Lecture #32 – Code Optimization
Good news, everyone!

• The unofficial end-term course feedback form is available.
Recap

• Debugging
  • Defects and Failures
  • Scientific Debugging
  • Tools

• Design
  • Managing complexity
  • Communication
  • Naming
  • Comments
Learning Goals

• Understand how we can optimize our code to improve efficiency and speed
• Learn about the optimizations GCC can perform
Plan for Today

- What is optimization?
- GCC Optimization
- Limitations of GCC Optimization
- Caching revisited

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class
Lecture Plan

• What is optimization?
• GCC Optimization
• Limitations of GCC Optimization
• Caching revisited
Optimization

• Optimization is the task of making your program faster or more efficient with space or time. Later you will learn about explorations of efficiency with Big-O notation!

• *Targeted, intentional* optimizations to alleviate bottlenecks can result in big gains. But it’s important to only work to optimize where necessary.
Optimization

Most of what you need to do with optimization can be summarized by:

1) If doing something seldom and only on small inputs, do whatever is simplest to code, understand, and debug
2) If doing things thing a lot, or on big inputs, make the primary algorithm’s Big-O cost reasonable
3) **Let gcc do its magic from there**
4) Optimize explicitly as a last resort
Lecture Plan

• What is optimization?
• GCC Optimization
• Limitations of GCC Optimization
• Caching revisited
GCC Optimization

• Today, we’ll be comparing two levels of optimization in the gcc compiler:
  • gcc -O0  // mostly just literal translation of C
  • gcc -O2  // enable nearly all reasonable optimizations
  • (we use -Og, like -O0 but with less needless use of the stack)

• There are other custom and more aggressive levels of optimization, e.g.:
  • -O3  //more aggressive than O2, trade size for speed
  • -Os  //optimize for size
  • -Ofast  //disregard standards compliance (!!)

• Exhaustive list of gcc optimization-related flags:
  • https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
Example: Matrix Multiplication

Here’s a standard matrix multiply, a triply-nested for loop:

```c
void mmm(double a[][DIM], double b[][DIM], double c[][DIM], int n) {
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
         for (int k = 0; k < n; k++) {
            c[i][j] += a[i][k] * b[k][j];
         }
      }
   }
}
```

./mult // -O0 (no optimization)
matrix multiply 25^2: cycles 0.43M
matrix multiply 50^2: cycles 3.02M
matrix multiply 100^2: cycles 24.82M

./mult_opt // -O2 (with optimization)
matrix multiply 25^2: cycles 0.13M (opt)
matrix multiply 50^2: cycles 0.66M (opt)
matrix multiply 100^2: cycles 5.55M (opt)
GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling
- The Force
GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling
- The Force

(may be not 😎)
GCC Optimizations

Optimizations may target one or more of:
• Static instruction count
• Dynamic instruction count
• Cycle count / execution time
GCC Optimizations

- **Constant Folding**
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling
Constant Folding

**Constant Folding** pre-calculates constants at compile-time where possible.

```c
int seconds = 60 * 60 * 24 * n_days;
```

What is the consequence of this for you as a programmer? What should you do differently or the same knowing that compilers can do this for you?
Constant Folding

int fold(int param) {
    char arr[5];
    int a = 0x107;
    int b = a * sizeof(arr);
    int c = sqrt(2.0);
    return a * param + (a + 0x15 / c + strlen("Hello") * b - 0x37) / 4;
}
Constant Folding: Before (-O0)

