Jiwon Cha
jc4va

In-lab report

Topic

Optimized code: Compare code generated normally to optimized code. To create optimized code, you will need to use the -O2 compiler flag. Can you make any guesses as to why the optimized code looks as it does? What is being optimized? Be sure to show your original sample code as well as the optimized version. Try loops and function calls to see what "optimizing" does. Be aware that if instructions are "not necessary" to the final output of the program then they may be optimized away completely! This does not lead to very interesting comparisons. Describe at least four (non-trivial) differences you see between 'normal' code and optimized code.

Code 1

C++ code:

- int foo(int a){return a+3};
- int main(){foo(3)};

Assembly code – without –O2 flag:

```
mov     dword ptr [rbp - 4], edi
mov     edi, dword ptr [rbp - 4]
add     edi, 3
mov     eax, edi
pop     rbp
ret
```

```
sub     rsp, 16
mov     edi, 3
call    __Z3fooi
xor     edi, edi
mov     dword ptr [rbp - 4], eax ## 4-byte Spill
mov     eax, edi
add     rsp, 16
pop     rbp
ret
.cfi_endproc
```

Assembly code – with –O2 flag:

```
lea     eax, [rdi + 3]
pop     rbp
ret
```

```
.cfi_def_cfa_register rbp
xor     eax, eax
pop     rbp
ret
```

Foo function:

First of all, it is readily apparent that the optimized version is much shorter than the original assembly code. The optimized version ignores several unnecessary steps such as moving the element in edi to a spot on the stack and then moving the element in that spot into edi again. This step is not necessary within the context of this code, so the assembly takes it out. It also uses lea instruction instead of mov, which makes sense because lea can add and load at the same time while mov requires separate instruction for load and add. For example, if you want to add several constants to the register, lea is much shorter and faster. The code above shows how several steps can be shortened to one instruction; setting up edi, add edi, 3; mov eax, edi → lea eax, [rdi+3]

Main method:

Main method is also very short compared to the version without –O2 flag. First of all, the original main method manually adds 3 to the first parameter and then moves that value to eax. It also performs additional operations such as moving rsp to the original spot. However, the optimized version does not need to perform this step because 3 was already added to the first parameter inside of foo function. Therefore, in the main method, it simply returns what is in rax by calling ret.

<u>Code 2</u>

Assembly code for this program only includes the function part (foo) as the main function is the same as code 1.

C++ code:

```
int foo(int a){
  for (int i = 0; i < 5; i++){
    a += i;
  }
  return a;
}

int main(){
  foo(3);
}
```

Assembly code – without –O2 flag:

```
Ltmp2:
        .cfi_def_cfa_register rbp
        mov     dword ptr [rbp - 4], 0
        mov     dword ptr [rbp - 12], 0
LBB0_1:                                 ## =>This Inner Loop Header: Depth=1
        cmp     dword ptr [rbp - 12], 5
        jge     LBB0_4
## BB#2:                                ##   in Loop: Header=BB0_1 Depth=1
        mov     eax, dword ptr [rbp - 12]
        add     eax, dword ptr [rbp - 8]
        mov     dword ptr [rbp - 8], eax
## BB#3:                                ##   in Loop: Header=BB0_1 Depth=1
        mov     eax, dword ptr [rbp - 12]
        add     eax, 1
        mov     dword ptr [rbp - 12], eax
        jmp     LBB0_1
LBB0_4:
        mov     eax, dword ptr [rbp - 4]
        pop     rbp
        ret
        .cfi_endproc
```

Assembly code – with –O2 flag:

```
.cfi_def_cfa_register rbp

lea     eax, [rdi + 10]
pop     rbp
ret
.cfi_endproc
```

The original version of assembly carries out each step of the loop. We see that arguments of the for loop (such as int i) are stored on the stack as local variables. int i starts from 0 and increments until it reaches 5, therefore it first moves 0 to a spot on the stack and then compares the value stored in that spot to 5. If it is equal to 5, it jumps to the last part of loop and returns the value in eax. Otherwise, it keeps adding the value of i to eax and then incrementing the value in spot [rbp-12].

However, the optimized version skips most of the steps described above. The function simply adds 10 to the parameter and moves that value to eax, which means it has already calculated the result of 0+1+2+3+4. This is very convenient because the method is simply adding 0, 1, 2, 3, and 4 to the parameter.

To conclude, the optimized version seems to skip many steps unlike the original versions, and sometimes take parameters passed in the main function directly in the callee (foo function); it carries out operations that are usually performed in the main method, inside of the callee.