CS107, Lecture 24
Optimization

Reading: B&O 5

Ed Discussion

Optimization

- Optimization is the task of making your program more efficient in **space** and **time**. You've studied Big-O notation in prerequisite courses, so you know something about efficiency already!
- **Targeted** and **intentional** optimizations designed to alleviate true bottlenecks can result in huge speed and memory gains.
  - But it's important to prioritize optimizations that really make an impact.

Optimization

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

1) If you're doing something infrequently, and only on small inputs, do whatever is simplest to code, understand, read, and debug.
2) If you're doing something often or on big inputs, make the primary algorithm's Big-O cost reasonably low.
3) Let **`gcc`** work its magic from there.
4) Optimize explicitly as a last resort.

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 also use **`–Og`**, which is like **`–O0`**, but it's more **`gdb`**-friendly)

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

- Optimizations may target one or more of:
  - static instruction count
  - dynamic instruction count, which translates to cycle count and execution time

- Exhaustive list of optimization-drive gcc flags is right here.

Example: Matrix Multiplication

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

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

```
./mult          // -O0 (no optimization)
matrix multiply  25^2: cycles   1.32M
matrix multiply  50^2: cycles  10.61M
matrix multiply 100^2: cycles  18.09M
```

```
./mult_opt    // -O2 (with optimization)
matrix multiply  25^2: cycles    0.19M (opt)
matrix multiply  50^2: cycles    2.04M (opt)
matrix multiply 100^2: cycles   13.92M (opt)
```

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 whenever possible.

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

```
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)

```
0000000000011b9 <fold>:
    11b9:   55                      push   %rbp
    11ba:   48 89 e5                mov    %rsp,%rbp
    11bd:   41 54                   push   %r12
    11bf:   53                      push   %rbx
    11c0:   48 83 ec 30             sub    $0x30,%rsp
    11c4:   89 7d cc                mov    %edi,-0x34(%rbp)
    11c7:   c7 45 ec 07 01 00 00    movl   $0x107,-0x14(%rbp)
    11ce:   8b 45 ec                mov    -0x14(%rbp),%eax
    11d1:   48 98                   cltq
    11d3:   89 c2                   mov    %eax,%edx
    11d5:   89 d0                   mov    %edx,%eax
    11d7:   c1 e0 02                shl    $0x2,%eax
    11da:   01 d0                   add    %edx,%eax
    11dc:   89 45 e8                mov    %eax,-0x18(%rbp)
    11df:   48 8b 05 2a 0e 00 00    mov    0xe2a(%rip),%rax        # 2010 <_IO_stdin_used+0x10>
    11e6:   66 48 0f 6e c0          movq   %rax,%xmm0
    11eb:   e8 b0 fe ff ff          call   10a0 <sqrt@plt>
    11f0:   f2 0f 2c c0             cvttsd2si %xmm0,%eax
    11f4:   89 45 e4                mov    %eax,-0x1c(%rbp)
    11f7:   8b 45 ec                mov    -0x14(%rbp),%eax
    11fa:   0f af 45 cc             imul   -0x34(%rbp),%eax
    11fe:   41 89 c4                mov    %eax,%r12d
    1201:   b8 15 00 00 00          mov    $0x15,%eax
    1206:   99                      cltd
    1207:   f7 7d e4                idivl  -0x1c(%rbp)
    120a:   89 c2                   mov    %eax,%edx
    120c:   8b 45 ec                mov    -0x14(%rbp),%eax
    120f:   01 d0                   add    %edx,%eax
    1211:   48 63 d8                movslq %eax,%rbx
    1214:   48 8d 05 ed 0d 00 00    lea    0xded(%rip),%rax        # 2008 <_IO_stdin_used+0x8>
    121b:   48 89 c7                mov    %rax,%rdi
    121e:   e8 1d fe ff ff          call   1040 <strlen@plt>
    1223:   8b 55 e8                mov    -0x18(%rbp),%edx
    1226:   48 63 d2                movslq %edx,%rdx
    1229:   48 0f af c2             imul   %rdx,%rax
    122d:   48 01 d8                add    %rbx,%rax
    1230:   48 83 e8 37             sub    $0x37,%rax
    1234:   48 c1 e8 02             shr    $0x2,%rax
    1238:   44 01 e0                add    %r12d,%eax
    123b:   48 83 c4 30             add    $0x30,%rsp
    123f:   5b                      pop    %rbx
    1240:   41 5c                   pop    %r12
    1242:   5d                      pop    %rbp
    1243:   c3                      ret
```

Constant Folding: After (-O2)