0000000000400626 <fold>:

```
0000000000400626:  55               push %rbp
0000000000400627:  53               push %rbx
0000000000400628:  48 83 ec 08       sub $0x8,%rsp
000000000040062c:  89 fd             mov %edi,%ebp
000000000040062e:  f2 0f 10 05 da 00 00 movsd 0xda(%rip),%xmm0
0000000000400635:  00               
0000000000400636:  e8 d5 fe ff ff    callq 400510 <sqrt@plt>
000000000040063f:  f2 0f 2c c8        cvttsd2si %xmm0,%ecx
0000000000400645:  69 ed 07 01 00 00  imul $0x107,%ebp,%ebp
000000000040064a:  b8 15 00 00 00      mov $0x15,%eax
000000000040064b:  48 69 c0 23 05 00 00  imul $0x523,%rax,%rax
0000000000400653:  bf 04 07 40 00      movslq %ebx,%rbx
0000000000400658:  48 8d 44 18 c9      lea -0x37(%rax,%rbx,1),%rax
0000000000400667:  48 83 ec 08       add %ebp,%eax
0000000000400672:  48 83 c4 08       add $0x8,%rsp
0000000000400677:  5b               pop %rbx
0000000000400678:  5d               pop %rbp
        c3               retq
```
Constant Folding: After (-O2)

```
000000000004004f0 <fold>:
  4004f0: 69 c7 07 01 00 00    imul $0x107,%edi,%eax
  4004f6: 05 a5 06 00 00    add $0x6a5,%eax
  4004fb: c3    retq
  4004fc: 0f 1f 40 00    nopl 0x0(%rax)
```
GCC Optimizations

- Constant Folding
- **Common Sub-expression Elimination**
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling
Common Sub-Expression Elimination prevents the recalculation of the same thing many times by doing it once and saving the result.

```c
int a = (param2 + 0x107);
int b = param1 * (param2 + 0x107) + a;
return a * (param2 + 0x107) + b * (param2 + 0x107);
```
Common Sub-Expression Elimination prevents the recalculation of the same thing many times by doing it once and saving the result.

```c
int a = (param2 + 0x107);
int b = param1 * (param2 + 0x107) + a;
return a * (param2 + 0x107) + b * (param2 + 0x107);
```

This optimization is done even at `-O0`!
GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- **Dead Code**
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling
Dead Code

**Dead code elimination** removes code that doesn’t serve a purpose:

```c
if (param1 < param2 && param1 > param2) {
    printf("This test can never be true!\n");
}

// Empty for loop
for (int i = 0; i < 1000; i++);

// If/else that does the same operation in both cases
if (param1 == param2) {
    param1++;
} else {
    param1++;
}

// If/else that more trickily does the same operation in both cases
if (param1 == 0) {
    return 0;
} else {
    return param1;
}
```
Dead Code: Before (-O0)

```
00000000004004d6 <dead_code>:
  4004d6:    b8 00 00 00 00 00            mov    $0x0,%eax
  4004db:    eb 03                        jmp    4004e0 <dead_code+0xa>
  4004dd:    83 c0 01                    add    $0x1,%eax
  4004e0:    3d e7 03 00 00              cmp    $0x3e7,%eax
  4004e5:    7e f6                        jle    4004dd <dead_code+0x7>
  4004e7:    39 f7                        cmp    %esi,%edi
  4004e9:    75 05                        jne    4004f0 <dead_code+0x1a>
  4004eb:    8d 47 01                    lea    0x1(%rdi),%eax
  4004ee:    eb 03                        jmp    4004f3 <dead_code+0x1d>
  4004f0:    8d 47 01                    lea    0x1(%rdi),%eax
  4004f3:    f3 c3                        repz retq
```
Dead Code: After (-O2)

```
00000000004004f0 <dead_code>:
    4004f0: 8d 47 01              lea 0x1(%rdi),%eax
    4004f3: c3                      retq
    4004f4: 66 2e 0f 1f 84 00 00    nopw %cs:0x0(%rax,%rax,1)
    4004fb: 00 00 00                 xchg %ax,%ax
    4004fe: 66 90
```
GCC Optimizations

• Constant Folding
• Common Sub-expression Elimination
• Dead Code
• **Strength Reduction**
• Code Motion
• Tail Recursion
• Loop Unrolling
Strength Reduction

**Strength reduction** changes divide to multiply, multiply to add/shift, and mod to AND to avoid using instructions that cost many cycles (multiply and divide).

```c
int a = param2 * 32;
int b = a * 7;
int c = b / 3;
int d = param2 % 2;

for (int i = 0; i <= param2; i++) {
    c += param1[i] + 0x107 * i;
}
return c + d;
```
Strength Reduction: After (-O3)

unsigned udiv19(unsigned arg) {
    return arg / 19;
}

What really happens here?

