Lecture 20 — Compiler Optimizations

Patrick Lam
patrick.lam@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

October 22, 2019

Lunch Time, Already?

"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.

You Do Your Job...

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.

References

Many pages on the Internet describe optimizations.

Here's one that contains good examples:

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

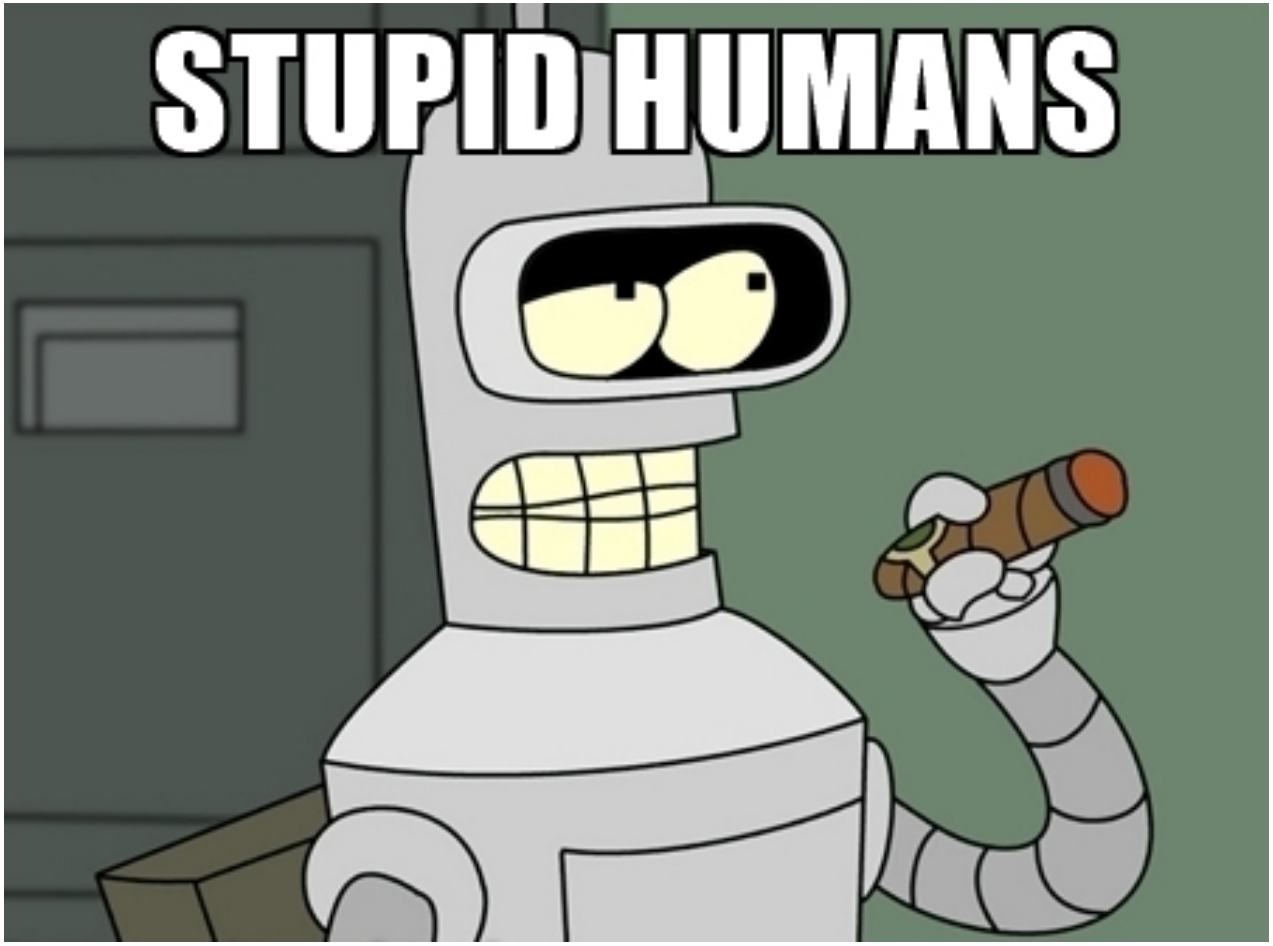
You can find a full list of `gcc` options here:

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

About Compiler Optimizations

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

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



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`.

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 at all optimization levels.

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.

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

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

Enabled at `-O2`, `-O3` *or with* `-fgcse`

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.

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

```
int 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.

Sane languages include array bounds checks.

Loop optimizations can eliminate array bounds checks if they can prove that the loop never iterates past the array bounds.

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 (int i = 0; i < 4;
     ++i)
    f(i)
```

$\Longrightarrow$

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

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

$\Longrightarrow$

```
for (int j = 0; j < M; ++j)
    for (int i = 0; i < N; ++i)
        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`.

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)
    b[i] = 7
```

$\Longrightarrow$

```
for (int i = 0; i < 100; ++i) {
    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 (int i = 0; i <
     100; ++i) {
    s = x * y;
    a[i] = s * i;
}
```

```
s = x * y;
for (int i = 0; i < 100; ++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.

`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 runtime behaviour...

In 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.

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

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.

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.

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 Graph

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.

(In general for 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.

```
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.

Devirtualization

‘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

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`.

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.

```
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.

```
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 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.

It Works!

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

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