Lecture 18 — Compiler Optimizations

Patrick Lam
patrick.lam@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

June 1, 2025

Lunch Time, Already?

“Is there any such thing as a free lunch?”

Compiler optimizations really do feel like a free lunch.

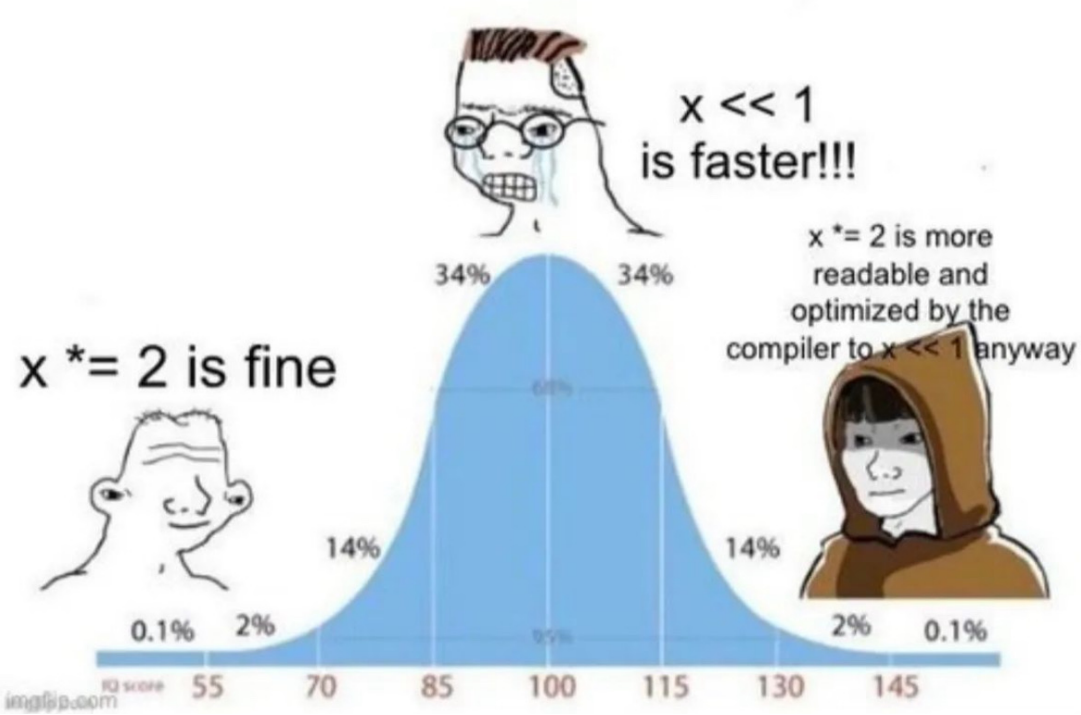
But what `do-O` or `-C opt-level=3` really mean?

We’ll see some representative compiler optimizations and discuss how they can improve program performance.

You Do Your Job...

I'll point out cases that stop compilers from being able to optimize your code.

You Do Your Job...



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.

Flame On!

When you want fast binaries, you want to disable debug information and enable compiler optimization.

