Lecture 17 - Compiler Optimizations

ECE 459: Programming for Performance

Jon Eyolfson

University of Waterloo

February 13, 2012

Introduction

- Today, we'll be looking at compiler optimizations
- Most are related to performance, good to avoid doing these by yourself, since:
  - Likely waste time
  - Make your code more unreadable
- Compiler's have a host of optimization options, we'll look at gcc

GCC Optimization Levels

-O1 (-O)

- Reduce code size and execution time
- No optimizations that increase compiliation time

-O2

- All optimizations except space-speed tradeoff ones

-O3

- All optimizations

-O0 (default)

- Fastest compilation time, debugging performs as expected

Disregard Standards, Acquire Speedup

`-Ofast`

- All `-O3` optimizations and non-standard compliant optimizations, namely `-ffast-math`

Turns off exact implementations of IEEE or ISO rules/specifications for math functions

Generally, if you don't care about the exact results, you can use this for a speedup

Constant Folding

```
i = 1024 * 1024
```

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 common subexpression elimination pass
- This pass also performs global constant and copy propagation
- Enabled with -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

Constant Propagation

Moves the constant values from definition to use

- Valid if there's no redefinition of the variable

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 required to run 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 that is 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 any loops with a set 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)
```

Loop Interchange

`-floop-interchange`

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]), the 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
}
```

There is a trade-off here between data locality and loop overhead, the opposite of this is called **loop fission**

Loop-Invariant Code Motion

- Moves invariants out of the 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;
}
```

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

Devirtualization (1)

`-fdevirtualize`

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

Virutal functions impose some overhead, for instance in C++, you must read the objects RTTI (run-time type information) then effectively branch to the correct function

Devirtualization (2)

Example:

```
class A {
    virtal void m();
};

class B : public A {
    virutal void m();
}

int main(int argc, char *argv[]) {

    std::unique_ptr<A> t(new B);
    t.m();
}
```

could eliminate reading the RTTI and just insert a call to B's `m`

Scalar Replacement of Aggregates

`-fipa-sra`

- Replace references by values when appropriate
- Enabled at -O2 and -O3

Example:

```
{
    std::unique_ptr<Fruit> a(new Apple);
    std::cout << color(a) << std::endl;
}
```

could be optimized to:

```
std::cout << "Red" << std::endl;
```

if the compiler knew what `color` does

Aliasing and Pointer Analysis

- We've seen using `restrict` to tell the compiler variables do not alias
- *Pointer analysis* tracks the variables in your program to determine whether or not they alias
- If they don't alias, we can reorder them and do other types of optimizations

Call Graph

- A directed graph that shows relationships between functions
- Relativity simple in C, hard for virtual function calls (C++/Java)
- Virtual calls require pointer analysis

Importance of Call Graphs

Having the call graph allows us to know if the following can be optimized:

```
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

`-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);
}
```

We can just replace the call to `bar` by a `goto` at the compiler level, this way we avoid having overhead of a function call and increasing our call stack

Branch Predictions

- GCC attempts to guess the probability of branches in order to do the best code ordering
- You can 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`)

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

Summary

- A feel of what the optimization levels do

- What some of the compiler optimizations are

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