CS107, Lecture 25
Heap Allocators + Optimization

Reading: B&O 5

This document is copyright (C) Stanford Computer Science and Nick Troccoli, licensed under
Creative Commons Attribution 2.5 License. All rights reserved.
Based on slides created by Cynthia Lee, Chris Gregg, Jerry Cain, Lisa Yan and others.
NOTICE RE UPLOADING TO WEBSITES: This content is protected and may not be shared,
uploaded, or distributed. (without expressed written permission)

CS107 Topic 6: How do the core `malloc/realloc/free` memory-allocation operations work?

Learning Goals

- Understand how we can optimize our code to improve efficiency and speed
- Learn about the optimizations compilers (we use GCC) can perform

Compiler Optimization

Compilers are wicked smart at making optimizations to our code during compilation! It's important to know about what the compiler can/will optimize for several reasons:

1. Shows us the power of compilers and how they try to understand the intended behavior of our code
2. Demonstrates how compiler optimization doesn't *always* make our code better (makes it harder to debug, may not always work well), and how it can have limitations.
3. Shows how compiler optimizations can let us focus on writing readable code, and the compiler can do some optimizations for us.

Lecture Plan

- Additional heap allocator tidbits
- What is optimization?
- GCC Optimization
- Limitations of GCC Optimization

```
cp -r /afs/ir/class/cs107/lecture-code/lect25 .
```

Lecture Plan

- **Additional heap allocator tidbits**
- What is optimization?
- GCC Optimization
- Limitations of GCC Optimization

```
cp -r /afs/ir/class/cs107/lecture-code/lect25 .
```

Accessing Heap Memory

When manipulating heap contents (e.g., headers, free list pointers), we must use pointers and dereferencing.

```
void *pointer_to_heap = …;
*(int *)pointer_to_heap = …;
```

Use appropriate types for pointers – **void *** if pointee type unknown or can vary, specific type if pointee type is known.

```
// no need for casting, more type checking, clearer
short *pointer_to_short …;
*pointer_to_short = …;
```

Accessing Heap Memory

Consider types of what you are trying to access or modify:

```
void *pointer_to_heap = …;
*(void **)pointer_to_heap = …;  // set void *
*(short *)pointer_to_heap = …;  // set short
```

Using Structs

When accessing adjacent fields, model them with structs:

```
typedef struct mystruct {
    int x;
    char *str;
} mystruct;

mystruct *pointer_to_fields = …;
pointer_to_fields->x = …;
pointer_to_fields->str = …;

// without structs, more verbose and difficult to access:
void *pointer_to_fields = …;
*(int *)pointer_to_fields = …;
*(char **)((char *)pointer_to_fields + sizeof(int)) = …;
```

Lecture Plan

- Additional heap allocator tidbits
- **What is optimization?**
- GCC Optimization
- Limitations of GCC Optimization

```
cp -r /afs/ir/class/cs107/lecture-code/lect25 .
```

Optimization

Optimization is the task of making your program faster or more efficient with space or time. There are many ways to do this, including:

1. Make modifications to our code ourselves in the name of optimization – e.g. making our core algorithms faster (e.g. Big-O runtimes) or making other modifications
2. Having the compiler (we use "GCC") optimize our code in certain ways

For #1, we don't want to over-optimize; this can result in code that is more cumbersome and complicated to read and debug. Optimize only where necessary to remove reasonable bottlenecks.

Optimization

We could make our program more efficient in many ways, including:

- **Speed** (e.g. wall clock time, or number of assembly instructions executed, or "CPU cycle count" (# CPU operations performed))
- **Memory usage**
- **Executable size** (how large compiled program is, or, as proxy, number of assembly instructions outputted – smaller means e.g. faster for users to download)

Key terms:

- **static instruction count** (# of written instructions in executable).
- **dynamic instruction count** (# of executed instructions when program is run).

Optimization

Our approach to optimization is a combination of manual optimizations and the compiler optimizing for us:

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

Lecture Plan

- Additional heap allocator tidbits
- What is optimization?
- **GCC Optimization**
- Limitations of GCC Optimization

```
cp -r /afs/ir/class/cs107/lecture-code/lect25 .
```

The compiler can compile our code in different ways to make it faster / more efficient with space or time!

