Gabriel Groover

Inlab.pdf

Optimizations

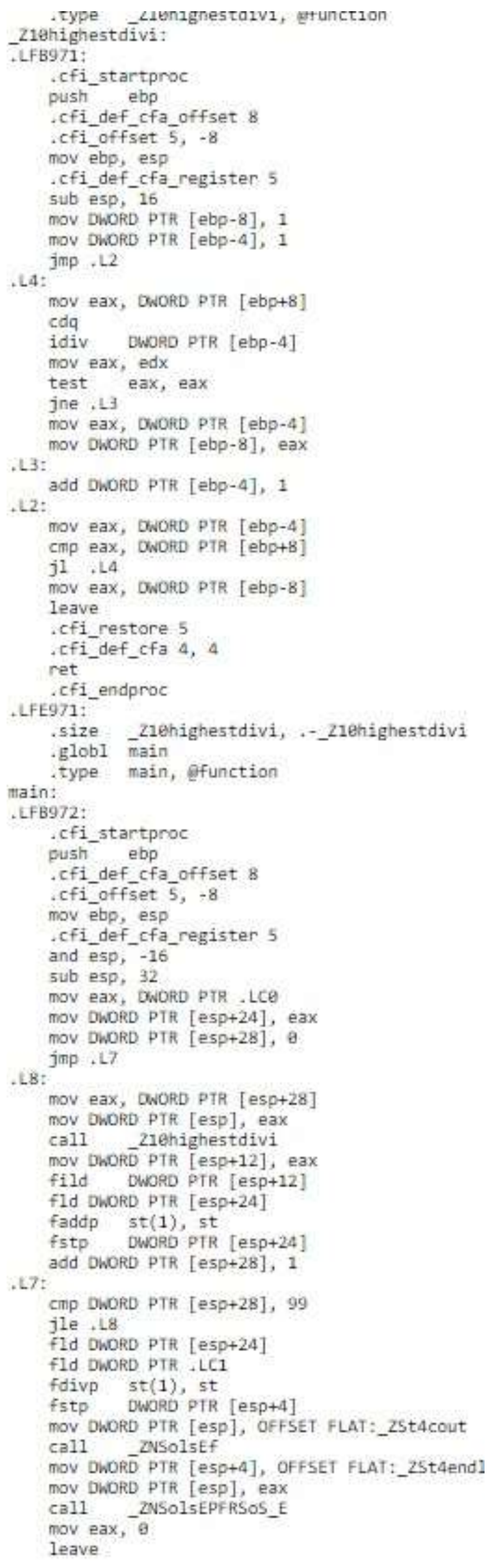
In order to test how g++ optimizes its code, I wrote a simple piece of code that would find the average highest divisor of all the numbers between one and one hundred. The main method calls the function highest divisor on each number in turn and sums them to find the average. I have included the code for the un-optimized version below on the left, and the optimized version on the right.

The first thing I notice about the optimized code is that it appears to actually be longer than its un-optimized counterpart. Each piece of code contains the same number of loops, which is surprising as I expected the compiler to make use of loop unrolling. One of the optimizations discussed in the prelab is the possibility of basing all of your parameters off of esp to avoid having to push ebp. The optimized code does exactly this. The only register pushed in the optimized version is esi to preserve its value. Ebp is never used and the parameter is located at esp+8. This also shortens the epilogue required to restore ebp. In the main section of the code, the optimized version does use the standard prologue, but it still bases the parameters off of esp.