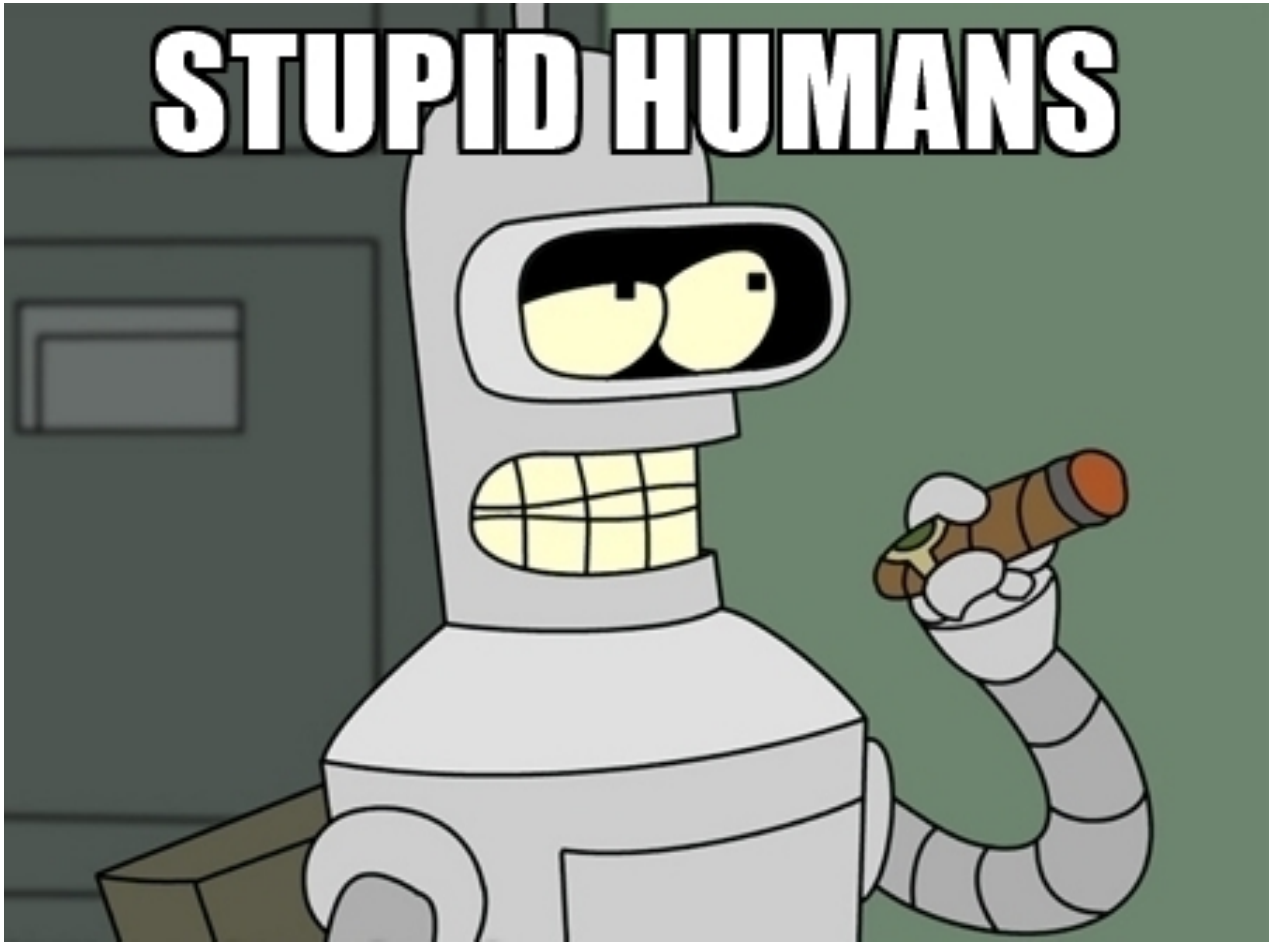
You also want link-time optimization (described below) by adding to your `Cargo.toml`:

```
[profile.release]
lto = true
```

About Compiler Optimizations

First of all, “optimization” is a bit of a misnomer.

Compilers generally don’t generate “optimal” code. They generate *better* code.



Optimization Levels

`rustc` to confirm that apart from some vectorization, most of Rust's optimization takes place at the backend LLVM level.

The `-C opt-level` option mostly sets inline limits and passes the requested optimization level to the backend:

- 0: no optimizations, also turns on cfg(debug_assertions).
- 1: basic optimizations
- 2: some optimizations
- 3: all optimizations
- "s": optimize for binary size
- "z": optimize for binary size, but also turn off loop vectorization.

Understand LLVM

Since Rust leverages LLVM optimizations, it's good to understand those. Many pages on the Internet describe optimizations.

Here's one that contains good examples for C/C++; I've translated appropriate cases to Rust in this lecture.

Scalar Optimizations: Constant Folding

Tag line: “Why do later something you can do now?”

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

Enabled always.

The compiler will not emit code that does the multiplication at runtime.

Common Subexpression Elimination

We can do common subexpression elimination when the same expression `x op y` is computed more than once.

Neither x nor y may change between the two computations.

```
pub fn add(c:i32, d: i32, y:i32, z:i32) -> (i32, i32, i32) {
    let a = (c + d) * y;
    let b = (c + d) * z;
    let w = 3; let x = f(); let y = x;
    let z = w + y;
    return (a, b, z);
}

pub fn f() -> i32 { return 5; }
```

Enabled at level 1.

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.

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`.

Redundant Code Optimizations

Dead code elimination: removes code that is guaranteed to not execute.


```
pub fn f(x:i32) ->
      i32 {
  return x * 2;
}
```

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

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.

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

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.

Rust has array bounds checks, but they can be eliminated if using an iterator.

You would usually iterate on an `IntoIterator`.

Loop Unrolling

This 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 i in &[1,2,3,4]
        {
    f(*i);
}
```

$\Longrightarrow$

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

Loop Interchange

Changes the nesting of loops to coincide with the ordering of array elements in memory.

