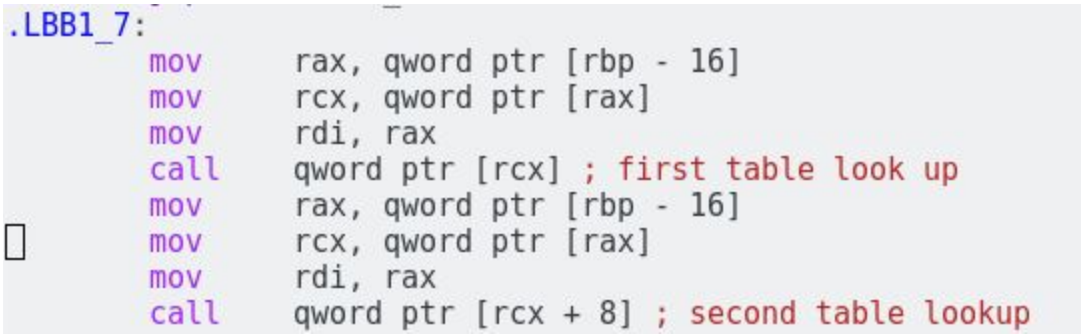
Holt Crews, jhc6we, 11/15/17, inlab09.pdf

Dynamic Dispatch

As we learned in lecture, when doing a dynamic dispatch via the virtual keyword, a virtual method table is created that is called upon at runtime to determine which function to use. These method tables have addresses to the method of the class and superclass. There are three pointer dereferences that have to take place in order for the program to locate the correct function. The virtual method has to follow the pointer to the object, then go to the virtual method table pointer, lookup that method pointer and then jump to that method. In assembly, the address of the virtual table is first moved to rax, then the address of the first function is loaded into rcx, then rax is moved to rdi, and then assembly calls the function at rcx if the first function of the table is need and increases by 8 ([rcx+8]) for every function after that. This behavior is demonstrated in the following code in which two virtual methods are called in main through dynamic



dispatch. The assembly code accesses a virtual method table that has been generated prior and then within that table accesses the method necessary.

This process allows for the C++ code to compile all of the C++ into the x86 language and then decide upon which function to use as the code is running as opposed to when the code is being compiled. Based on the situation, the correct function can then be called from the virtual method table dynamically.

-O2 flag optimization:

For this next section, a look into the -O2 flag for optimization, I focused on the code given to us in timer.cpp as an example. I compiled the timer.cpp file by itself and the following code is the result and the very first function in the cpp file which is the

```
_ZN5timerC2ERS_:                        # @_ZN5timerC2ERS_
        .cfi_startproc
# BB#0:
        push    rbp
.Ltmp3:
        .cfi_def_cfa_offset 16
.Ltmp4:
        .cfi_offset rbp, -16
        mov     rbp, rsp
.Ltmp5:
        .cfi_def_cfa_register rbp
        mov     qword ptr [rbp - 8], rdi
        mov     qword ptr [rbp - 16], rsi
        mov     rsi, qword ptr [rbp - 8]
        mov     rdi, qword ptr [rbp - 16]
        mov     al, byte ptr [rdi + 32]
        and     al, 1
        mov     byte ptr [rsi + 32], al
        mov     rdi, qword ptr [rbp - 16]
        mov     rcx, qword ptr [rdi]
        mov     qword ptr [rsi], rcx
        mov     rcx, qword ptr [rdi + 8]
        mov     qword ptr [rsi + 8], rcx
        mov     rcx, qword ptr [rbp - 16]
        mov     rdi, qword ptr [rcx + 16]
        mov     qword ptr [rsi + 16], rdi
        mov     rcx, qword ptr [rcx + 24]
        mov     qword ptr [rsi + 24], rcx
        pop     rbp
        ret
```

```
_ZN5timerC2ERS_:                        # @_ZN5timerC2ERS_
        .cfi_startproc
# BB#0:
        mov     al, byte ptr [rsi + 32]
        mov     byte ptr [rdi + 32], al
        movups  xmm0, xmmword ptr [rsi]
        movups  xmmword ptr [rdi], xmm0
        movups  xmm0, xmmword ptr [rsi + 16]
        movups  xmmword ptr [rdi + 16], xmm0
        ret
```