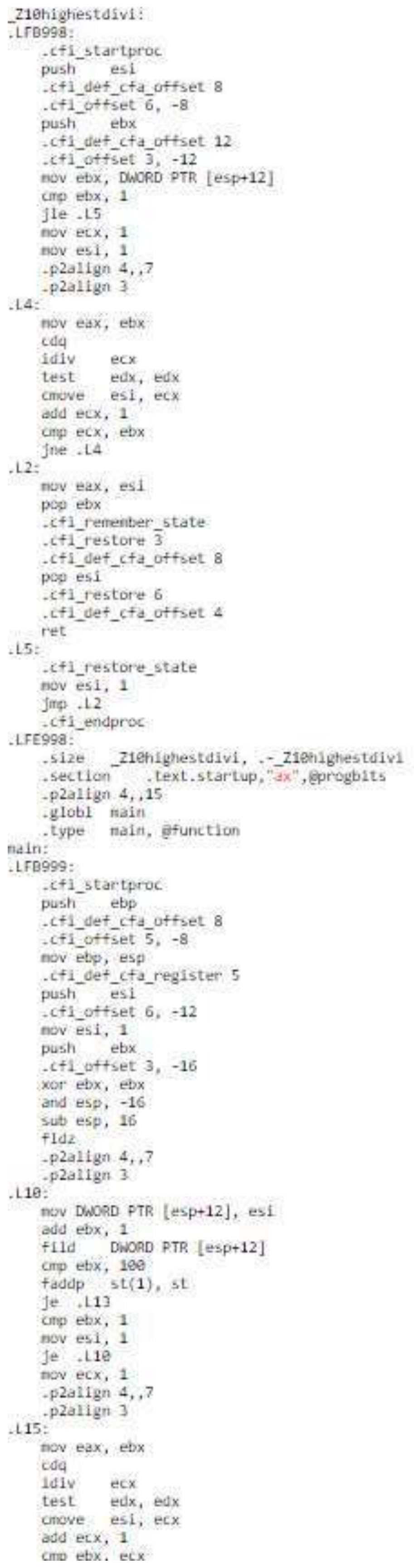
The second obvious difference is that the optimized code makes use of more registers to store values rather than accessing the address each time. The in the un-optimized highestdiv, each variable access requires DWORD PTR [ebp+-offset]. The optimized version only has to access the registers ebx, ecx and esi, allowing it to use a single DWORD command. A similar pattern can be seen in the main of each version. The un-optimized main requires numerous memory accesses relative to esp, but the optimized version only uses 2.

Possibly the most confusing difference is that the function calls for highestdiv were missing from the main section of the optimized code. In the un-optimized version, the call for highestdiv is inside the for loop as would be expected. I could not actually see how the highestdiv function is reached from the optimized main at all. Presumably, avoiding the function call removes the necessity of pushing the parameters and then jumping to another part of the code to increase speed. It is not clear how the optimized version achieves its own version of this.

The only other major difference that I could see (or at least understand) is when each version uses and esp, -16 and sub esp, #. Usually when we decrement esp, it is too make room for local variables. The un-optimized version of the code decrements esp by 32 instead of 16, allowing twice the amount of space for local variables. This is probably connected to the fact that the optimized version of the code uses more registers than local variables to limit memory access. Creating these variables is slower than modifying a register. The difference of 16 also affects the offset of later memory accesses. For example, the un-optimized version accesses values at esp+28, while the optimized version only needs esp+12. I'm sure that there are more optimizations that would be applicable if the program we generated was more complex than simple arithmetic.



```
    .type   _Z10highestdivi, @function
_Z10highestdivi:
.LFB971:
    .cfi_startproc
    push    ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    mov ebp, esp
    .cfi_def_cfa_register 5
    sub esp, 16
    mov DWORD PTR [ebp-8], 1
    mov DWORD PTR [ebp-4], 1
    jmp .L2
.L4:
    mov eax, DWORD PTR [ebp+8]
    cdq
    idiv    DWORD PTR [ebp-4]
    mov eax, edx
    test    eax, eax
    jne .L3
    mov eax, DWORD PTR [ebp-4]
    mov DWORD PTR [ebp-8], eax
.L3:
    add DWORD PTR [ebp-4], 1
.L2:
    mov eax, DWORD PTR [ebp-4]
    cmp eax, DWORD PTR [ebp+8]
    jl  .L4
    mov eax, DWORD PTR [ebp-8]
    leave
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
.LFE971:
    .size   _Z10highestdivi, .-_Z10highestdivi
    .globl  main
    .type   main, @function
main:
.LFB972:
    .cfi_startproc
    push    ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    mov ebp, esp
    .cfi_def_cfa_register 5
    and esp, -16
    sub esp, 32
    mov eax, DWORD PTR .LC0
    mov DWORD PTR [esp+24], eax
    mov DWORD PTR [esp+28], 0
    jmp .L7
.L8:
    mov eax, DWORD PTR [esp+28]
    mov DWORD PTR [esp], eax
    call    _Z10highestdivi
    mov DWORD PTR [esp+12], eax
    fild    DWORD PTR [esp+12]
    fld DWORD PTR [esp+24]
    faddp   st(1), st
    fstp    DWORD PTR [esp+24]
    add DWORD PTR [esp+28], 1
.L7:
    cmp DWORD PTR [esp+28], 99
    jle .L8
    fld DWORD PTR [esp+24]
    fld DWORD PTR .LC1
    fdivp   st(1), st
    fstp    DWORD PTR [esp+4]
    mov DWORD PTR [esp], OFFSET FLAT:_ZSt4cout
    call    _ZNSolsEf
    mov DWORD PTR [esp+4], OFFSET FLAT:_ZSt4endl
    mov DWORD PTR [esp], eax
    call    _ZNSolsEPFRSoS_E
    mov eax, 0
    leave
```