GCC Optimization

When compiling, you can add an "optimization level flag" that tells the compiler how much you want it to optimize your code. (source)

- **-O0** means "mostly just literal translation of C" (there's also -Og, meaning "more debugging-friendly")
- **-O2** means "enable nearly all reasonable optimizations"

How high can you crank the optimization?

- **-O3** means "more aggressive than O2, trade size for speed"
- **-Os** means "optimize for size"
- **-Ofast** means "disregard standards compliance" (!!)

Compiler optimizations

How many GCC optimization levels are there?

Asked 11 years, 3 months ago Active 5 months ago Viewed 62k times

109

35

How many GCC optimization levels are there?

I tried gcc -O1, gcc -O2, gcc -O3, and gcc -O4

If I use a really large number, it won't work.

However, I have tried

```
gcc -O100
```

and it compiled.

How many optimization levels are there?

Gcc supports numbers up to 3. Anything above is interpreted as 3

https://stackoverflow.com/questions/1778538/how-many-gcc-optimization-levels-are-there

Example: Matrix Multiplication

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

```
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   1.32M
matrix multiply  50^2: cycles  10.64M
matrix multiply 100^2: cycles  16.55M
```

```
./mult_opt      // -O2 (with optimization)
matrix multiply  25^2: cycles   0.33M (opt)
matrix multiply  50^2: cycles   2.04M (opt)
matrix multiply 100^2: cycles  13.60M (opt)
```

Wait a minute. The compiler can optimize my code *automatically*? Why wouldn’t I always want to do that?

GCC Optimization

Why not always turn on optimizations?

- Some trial and error sometimes required to examine benefits vs. drawbacks.
- Optimized code can be harder to debug and can take longer to compile.

Note: the compiler may do some optimizations even without you explicitly specifying e.g. –O2.

Goal for today: show you examples of what the compiler can (and cannot) optimize, and how we may need to fill in the gaps.

- **First**, let's go inside the mind of a compiler to see how it might think about our code and how to optimize it
- **Afterwards,** we'll go through a list of several different kinds of optimizations GCC might do when compiling our code.

Common For Loop Construction

C For loop

```
for (init; test; update) {
    body
}
```

C Equivalent While Loop

```
init
while(test) {
    body
    update
}
```

Assembly pseudocode

```
Init
Check opposite of code condition
Skip loop if test passes
Body
```

```
Update
Jump back to test
```

For loops and while loops are treated (essentially) the same when compiled down to assembly.

GCC For Loop Output

GCC Possible For Loop Output (#1)

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

Other Possible GCC For Loop Output (#2)

```
Initialization
Jump to test
Body
Update
Test
Jump to body if success
```

GCC For Loop Output

GCC Possible For Loop Output (#1)

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

```
for (int i = 0; i < n; i++)            // n = 100
```

GCC For Loop Output

GCC Possible For Loop Output (#1)

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

for (int i = 0; i < n; i++) // n = 100

```
Initialization
Test
No jump
Body
Update
Jump to test
Test
No jump
Body
Update
Jump to test
...
```

GCC For Loop Output

GCC Possible For Loop Output (#1)

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

for (int i = 0; i < n; i++)   // n = 100

```
Initialization
Test
No jump
Body
Update
Jump to test
Test
No jump
Body
Update
Jump to test
...
```

GCC For Loop Output

```
for (int i = 0; i < n; i++)          // n = 100

Initialization
Jump to test
Test
Jump to body
Body
Update
Test
Jump to body
Body
Update
Test
Jump to body
...
```

Other Possible GCC For Loop Output (#2)

```
Initialization
Jump to test
Body
Update
Test
Jump to body if success
```

GCC For Loop Output

```
for (int i = 0; i < n; i++)          // n = 100

Initialization
Jump to test
Test
Jump to body
Body
Update
Test
Jump to body
Body
Update
Test
Jump to body
...
```

Other Possible GCC For Loop Output (#2)

```
Initialization
Jump to test
Body
Update
Test
Jump to body if success
```

GCC For Loop Output

GCC Possible For Loop Output (#1)

```
Initialization
Test
Jump past loop if passes
Body
Update
Jump to test
```

Other Possible GCC For Loop Output (#2)

```
Initialization
Jump to test
Body
Update
Test
Jump to body if success
```

