ECE459: Programming for Performance Winter 2011

Lecture 17 — March 10, 2011

Patrick Lam *version 1*

Some clarifications from last time: 1) I mentioned the existence of *load fences*, but not why they were useful. They prevent loads from being delayed beyond the fence; for instance, a load might need to finish before the thread releases the lock. 2) There is no branch prediction overhead for a return, due to the use of a Return Address Stack.

Today we'll see some representative compiler optimizations and how they can improve programs. Because we're talking about Programming for Performance, I'll point out cases that stop compilers from being able to optimize your code.

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`

Scalar Optimizations. By scalar optimizations, I mean optimizations which affect scalar (non-array) operations. We'll see two examples of scalar optimizations.

I mentioned *common subexpression elimination* last time. Let's look at it in a bit more detail. 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;
```

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 the statement `z = 3 + y`.

Copy propagation is a bit more sophisticated and telescopes copies of variables from their definition to their use. So 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`.

Redundant Code Optimizations. In some sense, all 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.

Loop interchange is an optimization that helps with caches; it changes the nesting of loops to coincide with the ordering of array element in memory. *Loop fusion* is like the OpenMP collapse construct. *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. *Loop hoisting* or *loop-invariant code motion* moves calculations out of a loop, as I discussed last time.

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.

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.

You have seen manual *loop unrolling* in the meschach code. The idea is to let 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.

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.

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; and, as we saw in Assignment 3, we can better parallelize trees than lists. 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. 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 function pointers. In particular, you need pointer analysis information to construct the call graph.

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`.

Devirtualization. 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. 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 {
  void m() { ... }
}

class B extends A {
  void m() { ... }
}

class Main {
  public static void main(String[] argv) {
    A b = new B();
    b.m();
  }
}
```

"Rapid Type Analysis" analyzes the entire program, observes that only `B` objects are ever instantiated, and enables devirtualization of the `b.m()` call.

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.

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.

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 validate these assumptions 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.