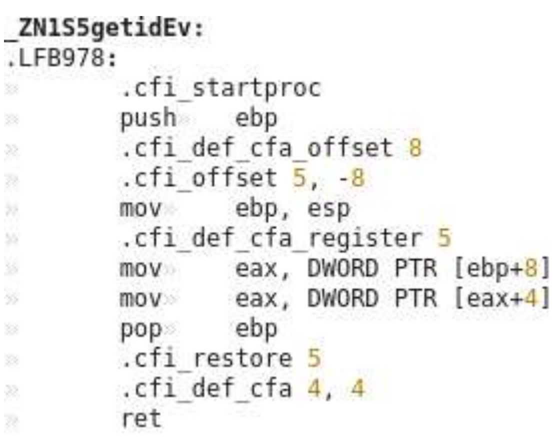
```
_Z10highestdivi:
.LFB998:
    .cfi_startproc
    push    esi
    .cfi_def_cfa_offset 8
    .cfi_offset 6, -8
    push    ebx
    .cfi_def_cfa_offset 12
    .cfi_offset 3, -12
    mov ebx, DWORD PTR [esp+12]
    cmp ebx, 1
    jle .L5
    mov ecx, 1
    mov esi, 1
    .p2align 4,,7
    .p2align 3
.L4:
    mov eax, ebx
    cdq
    idiv    ecx
    test    edx, edx
    cmove   esi, ecx
    add ecx, 1
    cmp ecx, ebx
    jne .L4
.L2:
    mov eax, esi
    pop ebx
    .cfi_remember_state
    .cfi_restore 3
    .cfi_def_cfa_offset 8
    pop esi
    .cfi_restore 6
    .cfi_def_cfa_offset 4
    ret
.L5:
    .cfi_restore_state
    mov esi, 1
    jmp .L2
    .cfi_endproc
.LFE998:
    .size   _Z10highestdivi, .-_Z10highestdivi
    .section    .text.startup,"ax",@progbits
    .p2align 4,,15
    .globl  main
    .type   main, @function
main:
.LFB999:
    .cfi_startproc
    push    ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    mov ebp, esp
    .cfi_def_cfa_register 5
    push    esi
    .cfi_offset 6, -12
    mov esi, 1
    push    ebx
    .cfi_offset 3, -16
    xor ebx, ebx
    and esp, -16
    sub esp, 16
    fldz
    .p2align 4,,7
    .p2align 3
.L10:
    mov DWORD PTR [esp+12], esi
    add ebx, 1
    fild    DWORD PTR [esp+12]
    cmp ebx, 100
    faddp   st(1), st
    je  .L13
    cmp ebx, 1
    mov esi, 1
    je  .L10
    mov ecx, 1
    .p2align 4,,7
    .p2align 3
.L15:
    mov eax, ebx
    cdq
    idiv    ecx
    test    edx, edx
    cmove   esi, ecx
    add ecx, 1
    cmp ebx, ecx
```