Which instructions are better when n = 0? n = 1000?

for (int i = 0; i < n; i++)

Optimizing Instruction Counts

- Both versions have the same **static instruction count** (# of written instructions).
- But they have different **dynamic instruction counts** (# of executed instructions when program is run).
  - If n = 0, #1 is best b/c fewer instructions
  - If n is large, #2 is best b/c fewer instructions

GCC Possible For Loop Output (#1)

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

Other Possible GCC For Loop Output (#2)

```
Initialization
Jump to test
Body
Update
Test
Jump to body if success
```

Optimizing Instruction Counts

- Both versions have the same **static instruction count** (# of written instructions).
- But they have different **dynamic instruction counts** (# of executed instructions when program is run).
  - If n = 0, #1 is best b/c fewer instructions
  - If n is large, #2 is best b/c fewer instructions
- The compiler may emit a static instruction count that is several times longer than an alternative, but it may be more efficient if loop executes many times.
- Does the compiler *know* that a loop will execute many times? (in general, no)
- So what if our code had loops that always execute a small number of times? How do we know when gcc makes a bad decision?
- (take EE108, EE180, CS316 for more!)

GCC Optimizations

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

GCC Optimizations

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

(kidding)

GCC Optimizations

- **<u>Constant Folding</u>**
- 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.

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

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)

```
00000000000011b9 <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                       ret
```

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?

GCC Optimizations

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

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 subexp(int param1, int param2) {
    int a = (param2 + 0x107);
    int b = param1 * (param2 + 0x107) + a;
    return a * (param2 + 0x107) + b * (param2 + 0x107);
}
```

Common Sub-Exp. Elim.: Before (-O0)

```
0000000000001199 <subexp>:
    1199:   55                      push   %rbp
    119a:   48 89 e5                mov    %rsp,%rbp
    119d:   89 7d ec                mov    %edi,-0x14(%rbp)
    11a0:   89 75 e8                mov    %esi,-0x18(%rbp)
    11a3:   8b 45 e8                mov    -0x18(%rbp),%eax
    11a6:   05 07 01 00 00          add    $0x107,%eax
    11ab:   89 45 fc                mov    %eax,-0x4(%rbp)
    11ae:   8b 45 e8                mov    -0x18(%rbp),%eax
    11b1:   05 07 01 00 00          add    $0x107,%eax
    11b6:   0f af 45 ec             imul   -0x14(%rbp),%eax
    11ba:   89 c2                   mov    %eax,%edx
    11bc:   8b 45 fc                mov    -0x4(%rbp),%eax
    11bf:   01 d0                   add    %edx,%eax
    11c1:   89 45 f8                mov    %eax,-0x8(%rbp)
    11c4:   8b 45 e8                mov    -0x18(%rbp),%eax
    11c7:   05 07 01 00 00          add    $0x107,%eax
    11cc:   0f af 45 fc             imul   -0x4(%rbp),%eax
    11d0:   89 c2                   mov    %eax,%edx
    11d2:   8b 45 e8                mov    -0x18(%rbp),%eax
    11d5:   05 07 01 00 00          add    $0x107,%eax
    11da:   0f af 45 f8             imul   -0x8(%rbp),%eax
    11de:   01 d0                   add    %edx,%eax
    11e0:   5d                      pop    %rbp
    11e1:   c3                      ret
```

Common Sub-Exp. Elim.: After (-O2)

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

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

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 every instance. Plus, it can help reduce redundancy!
- Makes code more readable!

GCC Optimizations

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

Dead Code

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

```
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)

```
0000000000011a9 <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        # 2004 <_IO_stdin_used+0x4>
    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                       ret
```

GCC Optimizations

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

Strength Reduction

Strength reduction (done even on –O0) changes divide to multiply, multiply to add/shift, and mod to AND to avoid using instructions that cost many cycles (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;
```

GCC Optimizations

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

Code Motion

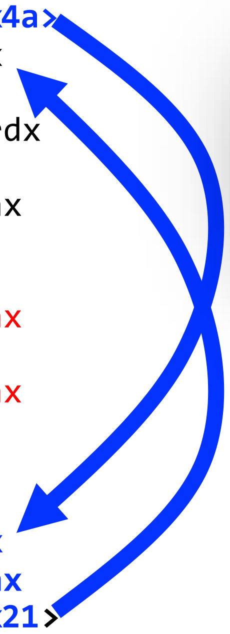
Code motion moves code outside of a loop if possible.

