



# **Code Optimization**

## **ICS312 Machine-Level and Systems Programming**

Henri Casanova ([henric@hawaii.edu](mailto:henric@hawaii.edu))

# Why?

- Code that runs faster is always a good goal
  - Saves on time, energy
- This is different from making an algorithm (i.e., with low asymptotic complexity, with low constant factors)
- Let's assume we have an implementation of a good algorithm with good data structures
- How can we make the code faster?
- The goal: use the hardware as best as possible
  - e.g., if you have a Multi-core machine (which you do), write multi-threaded code
  - e.g., if you have a cache (which you do), write code with good data locality behavior
  - “tweak” the code so that it runs faster

# Trade-offs

- Unfortunately, code performance often conflicts with other concerns that are also important
  - Correctness
    - When trying to make code go fast one often breaks it
  - Readability
    - Fast code can require more lines!
    - Modularity can hurt performance
      - e.g., too many method calls
  - Portability
    - Code that is fast on machine A can be slow on machine B
    - At the extreme, highly optimized code is not portable at all, and in fact is done in hardware!



# Ok, so now what?

- Your profiler told you that most of the time is spent in some part of the code
- You now look at this part of the code
  - For now let's pretend you have a dumb compiler that doesn't do any optimization
- Let's try to
  - Hand-optimize `code_to_optimize.c` (Web site)
    - Making a copy of it
  - How much faster can we get it to be?
    - Compiling it with `-O0`

# Code Optimization Techniques

- We (probably) did:
  - Moving invariant code out of loops

```
for (i=0; i < n; i++) {  
    x += i * (n * 3);  
}
```

```
int tmp = n*3;  
for (i=0; i < n; i++) {  
    x += i * tmp;  
}
```

- array removal

```
for (i=0; i < n; i++) {  
    A[i] = 1;  
}
```

```
int *A_ptr = &(A[0]);  
for (i=0; i < n; i++) {  
    *(A_ptr++) = 1;  
}
```

- loop unrolling

```
for (i=0; i < 20; i++) {  
    A[i] = 1;  
}
```

```
for (i=0; i < 20; i+=2) {  
    A[i] = 1; A[i+1] = 1;  
}
```

# Code Optimization Techniques

- We (probably) did:

- Moving invariant code out of loops

```
for (i=0; i < n; i++)  
    x += i * (n * 3);  
}
```

saves  
computation

```
tmp = n*3;  
for (i=0; i < n; i++) {  
    x += i * tmp;  
}
```

- array removal

```
for (i=0; i < n; i++)  
    A[i] = 1;  
}
```

saves  
address  
computation

```
A_ptr = &(A[0]);  
for (i=0; i < n; i++) {  
    *A_ptr++ = 1;  
}
```

- loop unrolling

```
for (i=0; i < 20; i++)  
    A[i] = 1;  
}
```

saves  
loop index  
comparison

```
for (i=0; i < 20; i+=2) {  
    A[i] = 1; A[i+1] = 1;  
}
```

# Code Optimization Techniques

- There are many others:
  - Inlining (remove function calls), assignment of variables to registers, dead code elimination, algebraic simplification, moving invariant code out of loops, constant propagation, control flow simplification, strength reduction, etc.
- Some are really simply but tedious for humans
  - e.g., strength reduction
    - e.g., don't do " $i * 2$ " but do " $i << 1$ "
    - e.g., don't do " $x = a / 4.0$ " but do " $x = a * 0.25$ "
- Some can only be done at the assembly language level
- A particular tricky one is instruction scheduling...

# Now what?

- We could apply as many optimization techniques to code as we can, but this would result in completely unreadable/undebuggable code
  - And we would write assembly quite a bit
- People in the 60's/70's used to do this though
  - And were very good at it
- But today, the **compiler** comes to the rescue
  - Otherwise, what are those decades of compiler research for?
- Modern compilers can do a lot for you



# Compiler Optimization

- All modern compilers perform some automatic optimization when generating code
  - In fact, you implement some of those in a graduate-level compiler class, and sometimes at the undergraduate level
- Most compilers provide several levels of optimization
  - -O0: No optimization
    - Even though a little bit is done
  - -O1, -O2, .... -OX, -Ofast
- The higher the optimization level the higher the probability that a debugger may have trouble dealing with the code
  - Always debug without compiler optimization (typically enforced by your compiler anyway)
- Some compiler will flat out tell you that higher levels of optimization may break some code!

