CS107, Lecture 15
Optimization

Reading: B&O 5

<u>CS107 Topic 6</u>: 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 GCC can perform

Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect15 .
```

Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect15 .
```

Optimization

- Optimization is the task of making your program faster or more efficient with space or time. You've seen 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


```
cp -r /afs/ir/class/cs107/lecture-code/lect15 .
```

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:

```
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
- 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

Optimizations may target one or more of:

- Static instruction count
- Dynamic instruction count
- Cycle count / execution time

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;
```

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>:
  400626:       55                      push   %rbp
  400627:       53                      push   %rbx
  400628:       48 83 ec 08             sub    $0x8,%rsp
  40062c:       89 fd                   mov    %edi,%ebp
  40062e:       f2 0f 10 05 da 00 00    movsd  0xda(%rip),%xmm0
  400635:       00
  400636:       e8 d5 fe ff ff          callq  400510 <sqrt@plt>
  40063b:       f2 0f 2c c8             cvttsd2si %xmm0,%ecx
  40063f:       69 ed 07 01 00 00       imul   $0x107,%ebp,%ebp
  400645:       b8 15 00 00 00          mov    $0x15,%eax
  40064a:       99                      cltd
  40064b:       f7 f9                   idiv   %ecx
  40064d:       8d 98 07 01 00 00       lea    0x107(%rax),%ebx
  400653:       bf 04 07 40 00          mov    $0x400704,%edi
  400658:       e8 93 fe ff ff          callq  4004f0 <strlen@plt>
  40065d:       48 69 c0 23 05 00 00    imul   $0x523,%rax,%rax
  400664:       48 63 db                movslq %ebx,%rbx
  400667:       48 8d 44 18 c9          lea    -0x37(%rax,%rbx,1),%rax
  40066c:       48 c1 e8 02             shr    $0x2,%rax
  400670:       01 e8                   add    %ebp,%eax
  400672:       48 83 c4 08             add    $0x8,%rsp
  400676:       5b                      pop    %rbx
  400677:       5d                      pop    %rbp
  400678:       c3                      retq
```

Constant Folding: After (-O2)

```
00000000004004f0 <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

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.

This optimization is done even at –O0!

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

```
00000000004004f0 <subexp>:
  4004f0:  81 c6 07 01 00 00      add    $0x107,%esi
  4004f6:  0f af fe               imul   %esi,%edi
  4004f9:  8d 04 77               lea    (%rdi,%rsi,2),%eax
  4004fc:  0f af c6               imul   %esi,%eax
  4004ff:  c3                     retq
```

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:

```
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          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
  4004fe: 66 90                 xchg   %ax,%ax
```

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).

```
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.

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.

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

You saw this in the last lab!

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.

```
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


```
cp -r /afs/ir/class/cs107/lecture-code/lect15 .
```

Limitations of GCC Optimization

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

```
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.

```
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. But *we* know its length doesn't change.**

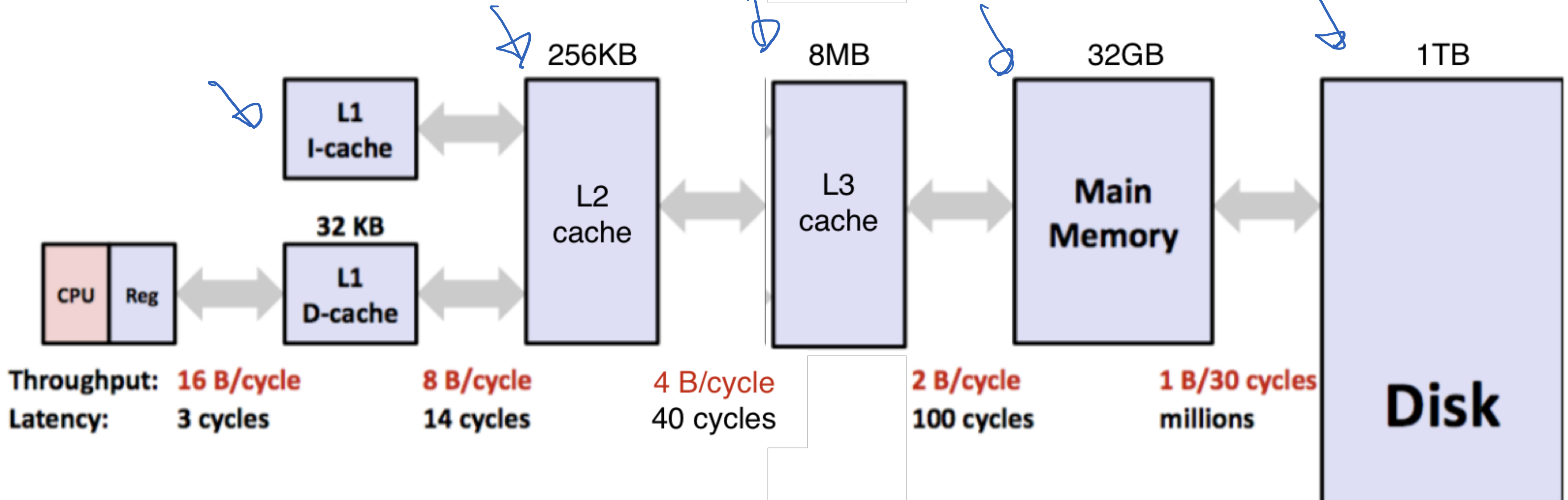
Demo: `limitations.c`

Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect15 .
```

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

Caching

All caching depends on locality.

Realistic scenario:

- 97% cache hit rate
- Cache hit costs 1 cycle
- Cache miss costs 100 cycles
- How much of your memory access time is spent on 3% of accesses that are cache misses?

Demo: `cache.c`

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…

Recap

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

Next time: wrap up

Live Session Slides

Post any questions you have to today's lecture thread on the discussion forum!

Plan For Today

- **10 minutes:** general review
- **5 minutes:** post questions or comments on Ed for what we should discuss
- **15 minutes**: open Q&A
- **25 minutes**: extra practice

Lecture 15 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.

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

Use -Og (items 1–2)

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

Questions you may have

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!
- Humans read your code too—not just computers ☺

Plan For Today

- **10 minutes:** general review
- **5 minutes:** post questions or comments on Ed for what we should discuss
- **15 minutes**: open Q&A
- **25 minutes**: extra practice

Lecture 15 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.

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.

This optimization is *not* done at –O0.

```
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
```

Tail recursion example: Lab6 bonus

Recall the factorial problem from Lecture 13:

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

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

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

- Infinite recursion → Literal stack overflow!
- Compiled with `-0g`!

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
```