ECE459: Programming for Performance Winter 2015

Lecture 29 — March 18, 2015

Patrick Lam *version 0*

Compiler Optimizations

What does it really mean when you say `-O2`? We'll see some representative compiler optimizations and discuss how they can improve program performance. Because we're talking about Programming for Performance, I'll point out cases that stop compilers from being able to optimize your code. In general, it's better if the compiler automatically does a performance-improving transformation rather than you doing it manually; it's probably a waste of time for you and it also makes your code less readable.

There are a lot of pages on the Internet with information about optimizations. Here's one that contains good examples:

`http://www.digitalmars.com/ctg/ctgOptimizer.html`

You can find a full list of `gcc` options here:

`http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

About Compiler Optimizations. First of all, "optimization" is a bit of a misnomer, since compilers generally do not generate "optimal" code. They just generate *better* code.

Often, what happens is that the program you literally wrote is too slow. The contract of the compiler (working with the architecture) is to actually execute a program with the same behaviour as yours, but which runs faster.

`gcc` optimization levels. Here's what `-On` means for `gcc`. Other compilers have similar (but not identical) optimization flags.

- `-O0` (default): Fastest compilation time. Debugging works as expected.
- `-O1` (`-O`): Reduce code size and execution time. No optimizations that increase compiliation time.
- `-O2`: All optimizations except space vs. speed tradeoffs.
- `-O3`: All optimizations.
- `-Ofast`: All `-O3` optimizations, plus non-standards compliant optimizations, particularly `-ffast-math`. (Like `-fast` on the Solaris compiler.)

  This flag turns off exact implementations of IEEE or ISO rules/specifications for math functions. Generally, if you don't care about the exact result, you can use this for a speedup.

Scalar Optimizations

By scalar optimizations, I mean optimizations which affect scalar (non-array) operations. Here are some examples of scalar optimizations.

Constant folding. Probably the simplest optimization one can think of. Tag line: "Why do later something you can do now?" We simply translate:

$$\texttt{i = 1024 * 1024} \quad \Longrightarrow \quad \texttt{i = 1048576}$$

Enabled at all optimization levels. The compiler will not emit code that does the multiplication at runtime. It will simply use the computed value.

Common subexpression elimination. We can do common subexpression elimination when the same expression `x op y` is computed more than once, and neither `x` nor `y` change between the two computations. In the below example, we need to compute `c + d` only once.

```
a = c + d * y;
b = c + d * z;

w = 3;
x = f();
y = x;
z = w + y;
```

Enabled at `-O2`*,* `-O3` *or with* `-fgcse`. Note that these flags actually enable a global CSE pass, where global means across-basic-blocks. This also enables global constant and copy propagation.

Constant propagation. Moves constant values from definition to use. The transformation is valid if there are no redefinitions of the variable between the definition and its use. In the above example, we can propagate the constant value 3 to its use in `z = w + y`, yielding `z = 3 + y`.

Copy propagation. A bit more sophisticated than constant propagation—telescopes copies of variables from their definition to their use. This usually runs after CSE. Using it, we can replace the last statement with `z = w + x`. If we run both constant and copy propagation together, we get `z = 3 + x`.

These scalar optimizations are more complicated in the presence of pointers, e.g. `z = *w + y`. More below.

Scalar Replacement of Aggregates. Censored. Too many people misunderstood it last year.

Redundant Code Optimizations. In some sense, most optimizations remove redundant code, but one particular optimization is *dead code elimination*, which removes code that is guaranteed to not execute. For instance:

```
int f(int x) {
  return x * 2;
}

int g() {
  if (f(5) % 2 == 0) {
    // do stuff...
  } else {
    // do other stuff
  }
}
```

By looking at the code, you can tell that the then-branch of the `if` statement in `g()` is always going to execute, and the else-branch is never going to execute.

The general problem, as with many other compiler problems, is undecidable.

Loop Optimizations

Loop optimizations are particularly profitable when loops execute often. This is often a win, because programs spend a lot of time looping. The trick is to find which loops are going to be the important ones. Profiling is helpful there.