```
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.

Code Motion: Before (-O0)

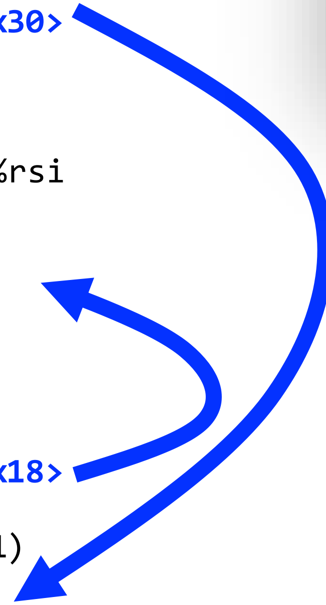
```
0000000000011b9 <motion>:
    11b9:   55                      push   %rbp
    11ba:   48 89 e5                mov    %rsp,%rbp
    11bd:   89 7d ec                mov    %edi,-0x14(%rbp)
    11c0:   89 75 e8                mov    %esi,-0x18(%rbp)
    11c3:   48 89 55 e0             mov    %rdx,-0x20(%rbp)
    11c7:   89 4d dc                mov    %ecx,-0x24(%rbp)
    11ca:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
    11d1:   c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%rbp)
    11d8:   eb 29                   jmp    1203 <motion+0x4a>
    11da:   8b 45 f8                mov    -0x8(%rbp),%eax
    11dd:   48 98                   cltq
    11df:   48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
    11e6:   00
    11e7:   48 8b 45 e0             mov    -0x20(%rbp),%rax
    11eb:   48 01 d0                add    %rdx,%rax
    11ee:   8b 10                   mov    (%rax),%edx
    11f0:   8b 45 e8                mov    -0x18(%rbp),%eax
    11f3:   83 c0 03                add    $0x3,%eax
    11f6:   0f af 45 ec             imul   -0x14(%rbp),%eax
    11fa:   01 d0                   add    %edx,%eax
    11fc:   01 45 fc                add    %eax,-0x4(%rbp)
    11ff:   83 45 f8 01             addl   $0x1,-0x8(%rbp)
    1203:   8b 45 f8                mov    -0x8(%rbp),%eax
    1206:   3b 45 dc                cmp    -0x24(%rbp),%eax
    1209:   7c cf                   jl     11da <motion+0x21>
    120b:   8b 45 fc                mov    -0x4(%rbp),%eax
    120e:   5d                      pop    %rbp
    120f:   c3                      ret
```



Here, foo * (bar + 3) is computed each time through the loop body.

Code Motion: After (-O2)

```
0000000000011f0 <motion>:
    11f0:   89 f8                   mov    %edi,%eax
    11f2:   85 c9                   test   %ecx,%ecx
    11f4:   7e 2a                   jle    1220 <motion+0x30>
    11f6:   8d 7e 03                lea    0x3(%rsi),%edi
    11f9:   48 63 c9                movslq %ecx,%rcx
    11fc:   0f af f8                imul   %eax,%edi
    11ff:   48 8d 34 8a             lea    (%rdx,%rcx,4),%rsi
    1203:   31 c0                   xor    %eax,%eax
    1205:   0f 1f 00                nopl   (%rax)
    1208:   8b 0a                   mov    (%rdx),%ecx
    120a:   48 83 c2 04             add    $0x4,%rdx
    120e:   01 f9                   add    %edi,%ecx
    1210:   01 c8                   add    %ecx,%eax
    1212:   48 39 f2                cmp    %rsi,%rdx
    1215:   75 f1                   jne    1208 <motion+0x18>
    1217:   c3                      ret
    1218:   0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
    121f:   00
    1220:   31 c0                   xor    %eax,%eax
    1222:   c3                      ret
    1223:   66 66 2e 0f 1f 84 00    data16 cs nopw 0x0(%rax,%rax,1)
    122a:   00 00 00 00
    122e:   66 90                   xchg   %ax,%ax
```



Here, foo * (bar + 3) is computed just once before the loop!