```
pub fn mul(a: &mut [[i32; 8];
      4], c: i32) {
   for i in 0..8 {
      for j in 0..4 {
        a[j][i] = a[j][i] * c;
      }
   }
}
```

$\Longrightarrow$

```
pub fn mul(a: &mut [[i32; 8];
      4], c: i32) {
   for j in 0..4 {
     for i in 0..8 {
       a[j][i] = a[j][i] * c;
     }
   }
}
```

Rust is row-major but swapping the order may affect performance on some CPUs.

Loop Fusion

This optimization is like the OpenMP collapse construct; we transform

```
for i in 0..100 {
    a[i] = 4;
}

for i in 0..100 {
    b[i] = 7;
}
```

$\Longrightarrow$

```
for i in 0..100 {
    a[i] = 4;
    b[i] = 7;
}
```

There's a trade-off between data locality and loop overhead.

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 i in 0..100 {
    s = x * y;
    a[i] = s * i;
}
```

```
s = x * y;
for i in 0..100 {
    a[i] = s * i;
}
```

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

Miscellaneous Low-Level Optimizations

I used to talk about likely/unlikely branch prediction hints, but Rust seems not keen to expose this.

Rust does expose the `#[cold]` attribute, which you can use to mark a method as unlikely to be called (e.g. `panic`).

Architecture-Specific

LLVM can also generate code tuned to particular processors.

You can specify this using `-C target-cpu` and `-C target-feature`.

This will enable specific instructions that not all CPUs support (e.g. SSE4.2).

`native` is a good target CPU if you run and compile on the same machine.

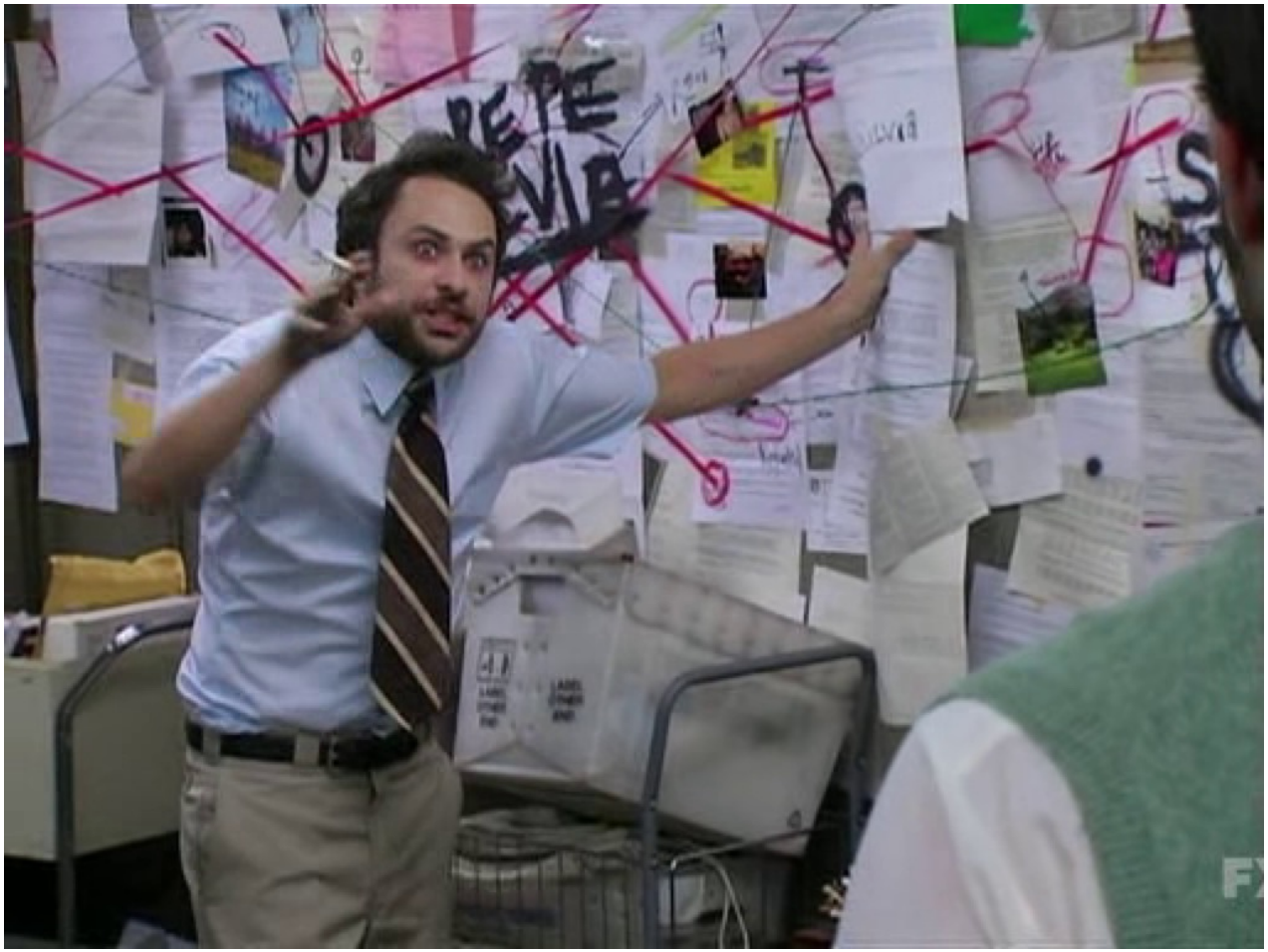
Good to use on your local machine or your cloud servers, not ideal for code you ship to others.