```
00000000000011b0 <fold>:
    11b0:   69 c7 07 01 00 00       imul   $0x107,%edi,%eax
    11b6:   05 a5 06 00 00          add    $0x6a5,%eax
    11bb:   c3                      retq
```

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: Less Contrived

```
unsigned int cf(unsigned long val) {
    unsigned long ones = ~0U / UCHAR_MAX;
    unsigned long highs = ones << (CHAR_BIT - 1);
    return (val - ones) & highs;
}
```

```
// compiled with -O0
push %rbp
mov %rsp,%rbp
mov %rdi,-0x18(%rbp)
movq $0x1010101,-0x8(%rbp)
mov -0x8(%rbp),%rax
shl $0x7,%rax
mov %rax,-0x10(%rbp)
mov -0x18(%rbp),%rax
mov %eax,%edx
mov -0x8(%rbp),%rax
sub %eax,%edx
mov -0x10(%rbp),%rax
and %edx,%eax
pop %rbp
retq
```

The compiler doesn't need to emit assembly for work that can be managed at CT. Here, it **folds** all constants into two instructions.

```
// compiled with -O2
lea -0x1010101(%rdi),%eax
and $0x80808080,%eax
retq
```

Common Sub-Expression Elimination

Common Sub-Expression Elimination prevents the recalculation of the same thing many times by doing it once and saving the result.

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

Common Sub-Expression Elimination

Common Sub-Expression Elimination prevents the recalculation of the same thing many times by doing it once and saving the result.

```
int a = (param2 + 0x107);
int b = param1 * (param2 + 0x107) + a;
return a * (param2 + 0x107) + b * (param2 + 0x107);
// = 2 * a * a + param1 * a * a
```

```
00000000000011b0 <subexp>:  // param1 in %edi, param2 in %esi
    11b0: lea    0x107(%rsi),%eax   // %eax stores a
    11b6: imul   %eax,%edi          // param1 * a
    11b9: lea    (%rdi,%rax,2),%esi // 2 * a + param1 * a
    11bc: imul   %esi,%eax          // a * (2 * a + param1 * a)
    11bf: retq
```

Common Sub-Expression Elimination

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 in every instance. Plus, it can minimize code replication.
- Makes code more readable.

Dead Code Elimination

Dead code elimination removes code that doesn't serve a purpose:

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

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

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

// If/else that more obliquely does the same thing in both cases
if (param1 == 0) {
    return 0;
} else {
    return param1;
}
```

Dead Code: Before (-O0)

```
00000000000011a9 <dead_code>:
    11a9:       55                      push   %rbp
    11aa:       48 89 e5                mov    %rsp,%rbp
    11ad:       48 83 ec 20             sub    $0x20,%rsp
    11b1:       89 7d ec                mov    %edi,-0x14(%rbp)
    11b4:       89 75 e8                mov    %esi,-0x18(%rbp)
    11b7:       8b 45 ec                mov    -0x14(%rbp),%eax
    11ba:       3b 45 e8                cmp    -0x18(%rbp),%eax
    11bd:       7d 1c                   jge    11db <dead_code+0x32>
    11bf:       8b 45 ec                mov    -0x14(%rbp),%eax
    11c2:       3b 45 e8                cmp    -0x18(%rbp),%eax
    11c5:       7e 14                   jle    11db <dead_code+0x32>
    11c7:       48 8d 05 36 0e 00 00    lea    0xe36(%rip),%rax
    11ce:       48 89 c7                mov    %rax,%rdi
    11d1:       b8 00 00 00 00          mov    $0x0,%eax
    11d6:       e8 65 fe ff ff          call   1040 <printf@plt>
    11db:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
    11e2:       eb 04                   jmp    11e8 <dead_code+0x3f>
    11e4:       83 45 fc 01             addl   $0x1,-0x4(%rbp)
    11e8:       81 7d fc e7 03 00 00    cmpl   $0x3e7,-0x4(%rbp)
    11ef:       7e f3                   jle    11e4 <dead_code+0x3b>
    11f1:       8b 45 ec                mov    -0x14(%rbp),%eax
    11f4:       3b 45 e8                cmp    -0x18(%rbp),%eax
    11f7:       75 06                   jne    11ff <dead_code+0x56>
    11f9:       83 45 ec 01             addl   $0x1,-0x14(%rbp)
    11fd:       eb 04                   jmp    1203 <dead_code+0x5a>
    11ff:       83 45 ec 01             addl   $0x1,-0x14(%rbp)
    1203:       83 7d ec 00             cmpl   $0x0,-0x14(%rbp)
    1207:       75 07                   jne    1210 <dead_code+0x67>
    1209:       b8 00 00 00 00          mov    $0x0,%eax
    120e:       eb 03                   jmp    1213 <dead_code+0x6a>
    1210:       8b 45 ec                mov    -0x14(%rbp),%eax
    1213:       c9                      leave
    1214:       c3                      ret
```