Practice: GCC Optimization

```
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? What (if anything) can GCC do?

Respond on PollEv:
pollev.com/cs107

L25. What is the bottleneck? What can GCC do?

strlen called every loop iteration - common subexpression elimination to eliminate redundancy

0%

strlen called every loop iteration - code motion to pull out of loop

0%

strlen called every loop iteration - GCC can't optimize

0%

i incremented every loop iteration - code motion to pull out of loop

0%

i incremented every loop iteration - GCC can't optimize

0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Practice: GCC Optimization

```
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? What (if anything) can GCC do?

strlen is called every loop iteration – code motion can pull it out of the loop

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.

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

Tail Recursion Comparison

Factorial at –O0

```
0000000000011c9 <factorial>:
    11c9: push   %rbp
    11ca: mov    %rsp,%rbp
    11cd: sub    $0x10,%rsp
    11d1: mov    %edi,-0x4(%rbp)
    11d4: cmpl   $0x1,-0x4(%rbp)
    11d8: ja     11e1 <factorial+0x18>
    11da: mov    $0x1,%eax
    11df: jmp    11f2 <factorial+0x29>
    11e1: mov    -0x4(%rbp),%eax
    11e4: sub    $0x1,%eax
    11e7: mov    %eax,%edi
    11e9: call   11c9 <factorial>
    11ee: imul   -0x4(%rbp),%eax
    11f2: leave
    11f3: ret
```

Factorial at –O2

```
0000000000001250 <factorial>:
    1250:   mov    $0x1,%eax
    1255:   cmp    $0x1,%edi
    1258:   jbe    126d <factorial+0x1d>
    125a:   nopw   0x0(%rax,%rax,1)
    1260:   mov    %edi,%edx
    1262:   sub    $0x1,%edi
    1265:   imul   %edx,%eax
    1268:   cmp    $0x1,%edi
    126b:   jne    1260 <factorial+0x10>
    126d:   ret
```

Tail Recursion

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

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

What happens differently with **`factorial(-1)`** in the -O2 version?

https://web.stanford.edu/class/cs107/lab6/extra.html

GCC Optimizations

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

Loop Unrolling

Loop Unrolling: Do **n** (e.g. 4 here) loop iterations' worth of work per actual loop iteration, so we save ourselves from doing the loop overhead (e.g. test and jump) every time, and instead incur overhead only every n-th time.

```
for (int i = 0; i < length; i++) {
    sum += arr[i];
}

// loop unrolling:
for (int i = 0; i < length; 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

- Additional heap allocator tidbits
- What is optimization?
- GCC Optimization
- **Limitations of GCC Optimization**

```
cp -r /afs/ir/class/cs107/lecture-code/lect25 .
```

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? **Can't pull strlen out of the loop. s is changing, so GCC doesn't know if length is constant across iterations. But *<u>we</u>* know its length doesn't change.**

Callgrind

- Callgrind is another tool in the Valgrind suite of tools
- Callgrind is a **profiler** that measures instruction counts – one way to measure efficiency
- Can measure the number of instructions executed in a given run of our program, and where they came from
- Useful for optimizing – we can see where large #s of instruction executions come from

Demo: `limitations.c` and callgrind

```
valgrind –tool=callgrind ./program args

callgrind_annotate –auto=yes outputfile
```

GCC Optimization

Compilers are wicked smart at making optimizations to our code during compilation! It's important to know about what the compiler can/will optimize for several reasons:

1. Shows us the power of compilers and how they try to understand the intended behavior of our code
2. Demonstrates how compiler optimization doesn't *always* make our code better (makes it harder to debug, may not always work well), and how it can have limitations.
3. Shows how compiler optimizations can let us focus on writing readable code, and the compiler can do some optimizations for us.

Why not always just compile with –O2? More difficult to debug (optimize at end), compiling may take longer, optimizations may not always improve your program. Experiment!

Recap

- Additional heap allocator tidbits
- What is optimization?
- GCC Optimization
- Limitations of GCC Optimization

Lecture 25 takeaway: Compilers can apply various optimizations to make our code more efficient, without us having to rewrite code. However, there are limitations to these optimizations, and sometimes we must optimize ourselves, using tools like Callgrind.

Next time: wrap up