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.



It’s all connected!

Alias and Pointer Analysis

Compiler optimizations often need to know about what parts of memory each statement reads.

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).

This is much harder in conventional languages when talking about pointers or arrays, which can alias.

The whole borrowing thing primarily controls aliasing.

Alias Analysis

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.

Controlled aliasing makes it easier to reason about than in other languages.

Shape analysis builds on pointer analysis to determine that data structures are indeed trees rather than lists.

Call Graph Analysis

Many interprocedural analyses require accurate call graphs.

Call Graph

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.

(In general for Rust and C++, the program must read the object's vtable.)

Plus, virtual calls impede other optimizations.

Devirtualization

Compilers can help by doing sophisticated analyses to compute the call graph and by replacing virtual method calls with nonvirtual method calls.

```
fn flag() -> bool { true }

fn main() {
    let mut to: &dyn Foo = &Bar;
    if flag() { to = &Baz; }
    to.foo();
}

trait Foo { fn foo(&self) -> i32; }

struct Bar;
impl Foo for Bar {
    fn foo(&self) -> i32 { println!("bar"); 0 }
}

struct Baz;
impl Foo for Baz {
    fn foo(&self) -> i32 { println!("baz"); 1 }
}
```

Devirtualization could eliminate vtable access; instead, we could just call `Baz.foo()` directly.

Devirtualization

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

Inlining

We have seen the notion of inlining:

- Instructs the compiler to just insert the function code in-place, instead of calling the function.
- Hence, no function call overhead!
- Compilers can also do better—context-sensitive—operations they couldn't have done before.

Inline Commands

In Rust, you can tell the compiler to inline a function using an annotation:

- #[inline] hints the compiler to perform an inline expansion.
- #[inline(always)] asks the compiler to always perform an inline expansion.
- #[inline(never)] asks the compiler to never perform an inline expansion.

No overhead… sounds like better performance…
let's inline everything!

The Other Side of Inlining

One big downside:

- Your program size is going to increase.

This is worse than you think:

- Fewer cache hits.
- More trips to memory.

Some inlines can grow very rapidly (C++ extended constructors).

Just from this your performance may go down easily.

Compilers on Inlining

Inlining is merely a suggestion to compilers.
They may ignore you.

For example:

- taking the address of an “inline” function and using it; or
- virtual functions (in C++),

will get you ignored quite fast.

From a Usability Point-of-View

Debugging is more difficult (e.g. you can't set a breakpoint in a function that doesn't actually exist).

- Most compilers simply won't inline code with debugging symbols on.
- Some do, but typically it's more of a pain.

Library design:

- If you change any inline function in your library, any users of that library have to **recompile** their program if the library updates.
  (non-binary-compatible change!)

Not a problem for non-inlined functions—programs execute the new function dynamically at runtime.

Inlining

Compilers can inline following compiler directives, but usually more based on heuristics.

Devirtualization enables more inlining.

Devirtualization and Inlining

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.

```
static mut N:i32 = 5;

fn f() { }

fn main() {
  unsafe {
    N = 2;
    f();
    println!("{}", N);
  }
}
```

We could propagate the constant value 2 to the print statement, as long as we know that `f()` does not write to `N`.

But idiomatic Rust helps us here.