Dead Code: After (-O2)

```
0000000000011b0 <dead_code>:
    11b0:   8d 47 01                lea    0x1(%rdi),%eax
    11b3:   c3                      retq
```

Strength Reduction

Strength reduction changes divide to multiply, multiply to add/shift, and mod to and to avoid more expensive instructions, like multiply and divide.

```
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;
```

Code Motion

Code motion moves code outside of a loop if possible.

```
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.

Tail Recursion

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

```
unsigned long factorial(unsigned long n) {
  if (n <= 1) {
    return 1;
  }
  return n * factorial(n - 1);
}
```

Tail Recursion

Recall the factorial implementation from the code generation lecture:

```
unsigned long factorial(unsigned long n) {
  if (n <= 1) {
    return 1;
  }
  return n * factorial(n - 1);
}
```

What happens with **`factorial(-1)`**?

- Infinite recursion → Literal stack overflow!
- Compiled with `-Og`!

Factorial: -Og vs –O2

```
401146 <+0>:   cmp    $0x1,%edi
401149 <+3>:   jbe    0x40115b <factorial+21>
40114b <+5>:   push   %rbx
40114c <+6>:   mov    %edi,%ebx
40114e <+8>:   lea    -0x1(%rdi),%edi
401151 <+11>:  callq  0x401146 <factorial>
401156 <+16>:  imul   %ebx,%eax
401159 <+19>:  pop    %rbx
40115a <+20>:  retq
40115b <+21>:  mov    $0x1,%eax
401160 <+26>:  retq
```

-O2:
- What happened?
- Did the compiler fix the infinite recursion?

```
4011e0 <+0>:  mov    $0x1,%eax
4011e5 <+5>:  cmp    $0x1,%edi
4011e8 <+8>:  jbe    0x4011fd <factorial+29>
4011ea <+10>: nopw   0x0(%rax,%rax,1)
4011f0 <+16>: mov    %edi,%edx
4011f2 <+18>: sub    $0x1,%edi
4011f5 <+21>: imul   %edx,%eax
4011f8 <+24>: cmp    $0x1,%edi
4011fb <+27>: jne    0x4011f0 <factorial+16>
4011fd <+29>: retq
```

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.

```
for (size_t 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
```

Limitations of GCC Optimization

GCC can't always optimize everything! But you may know more than **`gcc`** does.

```
size_t char_sum(char *s) {
    size_t 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.

```
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? **`strlen` called for every character**

What can GCC do? **nothing! `s` is changing, so `gcc` doesn’t know if length is constant across iterations. We, however, do!**

Callgrind

- `callgrind` is another tool in the `valgrind` suite
- `callgrind` is a **profiler** that measures instruction counts – another way to measure efficiency
- can measure the number of instructions executed in each program run and where they came from
- useful for optimizing – we can see where large #s of instruction executions come from

Demo: `limitations.c` and callgrind

Why not always optimize?

Why not always just compile with –O2?

- Difficult to debug optimized executables, so only optimize when complete
- Optimizations may not truly optimize program execution. The compiler does its best, but it may slow things down, etc. Experiment to see what works out.