The Intel 80386, part 15: Common compiler-generated code sequences

devblogs.microsoft.com/oldnewthing/20190207-00

February 8, 2019

Raymond Chen

The Microsoft compiler employs a few patterns in its code generation that you may want to become familiar with.

As we saw earlier, a call to a `__thiscall` function passes the `this` pointer in the *ecx* register, with the remaining parameters passed on the stack. A typical calling sequence would go something like this:

```
;   p->Foo(x, 42);

    push    42                          ; parameter 2
    push    dword ptr [ebp-24h]         ; parameter 1
    mov     ecx, dword ptr [ebp-20h]    ; "this" for call
    call    CThing::Foo                 ; call the function directly
```

If the method is virtual, then there is a vtable lookup:

```
;   p->Foo(x, 42);

    push    42                          ; parameter 2
    push    dword ptr [ebp-24h]         ; parameter 1
    mov     ecx, dword ptr [ebp-20h]    ; "this" for call
    mov     eax, dword ptr [ecx]        ; fetch vtable
    call    dword ptr [eax+10h]         ; call through the vtable
```

If the method uses `__stdcall` instead of `__thiscall` (typically because it is a COM method), then the *this* parameter is passed on the stack rather than in the *ecx* register.

```
;   p->Foo(x, 42);

; non-virtual call
    push    42                          ; parameter 2
    push    dword ptr [ebp-24h]         ; parameter 1
    push    dword ptr [ebp-20h]         ; "this" for call
    call    CThing::Foo                 ; call the function directly

; virtual call
    push    42                          ; parameter 2
    push    dword ptr [ebp-24h]         ; parameter 1
    mov     ecx, dword ptr [ebp-20h]    ; "this" for call
    push    ecx                         ; pass as stack parameter
    mov     eax, dword ptr [ecx]        ; fetch vtable
    call    dword ptr [eax+10h]         ; call through the vtable
```

The Microsoft compiler uses a jump table for dense `switch` statements, but it adds a level of indirection so that multiple cases that leads to the same target share the same jump entry.

Consider the following fragment:

```
    switch (value)
    {
    case 2:
    case 3:
    case 5:
    case 7:
        printf("prime");
        break;

   case 4:
   case 9:
        printf("perfect square");
        break;

   default:
        printf("I'm sure you're special");
        break;
    }
```

The resulting code may look like this:

```
    mov     eax, dword ptr [ebp-30h]    ; load value
    sub     eax, 2                      ; table starts with "case 2"
    cmp     eax, 8                      ; table has 8 entries
    jae     case_default                ; not in table, go to default case
    movzx   eax, byte ptr [eax+level1]  ; get the index into the second table
    jmp     dword ptr [eax+level2]      ; jump to handler

    ...

level2 dd  offset case_prime            ; slot 0 is for cases 2, 3, 5, and 7
       dd  offset case_square           ; slot 1 is for cases 4 and 9
       dd  offset case_default          ; slot 2 is for everybody else
level1 db  0, 0, 1, 0, 2, 0, 2, 1       ; generate the slots
;          2  3  4  5  6  7  8  9       ; corresponding cases
```

Adding a level of indirection allows the level-2 jump table to be smaller. The trade-off is that you have to pay for a level-1 jump table, but if there are a lot of duplicates (such as those created by all the missing cases that go to `default:`), the trade-off may be worth it.

If you stare at the output and do some reverse-compiling, you can imagine that the compiler internally rewrote it as

```
 enum SwitchResult { Prime, PerfectSquare, Default };
 static const unsigned char level1[] =
   { Prime, Prime, PerfectSquare, Prime,
     Default, Prime, Default, PerfectSquare };
 unsigned int index1 = (unsigned)value - 2;
 switch (index1 < 8 ? level1[index1] : Default)
 {
 case Prime:
     printf("prime");
     break;

case PerfectSquare:
     printf("perfect square");
     break;

case Default:
     printf("I'm sure you're special");
     break;

 default:
     __assume(0); // not reached
 }
```

Next time, we'll wrap up our quick tour of the 80386 by walking through a simple function.

Raymond Chen

Follow