

Brian Barbu (brb9da)

In-Lab 9

Optimized Code:

To analyze differences between optimized code made in the compiler using the -O2 versus regular code, I created a simple program and examined how their assembly files differed. Surprisingly, my first attempt at writing a C++ program did not result in a simpler assembly file when compiled with the -O2 flag. I attempted this using clang++ -mllvm --x86-asm-syntax=intel -m64 -S -O2 test1.cpp, but did not get any good results. I rewrote my program as a C program and was able to see definitive results. I compiled the code with the optimization flag using gcc -S -O2 test1.c.

The C Program:

```
#include <stdio.h>

int main(){
    int num;
    printf( "Enter a number: ");
    scanf("%d", &num);
    int result = num;
    while(num>0){
        result = result * num;
        num--;
    }
    if( result % 2 == 0){
        result = result * 2;
    }
    else{
        result = result * 3;
    }
    return result;
}
```

There were 72 lines of assembly code in the non-optimized version of my program. However, the optimized version produced a much simpler 56 line file of assembly code.

Non-optimized Assembly:


```

.file "test1.c"
.text
.section .rodata.str1.1,"aMS",@progbits,1
.LC0:
.string "Enter a number: "
.LC1:
.string "%d"
.section .text.startup,"ax",@progbits
.p2align 4,,15
.globl main
.type main, @function
main:
.LFB23:
.cfi_startproc
subq $24, %rsp
.cfi_def_cfa_offset 32
leaq .LC0(%rip), %rsi
movl $1, %edi
movq %fs:40, %rax
movq %rax, 8(%rsp)
xorl %eax, %eax
call __printf_chk@PLT
leaq 4(%rsp), %rsi
leaq .LC1(%rip), %rdi
xorl %eax, %eax
call __isoc99_scanf@PLT
movl 4(%rsp), %edx
testl %edx, %edx
jle .L2
movl %edx, %eax
.p2align 4,,10
.p2align 3
.L3:
imull %eax, %edx
subl $1, %eax
jne .L3
.L2:
leal (%rdx,%rdx), %ecx
leal (%rdx,%rdx,2), %eax
andl $1, %edx
cmovl %ecx, %eax
movq 8(%rsp), %rcx
xorq %fs:40, %rcx
jne .L10
addq $24, %rsp
.cfi_remember_state
.L2:
leal (%rdx,%rdx), %ecx
leal (%rdx,%rdx,2), %eax
andl $1, %edx
cmovl %ecx, %eax
movq 8(%rsp), %rcx
xorq %fs:40, %rcx
jne .L10
addq $24, %rsp
.cfi_remember_state
.L10:
.cfi_restore_state
call __stack_chk_fail@PLT
.cfi_endproc
.LFE23:
.size main, .-main
.ident "GCC: (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0"
.section .note.GNU-stack,"",@progbits

```

When the -O2 flag is used, the compiler will try to minimize the instructions needed to run out the output. This means that instructions that do not directly influence the return value will often be omitted in the -O2 optimized version. One example of this is with the use of scanf. Scanf will have a return value, but it will be unused in creating the output and is thus omitted in the optimized assembly version. Another thing I noticed is that there were less loops used in the optimized version in comparison with the regular. Extraneous loops can be very detrimental for runtime as many iterations of useless instructions may be executed and slowing down the program. The optimized versions only executes the critical tasks. Another way -O2 is optimizing this program is by saving significant data early. Instructions are repositioned to make important data available in memory early and removes the need for certain mov load instructions that are called late in the original assembly and would slow runtime. Another example of how -O2 optimizes the code is by ignoring the base pointer. Typical assembly would push the pointer and allocate room on the stack, but this is significant memory usage and is not important to the simpler tasks the program is intended to complete. -O2 optimization then chooses to ignore managing the base pointer. Optimization in compilers using the -O2 flag is greatly effective in eliminating unnecessary instructions and generating the correct output in the shortest runtime.