A loop induction variable is a variable that varies on each iteration of the loop; the loop variable is definitely a loop induction variable, but there may be others. *Induction variable elimination* gets rid of extra induction variables.

Scalar replacement replaces an array read `a[i]` occuring multiple times with a single read `temp = a[i]` and references to `temp` otherwise. It needs to know that `a[i]` won't change between reads.

Some languages include array bounds checks, and loop optimizations can eliminate array bounds checks if they can prove that the loop never iterates past the array bounds.

Loop unrolling. This optimization lets the processor run more code without having to branch as often. *Software pipelining* is a synergistic optimization, which allows multiple iterations of a loop to proceed in parallel. This optimization is also useful for SIMD. Here's an example.

```
for (int i = 0; i < 4; ++i)
    f(i)
```

could be transformed to:

```
f(0)
f(1)
f(2)
f(3)
```

Enabled with `-funroll-loops`.

Loop interchange. This optimization can give big wins for caches (which are key); it changes the nesting of loops to coincide with the ordering of array elements in memory. For instance, in C, we could change this:

```
for (int i = 0; i < N; ++i)
    for (int j = 0; j < M; ++j)
        a[j][i] = a[j][i] * c
```

to this:

```
for (int j = 0; j < M; ++j)
    for (int i = 0; i < N; ++i)
        a[j][i] = a[j][i] * c
```

since C is *row-major* (meaning a[1][1] is beside a[1][2]), rather than *column-major*.

Enabled with `-floop-interchange`.

Loop fusion. This optimization is like the OpenMP collapse construct; we transform

```
for (int i = 0; i < 100; ++i)
    a[i] = 4

for (int i = 0; i < 100; ++i)
    b[i] = 7
```

into this:

```
for (int i = 0; i < 100; ++i) {
    a[i] = 4
    b[i] = 7
}
```

There's a trade-off between data locality and loop overhead; hence, sometimes the inverse transformation, *loop fission*, will improve performance.

Loop-invariant code motion. Also known as *Loop hoisting*, this optimization moves calculations out of a loop. For instance,

```
for (int i = 0; i < 100; ++i) {
    s = x * y;
    a[i] = s * i;
}
```

would be transformed to this:

```
s = x * y;
for (int i = 0; i < 100; ++i) {
    a[i] = s * i;
}
```

This reduces the amount of work we have to do for each iteration of the loop.

Alias and Pointer Analysis

As we've seen in the above analyses, compiler optimizations often need to know about what parts of memory each statement reads to. This is easy when talking about scalar variables which are stored on the stack. This is much harder when talking about pointers or arrays (which can alias). *Alias analysis* helps by declaring that a given variable `p` does not alias another variable `q`; that is, they point to different heap locations. *Pointer analysis* abstractly tracks what regions of the heap each variable points to. A region of the heap may be the memory allocated at a particular program point.

When we know that two pointers don't alias, then we know that their effects are independent, so it's correct to move things around. This also helps in reasoning about side effects and enabling reordering.

We've talked about automatic parallelization previously in this course. At this point, I'll remind you that we used `restrict` so that the compiler wouldn't have to do as much pointer analysis. Shape analysis builds on pointer analysis to determine that data structures are indeed trees rather than lists.

Call Graphs. Many interprocedural analyses require accurate call graphs. A call graph is a directed graph showing relationships between functions. It's easy to compute a call graph when you have C-style function calls. It's much harder when you have virtual methods, as in C++ or Java, or even C function pointers. In particular, you need pointer analysis information to construct the call graph.

Devirtualization. This optimization attempts to convert virtual function calls to direct calls. Virtual method calls have the potential to be slow, because there is effectively a branch to predict. If the branch prediction goes well, then it doesn't impose more runtime cost. However, the branch prediction might go poorly. (In general for C++, the program must read the object's vtable. Plus, virtual calls impede other optimizations. Compilers can help by doing sophisticated analyses to compute the call graph and by replacing virtual method calls with nonvirtual method calls. Consider the following code:

```
class A {
    virtual void m();
};

class B : public A {
    virtual void m();
}

int main(int argc, char *argv[]) {

    std::unique_ptr<A> t(new B);
    t.m();
}
```

Devirtualization could eliminate vtable access; instead, we could just call B's `m` method directly. By the way, "Rapid Type Analysis" analyzes the entire program, observes that only `B` objects are ever instantiated, and enables devirtualization of the `b.m()` call.

