

# Lecture 20 — Compiler Optimizations

Patrick Lam

`patrick.lam@uwaterloo.ca`

Department of Electrical and Computer Engineering  
University of Waterloo

March 12, 2018

# 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.

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  $q$ .

**Pointer analysis** tracks what regions of the heap each variable points to.

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 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.

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.

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.

(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.

---

```
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.

‘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.*



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.

*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.

---

```
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`.

This optimization is mandatory in some functional languages; we replace a call by a goto at the compiler level.

---

```
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.

Biggest challenge for interprocedural optimizations = scalability, so it fits right in as a topic of discussion for this course.

An outline of interprocedural optimization:

- local generation (parallelizable)
- whole-program analysis (hard to parallelize!)
- local transformations (parallelizable)

## Transformations:

- 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.

Language	files	comment	code
[Chromium] C++	44991	1405276	11102212
[Firefox] C++	10973	605158	4467943
[Linux] C	26238	2573558	13092340

Whole-program analysis requires that all of this code (in IR) be available to the analysis & some summary of the code be in memory, along with the call graph.

Whole-program analysis  $\Rightarrow$  any part of the program may affect other parts.

First problem: 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? 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”.]

Today's gcc, with `-flto`, does work and includes optimizations including constant propagation and function specialization.



gcc LTO gives  $\sim 3\text{--}5\%$  perf improvements  
(good per compiler experts).

Developers can shift attention from  
manual factoring to letting compiler do it.