Tail Recursion Elimination

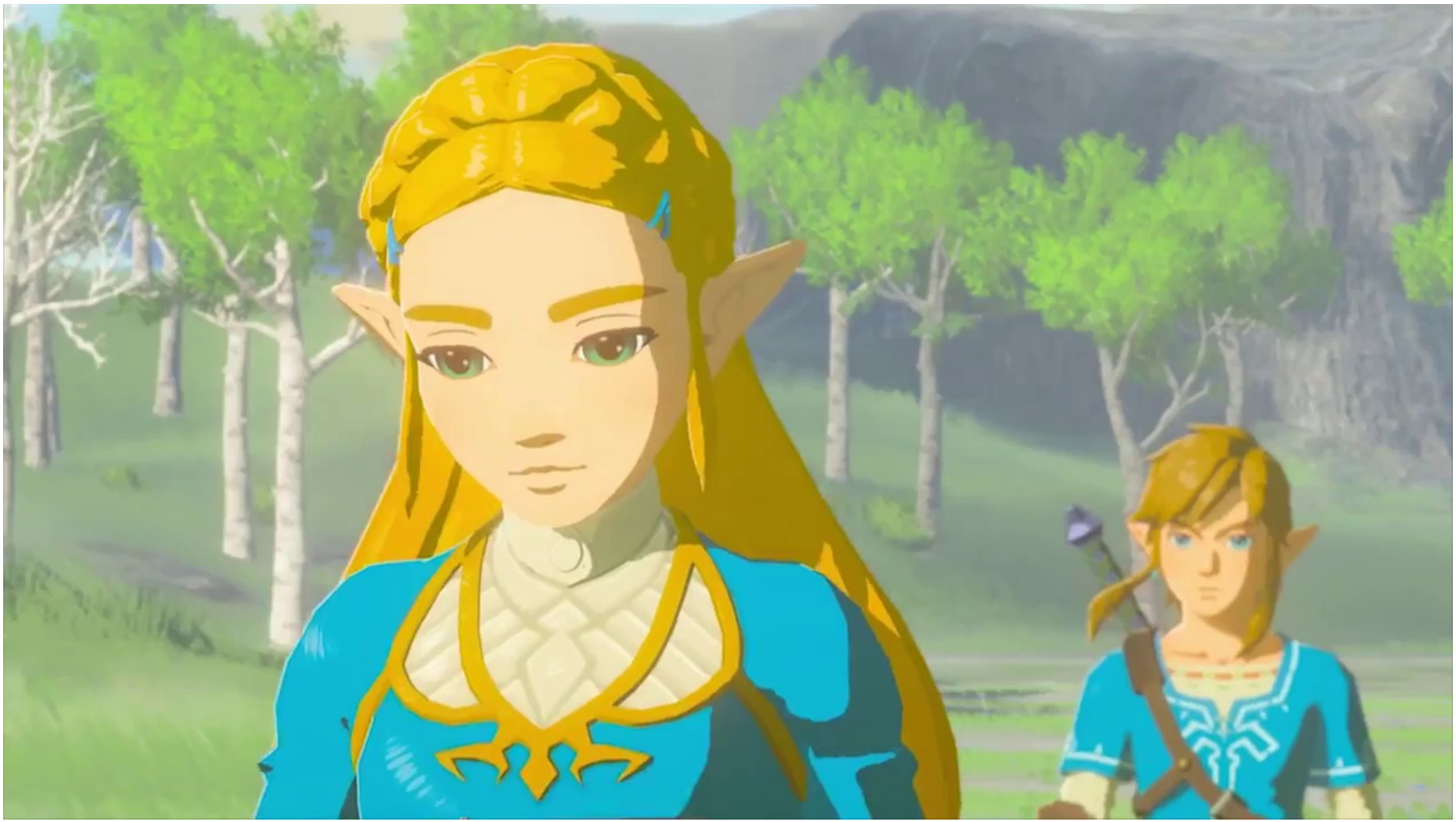
This optimization is mandatory in some functional languages; we replace a call by a `goto` at the compiler level.

```
pub fn fibonacci(n: u64) -> u64 {
    fn fibonacci_lr(n: u64, a: u64, b: u64) -> u64 {
        match n {
            0 => a,
            _ => fibonacci_lr(n - 1, a + b, a),
        }
    }
    fibonacci_lr(n, 1, 0)
}
```

Here, `fibonacci_lr` doesn't need to return control to its caller (because the recursive call is in tail position.

This avoids function call overhead and reduces call stack use.

Link-Time Optimization



“I’ve spent a hundred years trying to hold back Calamity Ganon and you’ve spent your time doing WHAT?”

Link-Time Optimization

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 in Code

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

Whole Program Analysis

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

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.

LLVM LTO can, however, optimize across source languages.

It Works!

gcc LTO gives $\sim$ 3–5% perf improvements (good per compiler experts).

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