Compiler Optimization

CSE 220: Systems Programming

Ethan Blanton

Department of Computer Science and Engineering

University at Buffalo

Big Wins *vs.* Many Wins

The biggest wins in optimization are algorithmic.

If you can:

- Reduce the size of your data
- Reduce the iterations of your loops
- Reduce the number of traversals
- …

…then you may make things asymptotically faster.[1]

However, constant factors matter too.

[1]See CSE 250.

Constant Factors

Small constant overheads can add up in a program.

E.g., the order in which you perform operations can matter.

In a previous lecture, we saw the order of array access make a factor of twenty difference.

You must understand the system to avoid these traps.

Optimizing Compilers

Modern compilers are optimizing compilers.

They understand the machine very deeply, and:

- Very effectively allocate resources such as registers
- Reorder and eliminate code
- Factor out common operations
- …

They cannot improve your algorithms, however. ¶

They can also be fooled by certain constructions.

Principles of Optimization

Optimizing compilers have their own Hippocratic Oath:
First, change not semantics.[2]

An optimizing compiler must not change correct program behavior.

They can also only work with static information.
That is, information that is known at compile time.

This can prevent compilers from making many optimizations.

[2] They don't actually take oaths.

Illegal Optimizations

For example, a compiler cannot optimize when:

- Code processes some data
- The data has certain properties known to the programmer
- Those properties ensure that a certain code path cannot run
- The data is not known to the compiler

Optimization Practice

Historically, C compilers have only optimized within functions.

Our gcc can do inter-procedural optimization.

These are quite limited in practice, however.

This means that function calls prevent optimization.

Platform-independent Optimization

Some optimizations are always a good idea. ¶

They are not machine-specific and can be used widely.

These optimizations typically have to do with code semantics.

We will look at some examples:

- Constant folding
- Code motion
- Reduction in strength

Constant Folding

Constant folding is computation of constants at compile time.

Consider this code:

```
int i = 2 + 3;
```

Could `i` be anything but 5 after this line?

Nope! Add it at compile time!

In the real world, this can be harder to find.[3]

[3] Consider: macros, named constants, *etc.*

Code Motion

The compiler can also move code to:

- Reduce redundancy
- Avoid costly but unnecessary operations

Common examples:

- Loop calculations independent of the loop index
- Early variable initializations

Motion Example

```
void set_row(double *dst, double *src,
             long row, long elements) {
    for (long col = 0; j < n; j++) {
        dst[elements*row + col] = src[col];
    }
}

void set_row(double *dst, double *src,
             long row, long elements) {
    long first = row*elements;
    for (long col = 0; j < n; j++) {
        dst[first + col] = src[col];
    }
}
```

Motion Example 2

```
/* Sum neighbors of i,j in 2D array */
above = val[(i-1)*n + j  ];    // (i*n) - n + j
below = val[(i+1)*n + j  ];    // (i*n) + n + j
left  = val[    i*n + j-1];    // (i*n) + j - 1
right = val[    i*n + j+1];    // (i*n) + j + 1
sum   = above + below + left + right;
```

Motion Example 2

```
long inj = i*n + j;
above = val[    inj - n  ];    // (i*n) - n + j
below = val[    inj + n  ];    // (i*n) + n + j
left  = val[    inj - 1  ];    // (i*n) + j - 1
right = val[    inj + 1  ];    // (i*n) + j + 1
sum   = above + below + left + right;
```

Reduction in Strength

Reduction in strength is replacement of expensive operations with cheaper operations.

Typically, expensive means slow and cheap means fast.

For example:

- multiply and divide are expensive
- shift operations are cheap
- Shifts are integer multiply/divide by powers of two

Sequential Computations

Sometimes the compiler can identify sequential computations:

```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        dst[n*i + j] = src[j];    // dst[i] = src[j]
    }
}
```

Note that n*i changes by n each iteration.

Therefore you can start with zero and add by n.

Sequential Computations

Sometimes the compiler can identify sequential computations:

```
int ni = 0;
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        dst[ni + j] = src[j];
    }
    ni += n;
}
```

Note that n*i changes by n each iteration.

Therefore you can start with zero and add by n.

Optimization Blockers

As previously mentioned, some things block optimizations.

- Data-dependent operations
- Procedure calls (without inter-procedural optimization)
- Pointer aliases (more than one pointer to an object)
- …

Optimizing Across Procedure Calls

Why are procedure calls problematic?

- They might have side effects (alter global state, do I/O).
- They might not be deterministic.
- They might modify pointers.

The compiler must not change semantics!

Optimizations around procedure calls are therefore weakened.

Forbidden Code Motion

```
for (size_t i = 0; i < strlen(s); i++) {
    s[i] = tolower(s[i]);
}
```

Our gcc will compute `strlen()` `strlen()` times.

This is $O(n^2)$![4]

For a 1 MB character string, this takes minutes.
(For comparison, a 1920x1080 screen is about 8 MB.)

Why can't it compute the `strlen()` once?

[4] See CSE 250 again.

What is `strlen()`, Anyway?

The compiler cannot assume that strlen does not alter `s`.

The compiler cannot assume that strlen(s) is always the same.

The compiler treats `strlen()` as a black box.

We can fix this ourselves: perform our own code motion.

Summary

- Algorithmic improvements remain key.
- Knowing how the compiler works help produce better code.
- Optimizing compilers must not change semantics.
- Compilers use static information.
- We covered:
  - Constant folding
  - Code motion
  - Reduction in strength
- Procedures are problematic.

References I

Required Readings

[1] Randal E. Bryant and David R. O'Hallaron. *Computer Science: A Programmer's Perspective*. Third Edition. Chapter 5: Intro, 5.1-5.6. Pearson, 2016.

License