ECE459: Programming for Performance Winter 2024

Lecture 18 — Compiler Optimizations

Patrick Lam *2026-02-11*

Compiler Optimizations

"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. 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. Rust lets you force the compiler to do certain optimizations (inlining) even if it might otherwise think it's a bad idea, which is a good compromise when it works.

Enabling compiler optimization. When you want fast binaries, you want to disable debug information and enable compiler optimization. Specify `cargo --release`. 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, 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. The contract of the compiler does not include any obligations if there is any undefined behaviour.

I looked at `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. Here's what the optimization levels mean:

- 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.

Reference material. 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.

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

If you happen to be working with C/C++ in the future, you can find a full list of `gcc` options here:

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

Scalar Optimizations

General note: we can use `https://godbolt.org/` to investigate what the compiler does. It will be easier to understand if you specify `-C overflow-checks=n`.

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

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 always. 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.

```
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. 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`.

In C, these scalar optimizations are more complicated in the presence of pointers, e.g. `z = *w + y`. Fortunately, we're not talking about C here. Unfortunately, probably the LLVM backend that does these optimizations does not know about the guarantees provided by uniqueness.

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:

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

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

We see that the then-branch 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. Let's not get too caught up in the

semantics of the *Entscheidungsproblem*, even if you do speak German and like to show it off by pronouncing that word correctly.

Loop Optimizations

Loop optimizations are often a win, because programs spend a lot of time looping. They are particularly profitable for loops with high iteration counts. The trick is to find which loops those are. Profiling is helpful.

A loop induction variable is a variable that varies on each iteration of the loop; a `for` loop variable is definitely a loop induction variable, but there may be others, which may be functions computable from a primary induction variable. *Induction variable elimination* finds and eliminates (of course!) extra induction variables.

Scalar replacement replaces an array read `a[i]` occurring 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.

Sane 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. This doesn't come up in idiomatic Rust because you would usually iterate on an `IntoIterator`. Language design for the win.

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. Rust does this. Here's an example.

```
for i in &[1,2,3,4] {
  f(*i);                    ⟹    f(0); f(1); f(2); f(3);
}
```

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. Although Rust supports 2-dimensional arrays, it looks like it can be idiomatic Rust to index manually (i.e. `[i*N+j]`) or to use a crate to do it for you, e.g. `ndarray`. For instance:

```
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;
        }
    }
}
```

⟹

```
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 (a[1][1] is beside a[1][2]) as items in a slice are laid out an equal distance from each other: `https://doc.rust-lang.org/std/primitive.slice.html`. OpenGL, on the other hand, is supposedly column-major.

Strangely enough, sometimes you want to do things the column-major way even though it's "wrong". If your two dimensional array is of an appropriate size then by intentionally hitting things in the "wrong" order, you'll trigger all your page faults up front and load all your pages into cache and then you can go wild. This was suggested as a way to make matrix multiplication faster for a sufficiently large matrix: `https://www.intel.com/content/www/us/en/developer/articles/technical/loop-optimizations-where-blocks-are-required.html?wapkw=loop%20optimization...`

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

$\Longrightarrow$

```
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

Some optimizations affect low level code generation; here are the ones that `rustc` can do.

Cold. 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 generate code tuned to particular processors and processor variants (by using instructions available for certain processors, and by modifying the cost model). 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're running where you compile. See [Wil20] for a more detailed discussion.

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. They are much less of an issue for Rust but you may well be programming in C or C++ someday soon.

Alias and Pointer Analysis

I made passing references above to the fact that compiler optimizations often need to know about what parts of memory each statement accesses—things like "neither `x` nor `y` change". This is easy to establish when talking about scalar variables which are stored on the stack. 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 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.

Automatic parallelization is a thing. In general, it's hard. Rayon does it a bit (brief mentions to it in Appendix C). In Rust, controlled aliasing makes automatic parallelization much more tractable. Shape analysis builds on pointer analysis to determine that data structures are indeed trees rather than lists.

For a Rust-centric discussion: `https://doc.rust-lang.org/nomicon/aliasing.html`.

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. For Rust, indirect function calls (function pointers) and dynamic dispatch through traits are challenges to call graph construction[1].

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 both Rust and 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[2]:

```
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. By the way, "Rapid Type Analysis" (applied to C++, not sure if it's used in Rust) analyzes the entire program, could hypothetically observe that only `Baz` objects are ever instantiated (not true here), and would in that case enable devirtualization of the `to.foo()` 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.

