Lecture20 — Compiler Optimizations

Compiler Optimizations

"Is there any such thing as a free lunch?"

Compiler optimizations really do feel like a free lunch. But what does it really mean when you say `-O2`? 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.

Many pages on the Internet describe optimizations. Here's one that contains good examples:

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

You can find a full list of `gcc` options here:

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

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.

`gcc` **optimization levels.**   Here's what `-On` means for `gcc`. Other compilers have similar (but not identical) optimization flags.

- `-O0` (default): Fastest compilation time. Debugging works as expected.
- `-O1` (`-O`): Reduce code size and execution time. No optimizations that increase compiliation time.
- `-O2`: All optimizations except space vs. speed tradeoffs.
- `-O3`: All optimizations.
- `-Ofast`: All `-O3` optimizations, plus non-standards compliant optimizations, particularly `-ffast-math`. (Like `-fast` on the Solaris compiler.)

  This flag turns off exact implementations of IEEE or ISO rules/specifications for math functions. Generally, if you don't care about the exact result, you can use this for a speedup.

Scalar Optimizations

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

Constant folding.   Probably the simplest optimization one can think of. Tag line: "Why do later something you can do now?" We simply translate:

$$i = 1024 * 1024 \implies i = 1048576$$

Enabled at all optimization levels. 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.

```
a = (c + d) * y;
b = (c + d) * z;

w = 3;
x = f(); y = x;
z = w + y;
```

Enabled at `-O2`*,* `-O3` *or with* `-fgcse`*.* These flags actually enable a global (i.e. across-basic-blocks) CSE pass. This also enables global constant and copy propagation.

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`.

These scalar optimizations are more complicated in the presence of pointers, e.g. `z = *w + y`. More next time.

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:

```
int f(int x) {
  return x * 2;
}
```

```
int 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 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.

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 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.

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.

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. Here's an example.

```
for (int i = 0; i < 4; ++i)         ⟹      f(0); f(1); f(2); f(3);
    f(i)
```

Enabled with -funroll-loops.

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. For instance, in C:

```
for (int i = 0; i < N; ++i)              for (int j = 0; j < M; ++j)
    for (int j = 0; j < M; ++j)    ⟹         for (int i = 0; i < N; ++i)
        a[j][i] = a[j][i] * c                    a[j][i] = a[j][i] * c
```

since C is *row-major* (meaning a[1][1] is beside a[1][2]), rather than *column-major*.

Enabled with -floop-interchange.

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...

Loop fusion. This optimization is like the OpenMP collapse construct; we transform

```
for (int i = 0; i < 100; ++i)
    a[i] = 4                             for (int i = 0; i < 100; ++i) {
                                             a[i] = 4
                               ⟹            b[i] = 7
for (int i = 0; i < 100; ++i)           }
    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 (int i = 0; i < 100; ++i) {         s = x * y;
    s = x * y;                          for (int i = 0; i < 100; ++i) {
    a[i] = s * i;              ⟹            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 two examples.

Branch Prediction. `gcc` attempts to guess the probability of each branch to best order the code. (For an `if`, fall-through is most efficient. Why?)

This isn't quite an optimization, but you can use `__builtin_expect(expr, value)` to help GCC, if you know the run-time characteristics of your program. An example, from the Linux kernel:

```
#define likely(x)      __builtin_expect((x),1)
#define unlikely(x)    __builtin_expect((x),0)
```

Architecture-Specific. `gcc` can also generate code tuned to particular processors and processor variants. You can specify this using `-march` and `-mtune`. (`-march` implies `-mtune`). This will enable specific instructions that not all CPUs support (e.g. SSE4.2). For example, `-march=corei7`.

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.

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 {
  public:
    virtual void m();
};
```

```
class B : public A {
  public:
    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 com-

pilers are still relatively new; Hubička [Hub14] 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 [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.

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`.

[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`.