# Compiler Optimization

- gcc is a pretty good, free compiler
  - -Os: Optimize for size
    - Some optimizations increase code size tremendously
- Let's do a “man gcc” and look at the many optimization options
  - one can pick and choose
  - or just use standard sets via O1, O2, O3
- The most fancy compilers are typically the ones done by vendors
  - You can't sell a good machine if it has a bad compiler
  - Compiler technology used to be really poor
  - also, languages used to be designed without thinking of compilers (FORTRAN, Ada)
  - no longer true: every language designer has in-depth understanding of compiler technology today



# Compiler Optimization

- Let's go back to our `code_to_optimize.c` and try things out:
  - How good is the compiler on the original code?
  - How good is it on the hand-optimized code?
  - Can we do only some hand-optimizations?
  - What about with another compiler?



# Let's use Compiler Explorer...

- Let's try to get a sense of how good compilers are
- Let's go to Compiler Explorer: <https://gcc.godbolt.org/>
- And let's see how code is optimized...



# **Were are we now?**

- There are many techniques to optimize code
- But compilers do a lot of them for us automatically
- So, does it mean that we, as software developers have nothing to worry about?
- Sort of... let's see an example

# By-hand Optimization of Matrix Multiplication

```
for(i = 0; i < SIZE; i++) {  
    for(j = 0; j < SIZE; j++) {  
        for(k = 0; k < SIZE; k++) {  
            c[i][j] += a[i][k] * b[k][j];  
        }  
    }  
}
```

- Turned array accesses into pointer dereferences
- Assign to each element of c just once

```
for(i = 0; i < SIZE; i++) {  
    int *orig_pa = &a[i][0];  
    for(j = 0; j < SIZE; j++) {  
        int *pa = orig_pa;  
        int *pb = &a[0][j];  
        int sum = 0;  
        for(k = 0; k < SIZE; k++) {  
            sum += *pa * *pb;  
            pa++;  
            pb += SIZE;  
        }  
        c[i][j] = sum;  
    }  
}
```

# A Few Results

<b>My laptop</b>	Simple	Optimized
gcc -O0	57.01s	34.02s
gcc -O4	16.01s	15.97s

<b>The Cray</b>	Simple	Optimized
gcc -O0	28.03s	12.03s
gcc -O4	3.93s	3.01s
icc -O0	25.95s	12.03s
icc -O4	1.919s	3.98s

<b>My Server</b>	Simple	Optimized
gcc -O0	42.99s	20.05s
gcc -O4	4.05s	4.97s

# A Few Results

My laptop	Simple	Optimized
gcc -O0	57.01s	34.02s
gcc -O4	16.01s	15.97s

If the compiler does not optimize, then our by-hand optimization are often useful

The Cray	Simple	Optimized
gcc -O0	28.03s	12.03s
gcc -O4	3.93s	3.01s
icc -O0	25.95s	12.03s
icc -O4	1.919s	1.98s

My Server	Simple	Optimized
gcc -O0	42.99s	20.05s
gcc -O4	4.05s	4.97s



# A Few Results

My laptop	Simple	Optimized
gcc -O0	57.01s	34.02s
gcc -O4	16.01s	15.97s

But if the compiler optimizes, out by-hand optimizations can be useless or even harmful!  
Or they can help!

The Cray	Simple	Optimized
gcc -O0	28.03s	12.03s
gcc -O4	3.93s	3.01s
icc -O0	25.95s	12.03s
icc -O4	1.919s	3.98s

My Server	Simple	Optimized
gcc -O0	42.99s	20.05s
gcc -O4	4.05s	4.97s

# A Few Results

<b>My laptop</b>	Simple	Optimized
gcc -O0	57.01s	34.02s
gcc -O4	16.01s	15.97s

Different compilers can be quite different on the same machine

<b>The Cray</b>	Simple	Optimized
gcc -O0	28.03s	12.03s
gcc -O4	3.93s	3.01s
icc -O0	25.95s	12.03s
icc -O4	1.919s	3.98s

<b>My Server</b>	Simple	Optimized
gcc -O0	42.99s	20.05s
gcc -O4	4.05s	4.97s

# Limits to Compiler Optimization

- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
  - e.g., data ranges may be more limited than variable types suggest
    - e.g., using an “`int`” in C for what could be an enumerated type
- Most analysis is performed only **within procedures**
  - whole-program analysis is too expensive in most cases
- Analysis is based only on *static* information
  - compiler cannot anticipate run-time inputs
- When in doubt, the compiler must be **conservative**
  - Cannot perform optimization if it changes program behavior under *any* realizable circumstance
    - Even if completely unlikely to a human (“But x will never be negative! Why can’t the compiler understand this???”)

# Conclusion

- Code optimization is fascinating and difficult
- Compilers are not all the same
  - Some are better than other
  - Often commercial ones are great (but expensive)
- Sometimes the compiler optimize things in ways you could never have done
- Sometimes the compiler will not optimize things that seem obvious to you as a human
- There is no golden rule besides: **Doing all optimization by hand is a bad idea in general**