

CS107, Lecture 15

Optimization

Reading: B&O 5

**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 GCC can perform

Lecture Plan

- What is optimization? 5
- GCC Optimization 8
- Limitations of GCC Optimization 35
- Caching 40
- Live Session Slides 47

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

Lecture Plan

- **What is optimization?** 5
- GCC Optimization 8
- Limitations of GCC Optimization 35
- Caching 40
- Live Session Slides 47

```
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

- What is optimization? 5
- **GCC Optimization** 8
- Limitations of GCC Optimization 35
- Caching 40
- Live Session Slides 47

```
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 also use `-Og`, like `-O0` but more debugging 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 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 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)
```

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

- **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.

```
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 (-00)

```
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 callq  10a0 <sqrt@plt>
11f0: f2 0f 2c c0 cvtsd2si %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         cld
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 3d ed 0d 00 00 lea   0xded(%rip),%rdi      # 2008 <_IO_stdin_used+0x8>
121b: e8 20 fe ff ff callq  1040 <strlen@plt>
1220: 8b 55 e8    mov    -0x18(%rbp),%edx
1223: 48 63 d2    movslq %edx,%rdx
1226: 48 0f af c2 imul  %rdx,%rax
122a: 48 01 d8    add   %rbx,%rax
122d: 48 83 e8 37 sub    $0x37,%rax
1231: 48 c1 e8 02 shr    $0x2,%rax
1235: 44 01 e0    add   %r12d,%eax
1238: 48 83 c4 30 add    $0x30,%rsp
123c: 5b         pop   %rbx
123d: 41 5c      pop   %r12
123f: 5d         pop   %rbp
1240: c3         retq
```

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?

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.

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

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

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 (-00)

```
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 19      jge    11d8 <dead_code+0x2f>
 11bf: 8b 45 ec    mov    -0x14(%rbp),%eax
 11c2: 3b 45 e8    cmp    -0x18(%rbp),%eax
 11c5: 7e 11      jle    11d8 <dead_code+0x2f>
 11c7: 48 8d 3d 36 0e 00 00 lea    0xe36(%rip),%rdi          # 2004 <_IO_stdin_used+0x4>
 11ce: b8 00 00 00 00 mov    $0x0,%eax
 11d3: e8 68 fe ff ff callq  1040 <printf@plt>
 11d8: c7 45 fc 00 00 00 00 movl   $0x0,-0x4(%rbp)
 11df: eb 04      jmp    11e5 <dead_code+0x3c>
 11e1: 83 45 fc 01 addl   $0x1,-0x4(%rbp)
 11e5: 81 7d fc e7 03 00 00 cmpl   $0x3e7,-0x4(%rbp)
 11ec: 7e f3      jle    11e1 <dead_code+0x38>
 11ee: 8b 45 ec    mov    -0x14(%rbp),%eax
 11f1: 3b 45 e8    cmp    -0x18(%rbp),%eax
 11f4: 75 06      jne    11fc <dead_code+0x53>
 11f6: 83 45 ec 01 addl   $0x1,-0x14(%rbp)
 11fa: eb 04      jmp    1200 <dead_code+0x57>
 11fc: 83 45 ec 01 addl   $0x1,-0x14(%rbp)
 1200: 83 7d ec 00 cmpl   $0x0,-0x14(%rbp)
 1204: 75 07      jne    120d <dead_code+0x64>
 1206: b8 00 00 00 00 mov    $0x0,%eax
 120b: eb 03      jmp    1210 <dead_code+0x67>
 120d: 8b 45 ec    mov    -0x14(%rbp),%eax
 1210: c9        leaveq
 1211: c3        retq
```

Dead Code: After (-02)

00000000000011b0 <dead_code>:

11b0: 8d 47 01

11b3: c3

lea 0x1(%rdi),%eax

retq

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);  
}
```


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

- What is optimization? 5
- GCC Optimization 8
- **Limitations of GCC Optimization** 35
- Caching 40
- Live Session Slides 47

```
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?
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.

Demo: limitations.c



Why not always optimize?

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!

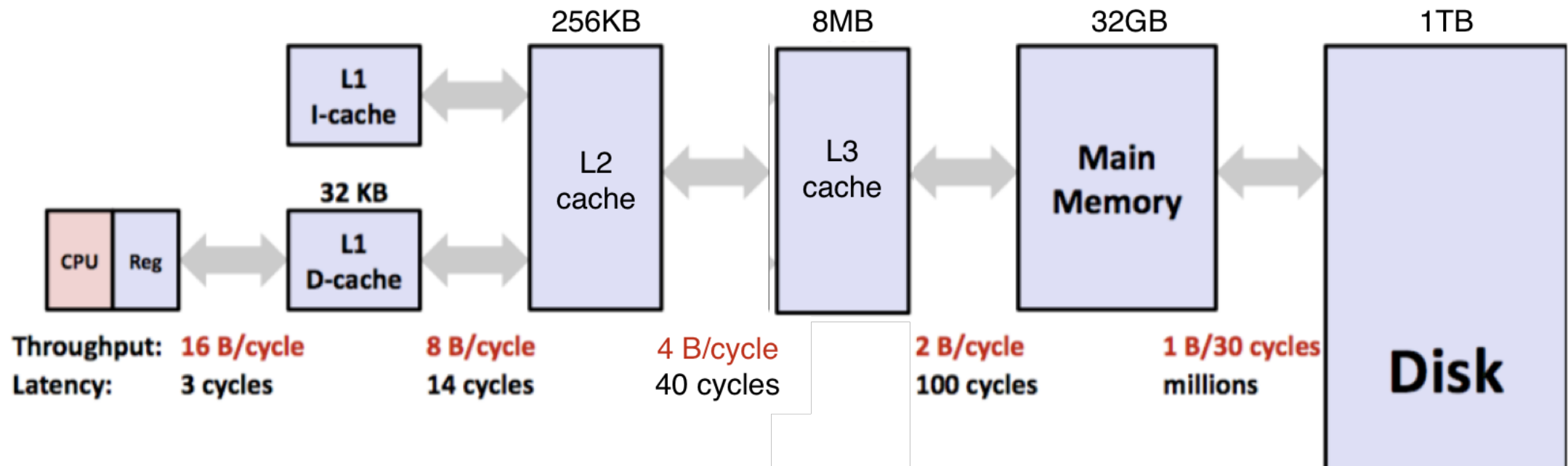
Lecture Plan

- What is optimization? 5
- GCC Optimization 8
- Limitations of GCC Optimization 35
- **Caching** 40
- Live Session Slides 47

```
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

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



Don't
use e.g.
-O2

Compiler optimizations

How many GCC optimization levels are there?

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



How many [GCC](#) optimization levels are there?

109

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

35



```
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>

Plan For Today

- **10 minutes:** general review
- **5 minutes:** post questions or comments on Ed for what we should discuss

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.

```
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 `-Og`!

Factorial: -0g vs -02



```
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
```

-02:

- 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
```