The `volatile` qualifier notifies the compiler that a variable may be changed by "external forces". It therefore ensures that the compiled code does an actual read from a variable every time a read appears (i.e. the compiler can't optimize away the read). It does not prevent re-ordering nor does it protect against races.

Here's an example.

```
int i = 0;

while (i != 255) { ... }
```

`volatile` prevents this from being optimized to:

```
int i = 0;

while (true) { ... }
```

Note that the variable will not actually be `volatile` in the critical section; most of the time, it only prevents useful optimizations. `volatile` is usually wrong unless there is a *very* good reason for it.

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[1] 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;
}
```

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

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 }
```

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.

Dependencies

I've said that some computations appear to be "inherently sequential". Here's why.

Main Idea. A *dependency* prevents parallelization when the computation XY produces a different result from the computation YX.

Loop- and Memory-Carried Dependencies. We distinguish between *loop-carried* and *memory-carried* dependencies. In a loop-carried dependency, an iteration depends on the result of the previous iteration. For instance, consider this code to compute whether a complex number $x_0 + iy_0$ belongs to the Mandelbrot set.

```
// Repeatedly square input, return number of iterations before
// absolute value exceeds 4, or 1000, whichever is smaller.
int inMandelbrot(double x0, double y0) {
  int iterations = 0;
  double x = x0, y = y0, x2 = x*x, y2 = y*y;
  while ((x2+y2 < 4) && (iterations < 1000)) {
    y = 2*x*y + y0;
    x = x2 - y2 + x0;
    x2 = x*x; y2 = y*y;
    iterations++;
  }
  return iterations;
}
```

In this case, it's impossible to parallelize loop iterations, because each iteration *depends* on the (x, y) values calculated in the previous iteration. For any particular $x_0 + iy_0$, you have to run the loop iterations sequentially.

Note that you can parallelize the Mandelbrot set calculation by computing the result simultaneously over many points at once. Indeed, that is a classic "embarassingly parallel" problem, because the you can compute the result for all of the points simultaneously, with no need to communicate.

On the other hand, a memory-carried dependency is one where the result of a computation *depends* on the order in which two memory accesses occur. For instance:

```
int val = 0;

void g() { val = 1; }
void h() { val = val + 2; }
```

What are the possible outcomes after executing `g()` and `h()` in parallel threads?

RAW, WAR, WAW and RAR

The most obvious case of a dependency is as follows:

```
int y = f(x);
int z = g(y);
```

This is a read-after-write (RAW), or "true" dependency: the first statement writes `y` and the second statement reads it. Other types of dependencies are:

	Read 2nd	**Write 2nd**
Read 1st	Read after read (RAR)	Write after read (WAR)
	No dependency	Antidependency
Write 1st	Read after write (RAW)	Write after write (WAW)
	True dependency	Output dependency

The no-dependency case (RAR) is clear. Declaring data immutable in your program is a good way to ensure no dependencies.

Let's look at an antidependency (WAR) example.

```
void antiDependency(int z) {
  int y = f(x);
  x = z + 1;
}
```

```
void fixedAntiDependency(int z) {
  int x_copy = x;
  int y = f(x_copy);
  x = z + 1;
}
```

Why is there a problem?

Finally, WAWs can also inhibit parallelization:

```
void outputDependency(int x, int z) {
  y = x + 1;
  y = z + 1;
}
```

```
void fixedOutputDependency(int x, int z) {
  y_copy = x + 1;
  y = z + 1;
}
```

In both of these cases, renaming or copying data can eliminate the dependence and enable parallelization. Of course, copying data also takes time and uses cache, so it's not free. One might change the output locations of both statements and then copy in the correct output.