[1] `https://blog.japaric.io/stack-analysis-2/`
[2] Inspired by code in previous footnote.

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.

OK, so inlining removes overhead. Sounds like better performance! Let's inline everything!

The Other Side of Inlining. Inlining has one big downside: your program size is going to increase. This is worse than you think: fewer cache hits and therefore more trips to memory. Some inlines can grow very rapidly – just from this your performance may go down.

Note also that inlining is merely a suggestion to compilers [GNU16]. They may ignore you. For C/C++taking the address of an "inline" function and using it; or virtual functions (in C++) will get you ignored quite fast.

Implications of inlining. Inlining can make your life worse in two ways. First, 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.

Second, it can be a problem for 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. (Congratulations, you made a non-binary-compatible change!). This would not be a problem for non-inlined functions—programs execute the new function dynamically at runtime.

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 obviously terrible Rust code:

```
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. If `N` was instead some memory location o with a unique pointer to it, then we would know whether or not `f()` has access to that unique pointer (and, in particular, there wouldn't exist some other pointer also pointing to o). For a shared object, we check whether the callee requests write permission to any object. In any case, we're less likely to have random state hanging around that may or may not be accessed by a function.

Tail Recursion Elimination. This optimization is mandatory in some functional languages; we replace a call by a `goto` at the compiler level. It is not mandatory in C/C++/Rust. Consider this example[3]:

```
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)
}
```

[3] https://stackoverflow.com/questions/59257543/when-is-tail-recursion-guaranteed-in-rust

Here, `fibonacci_lr` doesn't need to return control to its caller (because the recursive call is in tail position, i.e. the last thing that happens in the function). Doing the tail recursion elimination avoids function call overhead and reduces call stack use.

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 [Hub14] summarizes more recent history (compared to how long compilers have been around). 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 [Hub15] 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. LLVM and Rust's use of it also include various flavours of LTO. I couldn't find much information about what happens specifically for Rust; I'd expect the LLVM details below to apply. LLVM LTO can, however, optimize across source languages, i.e. if your program contains both C and Rust, the compiler and linker can optimize both using the intermediate representation.

Impact. gcc LTO 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 [LLV17], while gcc details can be found at [Die09].

References

[Die09] Diego Novillo. LinkTimeOptimization, 2009. Online; accessed 22-December-2017. URL: `https://gcc.gnu.org/wiki/LinkTimeOptimization`.

[GNU16] GNU Compiler Collection. An inline function is as fast as a macro, 2016. Online; accessed 6-January-2016. URL: `https://gcc.gnu.org/onlinedocs/gcc/Inline.html`.

[Hub14] Jan Hubička. Linktime optimization in GCC, part 1–brief history, 2014. Online; accessed 22-December-2017. URL: `http://hubicka.blogspot.ca/2014/04/linktime-optimization-in-gcc-1-brief.html`.

[Hub15] Jan Hubička. Link time and inter-procedural optimization improvements in GCC 5, 2015. Online; accessed 22-December-2017. URL: `http://hubicka.blogspot.ca/2015/04/GCC5-IPA-LTO-news.html`.

[LLV17] LLVM Project. LLVM link time optimization: Design and implementation, 2017. Online; accessed 22-December-2017. URL: `https://llvm.org/docs/LinkTimeOptimization.html`.

[Wil20] Nick Wilcox. Target Feature vs Target CPU for Rust, July 2020. Online; accessed 2020-11-19. URL: `https://www.nickwilcox.com/blog/target_cpu_vs_target_feature/`.