Lecture 29—Compiler Optimizations

ECE 459: Programming for Performance

March 18, 2014

Last Time

Investigated impact of using abstractions.

Today: how compilers actually make code faster.

Part I

Compiler Optimizations

Introduction

We'll be looking at compiler optimizations.

Mostly related to performance.
Better gcc than you:

- For you, probably a waste of time
- Plus: optimizations make your code less readable.

Compilers have a host of optimization options.
We'll look at gcc.

Compiler Optimization

"Optimization" is a bit of a misnomer:
compilers don't generate "optimal" code.

Compilers do generate better code, though.

The program you wrote is too slow.
Contract of a compiler:
produce a program with same behaviour, but faster.

gcc Optimization Levels

`-O1` (`-O`)

- Reduce code size and execution time.
- No optimizations that increase compilation time.

`-O2`

- All optimizations except space vs. speed tradeoffs.

`-O3`

- All optimizations.

`-O0` (default)

- Fastest compilation time, debugging works as expected.

Disregard Standards, Acquire Speedup

`-Ofast`

- All `-O3` optimizations and non-standards compliant optimizations, particularly `-ffast-math`. (Like `-fast` on Solaris.)

This 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

Constant Folding

```
i = 1024 * 1024
```

Why do something later you can do now?

The compiler will not emit code that does the multiplication at runtime.
It will simply use the computed value

```
i = 1048576
```

- Enabled at all optimization levels.

Common Subexpression Elimination

`-fgcse`

- Perform a global[1] common subexpression elimination pass.
- (Also performs global constant and copy propagation.)
- Enabled at `-O2`, `-O3`.

Example:

```
a = b * c + g;
d = b * c * d;
```

Instead of computing `b * c` twice, we compute it once, and reuse the value in each expression

[1] across basic blocks

Constant Propagation

Moves constant values from definition to use.

- Valid if there's no intervening redefinition.

Example:

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

with constant propagation would produce:

```
int x = 14;
int y = 0;
return 0;
```

Copy Propagation

Replaces direct assignments with their values.
Usually runs after common subexpression elimination.

Example:

```
y = x
z = 3 + y
```

with copy propagation would produce:

```
z = 3 + x
```

Dead Code Elimination

`-fdce`

- Remove any code guaranteed not to execute.
- Enabled at all optimization levels.

Example:

```
if (0) {
    z = 3 + x;
}
```

would not be included in the final executable

Loop Unrolling

`-funroll-loops`

- Unroll loops, using a fixed number of iterations.

Example:

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

would be transformed to:

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

(Also useful for SIMD).

Loop Interchange

`-floop-interchange`
Enhances locality; big wins possible.

Example: in C the following:

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

would be transformed to this:

```
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]).
(Other possibility is **column-major**.)

Loop Fusion

Example:

```
for (int i = 0; i < 100; ++i)
    a[i] = 4

for (int i = 0; i < 100; ++i)
    b[i] = 7
```

would be transformed to this:

```
for (int i = 0; i < 100; ++i) {
    a[i] = 4
    b[i] = 7
}
```

Trade-off between data locality and loop overhead;
the inverse is also an optimization, called **loop fission**.

Loop-Invariant Code Motion

- Moves invariants out of a loop.
- Also called loop hoisting.

Example:

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

would be transformed to this:

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

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

Devirtualization

`-fdevirtualize`

- Attempt to convert virtual function calls to direct calls.
- Enabled with `-O2`, `-O3`.

Virtual functions impose overhead and impede other optimizations.

C++: must read the object's vtable and branch to the correct function.

Devirtualization (example)

Example:

```
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 RTTI access;

instead just call to B's `m`.

Needs call graph, which can be tricky to compute (more soon).

Scalar Replacement of Aggregates

Aliasing and Pointer Analysis

We've seen `restrict`: tell compiler that variables don't alias.

Pointer analysis automatically tracks the variables in your program to determine whether or not they alias.

If they don't alias, we can reason about side effects, reorder accesses, and do other types of optimizations.

Call Graph

- A directed graph showing relationships between functions.

Relatively simple to compute in C
(function pointers complicate things).
Hard for virtual function calls (C++/Java).

Virtual calls require pointer analysis to disambiguate.

Importance of Call Graphs

The call graph can enable optimization of the following:

```
int n;

int f() { /* opaque */ }

int main() {
    n = 5;
    f();
    printf("%d\n", n);
}
```

We could propagate the constant value 5 in `main()`, as long as `f()` does not write to `n`.

Tail Recursion Elimination

`-foptimize-sibling-calls`

- Optimize sibling and tail recursive calls.
- Enabled at `-O2` and `-O3`.

Example:

```
int bar(int N) {
    if (A(N))
        return B(N);
    else
        return bar(N);
}
```

Compiler can just replace the call to `bar` by a `goto`.
Avoids function call overhead and reduces call stack use.

Branch Predictions

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

Use `__builtin_expect(expr, value)` to help GCC, if you know the run-time characteristics of your program.

Example (in the Linux kernel):

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

Architecture-Specific

Two common ones: `-march` and `-mtune`
(`-march` implies `-mtune`).

- enable specific instructions that not all CPUs support (SSE4.2, etc.)
- **Example:** `-march=corei7`
- Good to use on your local machine, not so much for shipped code.

Part II

Profile-Guided Optimization

Moving Beyond Purely-Static Approaches

So far: purely-static (compile-time) optimizations.

Static approach has limitations:

- how aggressively to inline?
- which branch to predict?

Dynamic information answers these questions better.

Profile-Guided Optimization Workflow

1. Compile the program;
   produce an instrumented binary.
2. Run the instrumented binary:
   on a representative test suite;
   get run-time profiling information.
3. Compile final version, with the profiles you've collected.

Price of Performance from Profile-Guided Optimization

(–) Complicates the compilation process.
(+) Produces faster code (if inputs are good).

How much faster? 5–25%, per Microsoft.

Solaris, Microsoft VC++, and GNU gcc all support profile-guided optimization (to some extent).

Another Option

Just-in-time compilers (like Java Virtual Machines) automatically do profile-guided optimizations:

- compile code on-the-fly anyway;
- always collecting measurements for the current run.

What's Profiling Data Good For?

Synergistic optimizations:

- Inlining: inlining rarely-used code is a net lose;
  inlining often-executed code is a win,
  enables more optimizations.

  Everyone does this.

- Improving Cache Locality: make commonly-called functions, basic blocks adjacent;
  improves branch prediction (below) and with `switch`.

More Uses of Profiling Data

- Branch Prediction: a common architecture assumption—
  forward branches not taken, backwards branches taken.
  can make these assumptions more true with profiles;
  for a forward branch, make sure common case is fall-through.

- Virtual Call Prediction: inline the most common target
  (guarded by a conditional).

Part III

Summary

Summary

Got a feel for what the optimization levels do.
We saw examples of compiler optimizations. For instance:

- Scalar optimizations.
  (e.g. common subexpression elimination,
  constant propagation, copy propagation).
- Redundant code optimizations.
  (e.g. dead code elimination).
- Loop optimizations.
  (e.g. loop interchange, loop fusion, loop fission,
  software pipelining).
- Alias and pointer analysis; call graphs.
  (uses: devirtualization, inlining, tail recursion elimination).

Summary II

Full list of gcc options:

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

More optimization examples:

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

Also, profile-guided optimization:

see what actually happens at runtime, optimize for that.