Enabled with `-O2`*,* `-O3`*, or with* `-fdevirtualize`*.*

Obviously, inlining and devirtualization require call graphs. But so does any analysis that needs to know about the heap effects of functions that get called; for instance, consider this code:

```
int n;

int f() { /* opaque */ }

int main() {
  n = 5;
  f();
  printf("%d\n", n);
}
```

We could propagate the constant value 5, as long as we know that `f()` does not write to `n`.

Tail Recursion Elimination. This optimization is mandatory in some functional languages; we replace a call by a `goto` at the compiler level. Consider this example, courtesy of Wikipedia:

```
int bar(int N) {
  if (A(N))
    return B(N);
  else
    return bar(N);
}
```

For both calls, to `B` and `bar`, we don't need to return control to the calling `bar()` before returning to its caller. This avoids function call overhead and reduces call stack use.

Enabled with `-foptimize-sibling-calls`*.* Also supports sibling calls as well as tail-recursive calls.

Miscellaneous Low-Level Optimizations

Some optimizations affect low level code generation; here are two examples.

Branch Prediction. `gcc` attempts to guess the probability of each branch to best order the code. (For an `if`, fall-through is most efficient. Why?)

This isn't quite an optimization, but you can use `__builtin_expect(expr, value)` to help GCC, if you know the run-time characteristics of your program. An example, from the Linux kernel:

```
#define likely(x)       __builtin_expect((x),1)
#define unlikely(x)     __builtin_expect((x),0)
```

Architecture-Specific. `gcc` can also generate code tuned to particular processors and processor variants. You can specify this using `-march` and `-mtune`. (`-march` implies `-mtune`). This will enable specific instructions that not all CPUs support (e.g. SSE4.2). For example, `-march=corei7`.

Good to use on your local machine, not ideal for shipped code.

Profile-guided optimization

The optimizations we've discussed so far have been purely-static, or compile-time, optimizations. However, we already know about two cases where the compiler really needs to understand the *run-time* behaviour of the software: 1) deciding whether or not to inline; and 2) helping the processor out with branch prediction. *Dynamic* information collected at run-time (or from a previous run-time) gives better answers to these questions.

How to use profile-guided optimization. The basic workflow is like this:

- Run the compiler, telling it to produce an instrumented binary to collect profiles.
- Run the instrumented binary on a representative test suite.
- Run the compiler again, but tell it about the profiles you've collected.

We can see that collecting profile data complicates the profiling process. However, it can produce code that's faster on typical workloads, as long as the test suite is representative. (How much faster? Maybe 5-25% faster, according to Microsoft.)

Note that just-in-time compilers (e.g. Java virtual machines) can automatically do profile-guided optimizations, since they are compiling code on-the-fly and can collect performance measurements tuned for a particular run. Solaris, Microsoft VC++, and GNU gcc all support profile-guided optimization to some extent these days.

Optimizations that PGO enables. So, what do we do with this profile data?

- **Inlining.** One of the easiest and most profitable applications of this data is making inlining decisions. Inlining rarely-used code is going to hinder performance, while inlining often-executed code is going to improve performance and enable further optimizations. Everyone does this.
- **Improving Cache Locality.** Profile-guided optimization can help with cache locality by placing commonly-called functions next to each other, as well as often-executed basic blocks. This intersects with branch prediction, described below; however, it also applies to `switch` statements.
- **Branch Prediction/Virtual Call Prediction.** Architectures often assume that forward branches are not taken while backwards branches are taken. We can make these assumptions true more often using profile-guided optimization, and for a forward branch, we can make sure that the common case is the fall-through case.

  On the topic of prediction, we can inline the most common case for a virtual method call, guarded by a conditional.