\[
\frac{a}{19} \approx \frac{a \cdot \frac{2938661835}{2^{32}} + a \cdot \frac{2938661835}{2^{132}}}{2^4}
\]
\[
\frac{a}{19} \approx (a \cdot 2938661835 \cdot 2^{-32} + (a - a \cdot 2938661835 \cdot 2^{-32}) \cdot 2^{-1}) \cdot 2^{-4}
\]
\[
\frac{a}{19} \approx a \cdot \frac{7233629131}{137438953472}
\]

https://godbolt.org/z/Wq8ra3
GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- **Code Motion**
- Tail Recursion
- Loop Unrolling
Code Motion

**Code motion** moves code outside of a loop if possible.

```java
for (int i = 0; i < n; i++) {
    sum += arr[i] + foo * (bar + 3);
}
```

Common subexpression elimination deals with expressions that appear multiple times in the code. Here, the expression appears once, but is calculated each loop iteration.
GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling
Tail Recursion

Tail recursion is an example of where GCC can identify recursive patterns that can be more efficiently implemented iteratively.

```c
long factorial(int n) {
    if (n <= 1) {
        return 1;
    }
    else return n * factorial(n - 1);
}
```
GCC Optimizations

• Constant Folding
• Common Sub-expression Elimination
• Dead Code
• Strength Reduction
• Code Motion
• Tail Recursion
• **Loop Unrolling**
Loop Unrolling

Loop Unrolling: Do n loop iterations’ worth of work per actual loop iteration, so we save ourselves from doing the loop overhead (test and jump) every time, and instead incur overhead only every n-th time.

```c
for (int i = 0; i <= n - 4; i += 4) {
    sum += arr[i];
    sum += arr[i + 1];
    sum += arr[i + 2];
    sum += arr[i + 3];
} // after the loop handle any leftovers
```
Lecture Plan

• What is optimization?
• GCC Optimization
• Limitations of GCC Optimization
• Caching revisited
Limitations of GCC Optimization

GCC can’t optimize everything! You ultimately may know more than GCC does.

```c
int char_sum(char *s) {
    int sum = 0;
    for (size_t i = 0; i < strlen(s); i++) {
        sum += s[i];
    }
    return sum;
}
```

What is the bottleneck? `strlen` called for every character
What can GCC do? code motion – pull `strlen` out of loop
Limitations of GCC Optimization

GCC can’t optimize everything! You ultimately may know more than GCC does.

```c
void lower1(char *s) {
    for (size_t i = 0; i < strlen(s); i++) {
        if (s[i] >= 'A' && s[i] <= 'Z') {
            s[i] -= ('A' - 'a');
        }
    }
}
```

What is the bottleneck?
What can GCC do?

*strlen* called for every character
nothing!  *s* is changing, so GCC doesn’t know if length is constant across iterations. But we know its length doesn’t change.
Lecture Plan

• What is optimization?
• GCC Optimization
• Limitations of GCC Optimization
• Caching revisited
Caching

• Processor speed is not the only bottleneck in program performance – memory access is perhaps even more of a bottleneck!
• Memory exists in levels and goes from really fast (registers) to really slow (disk).
• As data is more frequently used, it ends up in faster and faster memory.
Caching

All caching depends on locality.

Temporal locality
• Repeat access to the same data tends to be co-located in TIME
• Intuitively: things I have used recently, I am likely to use again soon

Spatial locality
• Related data tends to be co-located in SPACE
• Intuitively: data that is near a used item is more likely to also be accessed
Optimizing Your Code

• Explore various optimizations you can make to your code to reduce instruction count and runtime.
  • More efficient Big-O for your algorithms
  • Explore other ways to reduce instruction count
    • Look for hotspots using callgrind
    • Optimize using –O2
    • And more...
Compiler Optimizations

Why not always just compile with -O2?

• Difficult to debug optimized executables – only optimize when complete
• Optimizations may not always improve your program. The compiler does its best, but may not work, or slow things down, etc. Experiment to see what works best!

Why should we bother saving repeated calculations in variables if the compiler has common subexpression elimination?

• The compiler may not always be able to optimize every instance. Plus, it can help reduce redundancy!
Recap

• What is optimization?
• GCC Optimization
• Limitations of GCC Optimization
• Caching revisited

Next time: Linking