Dynamic Dispatch

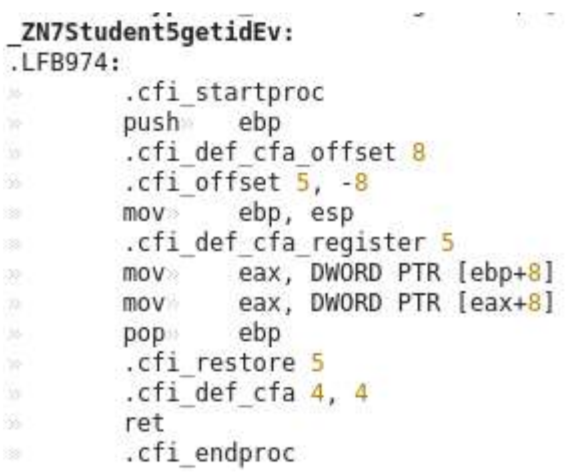
From the slides we know that when objects are created using virtual dispatch, an address to the virtual method table is kept alongside the object fields. The virtual method has to access the object at that address, lookup the method pointer in the virtual method table and finally jump to method. We should be able to see this process occurring in our x86 code. In order to test how this worked in assembly, I created two classes, person and student, that each had an id associated with them. Each class contained virtual methods to set and retrieve this id. For the sake of comparison I created two identical classes, but without the virtual methods.

Callee – It seems that the callee function is structured exactly the same way regardless of the type of dispatching. I compared the getId() methods as shown below and found them to be identical for objects of the same type. This is what we should expect, given how dynamic dispatch works. Once the function call is reached, the compiler has already completed all of the pointer dereferences that it needs to and jumped to the correct function. It only needs to move the correct value into the return register and return it.

Dynamic dispatch on the left, static on the left:



```
_ZN1S5getidEv:
.LFB978:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        mov     eax, DWORD PTR [ebp+8]
        mov     eax, DWORD PTR [eax+4]
        pop     ebp
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
```

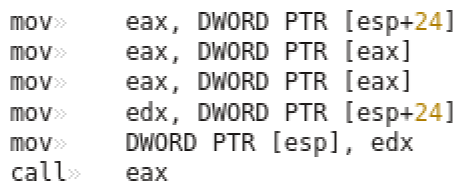


```
_ZN7Student5getidEv:
.LFB974:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        mov     eax, DWORD PTR [ebp+8]
        mov     eax, DWORD PTR [eax+8]
        pop     ebp
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

We can see that the only difference is the address used.

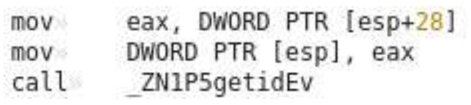
Caller – This is where the really important difference occurs. We see that for the static dispatch, each function is called directly, while in dynamic dispatch the function address has to be obtained from the virtual table. The first thing the function does is load the address of the virtual table that is stored with the object and put it in eax. We know that it should then load the function address into another register so that it can call on that function. It does that with mov eax, DWORD PTR [eax] so that the function address is now stored in eax. Calling eax is now the same as calling the correct function. Immediately after, the compiler uses call eax. The static dispatch version does not have to go through this process and can simply use call functionname, because it is clear what function is being referred to. I have included a single function call for each of the functions above.

Dynamic Student getid:



```
mov     eax, DWORD PTR [esp+24]
mov     eax, DWORD PTR [eax]
mov     eax, DWORD PTR [eax]
mov     edx, DWORD PTR [esp+24]
mov     DWORD PTR [esp], edx
call    eax
```

Static S getid:



```
mov     eax, DWORD PTR [esp+28]
mov     DWORD PTR [esp], eax
call    _ZN1P5getidEv
```

Sources: http://stackoverflow.com/questions/20147054/how-does-dynamic-dispatch-happen-in-assembly
https://en.wikipedia.org/wiki/Dynamic_dispatch