ECE459: Programming for Performance Winter 2018

Lecture 20 — Compiler Optimizations

Patrick Lam

Compiler Optimizations: Interprocedural Analysis and Link-Time Optimizations

"Are economies of scale real?"

In this context, does a whole-program optimization really improve your program? We'll start by first talking about some information that is critical for whole-program optimizations.

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.

Inlining. We talked about inlining in Lecture 18. Compilers can inline following compiler directives, but usually more based on heuristics. Devirtualization enables more inlining. The compiler always inlines functions marked with the always_inline attribute, as seen in passing in Lecture 3.

Enabled with -O2 *and* -O3.

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 (because bar() is done anyway). 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.

Link-Time Optimizations

Next up: mechanics of interprocedural optimizations in modern open-source compilers. Conceptually, interprocedural optimizations have been well-understood for a while. But practical implementations in open-source compilers are still relatively new; Hubička [?] summarizes recent history. In 2004, the only real interprocedural optimization in gcc was inlining, and it was quite ad-hoc.

The biggest challenge for interprocedural optimizations is scalability, so it fits right in as a topic of discussion for this course. Here's an outline of how it works:

- local generation (parallelizable): compile to Intermediate Representation. Must generate compact IR for whole-program analysis phase.
- whole-program analysis (hard to parallelize!): create call graph, make transformation decisions. Possibly partition the program.
- local transformations (parallelizable): carry out transformations to local IRs, generate object code. Perhaps use call graph partitions to decide optimizations.

There were a number of conceptually-uninteresting implementation challenges to be overcome before gcc could have its intermediate code available for interprocedural analysis (i.e. there was no stable on-disk IR format). The transformations look like this:

- global decisions, local transformations:
  - devirtualization
  - dead variable elimination/dead function elimination
  - field reordering, struct splitting/reorganization
- global decisions, global transformations:
  - cross-module inlining
  - virtual function inlining
  - interprocedural constant propagation

The interesting issues arise from making the whole-program analysis scalable. Firefox, the Linux kernel, and Chromium contain tens of millions of lines of code. Whole-program analysis requires that all of this code (in IR) be available to the analysis and that at least some summary of the code be in memory, along with the call graph. (Since it's a whole-program analysis, any part of the program may affect other parts). The first problem is getting it into memory; loading the IR for tens of millions of lines of code is a non-starter. Clearly, anything that is more expensive than linear time can cause problems. Partitioning the program can help.

How did gcc get better? Hubička [**?**] explains how. In line with what I've said earlier, it's avoiding unnecessary work.

- gcc 4.5: initial version of LTO;
- gcc 4.6: parallelization; partitioning of the call graph (put closely-related functions together, approximate functions in other partitions); the bottleneck: streaming in types and declarations;
- gcc 4.7–4.9: improve build times, memory usage ["chasing unnecessary data away".]

As far as I can tell, today's gcc, with `-flto`, does work and includes optimizations including constant propagation and function specialization.

Impact. gcc CTO appears to give 3–5% improvements in performance, which compiler experts consider good. Like we discussed last time, this allows developers to shift their attention from manual factoring of translation units to letting the compiler do it. (This is kind of like going from manual transmissions to automatic transmissions for cars. . .).

The LLVM project provides more details at [**?**], while gcc details can be found at [**?**].