timer constructor. In the screenshot on the left, the unoptimized code, we can see that there are many mov instructions that are allocating memory on the stack for the various variables within the class. The optimized code minimizes these mov commands and uses the more nuanced movups command which moves the larger, 128-bit xmm registers around on the stack. Storing data in these large registers reduces the number of mov commands necessary for the constructor.

```
# BB#0:
        push    rbp
.Ltmp9:
        .cfi_def_cfa_offset 16
.Ltmp10:
        .cfi_offset rbp, -16
        mov     rbp, rsp
.Ltmp11:
        .cfi_def_cfa_register rbp
        sub     rsp, 32
        mov     qword ptr [rbp - 16], rdi
        mov     rdi, qword ptr [rbp - 16]
        test    byte ptr [rdi + 32], 1
        mov     qword ptr [rbp - 24], rdi # 8-byte Spill
        je      .LBB3_2
# BB#1:
        xor     eax, eax
        mov     esi, eax
        mov     rcx, qword ptr [rbp - 24] # 8-byte Reload
        mov     byte ptr [rcx + 32], 0
        add     rcx, 16
        mov     rdi, rcx
        call    gettimeofday
        mov     dword ptr [rbp - 4], 0
        mov     dword ptr [rbp - 28], eax # 4-byte Spill
        jmp     .LBB3_3
.LBB3_2:
        mov     dword ptr [rbp - 4], 1
.LBB3_3:
        mov     eax, dword ptr [rbp - 4]
        add     rsp, 32
        pop     rbp
```

```
_ZN5timer4stopEv:                       # @_ZN5timer4stopEv
        .cfi_startproc
# BB#0:
        push    rbx
.Ltmp2:
        .cfi_def_cfa_offset 16
.Ltmp3:
        .cfi_offset rbx, -16
        mov     ebx, 1
        cmp     byte ptr [rdi + 32], 0
        je      .LBB2_2
# BB#1:
        mov     byte ptr [rdi + 32], 0
        add     rdi, 16
        xor     ebx, ebx
        xor     esi, esi
        call    gettimeofday
.LBB2_2:
        mov     eax, ebx
        pop     rbx
        ret
```

The assembly code above is a representation of how a function (timer.stop()) is created in two different ways based on the optimization flag -O2. The left screenshot is the unoptimized code in which there is much more setup and commands used for execution. The optimized code only does the bare necessities of making the comparison for the if statement, setting a variable on the stack (running) to 0, add the required number of memory spaces to the address already at rdi, and then call “gettimeofday” with the address in the rdi register as the parameter. Here the second parameter is a NULL value and is acquired by directly xor-ing the esi parameter. The unoptimized code takes the extra time to xor eax and then mov eax into esi (a NULL value). The unoptimized code is much more regimented with values being moved to registers and then being operated on immediately following, the optimized code does this in one step. The unoptimized code also updates different global variables and takes the time to reset everything as it executes a function.

All in all, the -O2 assembly chooses certain commands that are more specialized for certain situations. The -O2 assembly also is much less regimented as it compiles, using registers on the fly and using the stack only when it has to. -O2 jumps straight into the constructor and set up of a class instead of going through a bunch bureaucratic assembly set up. -O2 does what is necessary as quickly and as in few lines as possible.

Sources:

Dynamic Dispatch:
http://loci-lang.org/DynamicDispatch.html
-O2 optimization:
http://x86.renejeschke.de/html/file_module_x86_id_208.html
https://gcc.gnu.org/onlinedocs/gcc-4.3.0/gcc/Optimize-Options.html#Optimize-Options