Wright Kim wdk7bj

postlab9 Report

1. Dynamic Dispatch

When you want to invoke subroutine, the compiler knows which subroutine to invoke at compile time, which is called static dispatch. However, in some cases compiler may not know which subroutine to invoke at compile time and can determine only at run time. This is called dynamic dispatch, and can use dynamic dispatch when there is an overwritten function inside the child class, meaning when both parent and child class have a function with same name. For static dispatch, if you create an object of the parent class and call the member function, say foo(), it will always call the parents subroutine, although the pointer is pointing to the child object. Therefore, the pointee is not important for the static dispatch since it will always call the function of the pointer type. In order to avoid this, we use dynamic dispatch. In order to use dynamic dispatch, simply add 'virtual' keyword in front of the subroutines. Then, at run time, the compiler will know which function to call, which will be the function of an object that pointer is pointing to. The screenshot on right side is my c++ code from class slide that I modified for the lab.

```
class A {
public:
    A(){}
    ~A(){}
    virtual void foo(){
        cout << "A" << endl;
    }
    virtual void add(){

    }
};
class B : public A {
public:
    B(){}
    ~B(){}
    virtual void foo() {
        cout << "B" << endl;
    }
    virtual void add(){

    }
};

int main() {
    int which = rand() % 2;
    A *bar;
    if (which) {
        bar = new A();
    } else {
        bar = new B();
    }
    bar->foo();
    bar->add();
    return 0;
}
```

As you can see from the assembly code on the screenshot on next page, constructor is called and it first allocates memory by calling _Znwj and create an 'A' object then set the VMT (mov [esp+24], ebx). Then, it allocates memory again and create a 'B' object and updates VMT.

```
main:
.LFB991:
    .cfi_startproc
    push    ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    mov ebp, esp
    .cfi_def_cfa_register 5
    push    ebx
    and esp, -16
    sub esp, 32
    .cfi_offset 3, -12
    call    rand
    cdq
    shr edx, 31
    add eax, edx
    and eax, 1
    sub eax, edx
    mov DWORD PTR [esp+28], eax
    cmp DWORD PTR [esp+28], 0
    je  .L8
    mov DWORD PTR [esp], 4
    call    _Znwj
    mov ebx, eax
    mov DWORD PTR [esp], ebx
    call    _ZN1AC1Ev
    mov DWORD PTR [esp+24], ebx
    jmp .L9
.L8:
    mov DWORD PTR [esp], 4
    call    _Znwj
    mov ebx, eax
    mov DWORD PTR [esp], ebx
    call    _ZN1BC1Ev
    mov DWORD PTR [esp+24], ebx
.L9:
    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
    mov eax, DWORD PTR [esp+24]
    mov eax, DWORD PTR [eax]
    add eax, 4
    mov eax, DWORD PTR [eax]
    mov edx, DWORD PTR [esp+24]
    mov DWORD PTR [esp], edx
    call    eax
    mov eax, 0
    mov ebx, DWORD PTR [ebp-4]
    leave
```

This is because the compiler does not know which object's method will be used. During run time, when the compiler can determine which method will be used, it follows the path of pointer to the method, meaning dereferences three times to be use the method at that address. Therefore, mov ex, [esp+24] is this pointer pointing an object, 'B' object in this case, thus first dereference. Then, the 'B' object is pointing the VMT by mov eax, [eax] which is the second dereference. Lastly, VMT points the method of an object by mov eax, [eax], the third dereference. Since the first method of an object is located at the very first of VMT, it dereferences by mov eax, [eax]. For the second method, everything was same except that before the third dereference is add eax, 4 since second method is located right next to the first method at VMT. Before calling the method by call eax, compiler pushes the this pointer of an object to the stack because it is a method of an object not a function by mov edx, [esp+24] and mov [esp], edx.

Interesting thing was, when I first tested with a different code that I wrote, although I placed 'virtual' keyword in front of all of the methods, if compiler knows which method will be used, which was 'child' object in this case, it skipped the process of creating parent's object and directly created child's object. Thus, 'virtual' keyword itself does not guarantee dynamic dispatch, but the compiler must not know which method to use in order to implement dynamic dispatch. The c++ and assembly code screenshots are located at next page.

```
class parent {
public:
    parent(){}
    ~parent(){}
    virtual void print(){
        cout << "parent" << endl;
    }
    virtual void add() {
        cout << "parent add" << endl;
    }
};
class child : public parent {
public:
    child(){}
    ~child(){}
    virtual void print() {
        cout << "child" << endl;
    }
    virtual void add() {
        cout << "child add" << endl;
    }
};
int main() {
    parent *p;
    p = new child();
    p->print();
    p->add();
    return 0;
}
```

```
main:
.LFB987:
    .cfi_startproc
    push    ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    mov ebp, esp
    .cfi_def_cfa_register 5
    push    ebx
    and esp, -16
    sub esp, 32
    .cfi_offset 3, -12
    mov DWORD PTR [esp], 4
    call    _Znwj
    mov ebx, eax
    mov DWORD PTR [esp], ebx
    call    _ZN5childC1Ev
    mov DWORD PTR [esp+28], ebx
    mov eax, DWORD PTR [esp+28]
    mov eax, DWORD PTR [eax]
    mov eax, DWORD PTR [eax]
    mov edx, DWORD PTR [esp+28]
    mov DWORD PTR [esp], edx
    call    eax
    mov eax, DWORD PTR [esp+28]
    mov eax, DWORD PTR [eax]
    add eax, 4
    mov eax, DWORD PTR [eax]
    mov edx, DWORD PTR [esp+28]
    mov DWORD PTR [esp], edx
    call    eax
    mov eax, 0
    mov ebx, DWORD PTR [ebp-4]
    leave
    .cfi_restore 5
    .cfi_restore 3
    .cfi_def_cfa 4, 4
    ret
```

2. Optimization

a) Unused parameters:

```
int main() {
    int a = 1;
    int b = 2;
    int c = 3;
    int d = 4;

    int add = a + b;

    return 0;
}
```

```
sub esp, 28
.cfi_def_cfa_offset 32
mov DWORD PTR [esp], 0F
call    _ZNSt8ios_base4
mov DWORD PTR [esp+8],
mov DWORD PTR [esp+4],
mov DWORD PTR [esp], 0F
call    __cxa_atexit
add esp, 28
.cfi_def_cfa_offset 4
ret
```

```
push    ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
mov ebp, esp
.cfi_def_cfa_register 5
sub esp, 32
mov DWORD PTR [ebp-20], 1
mov DWORD PTR [ebp-16], 2
mov DWORD PTR [ebp-12], 3
mov DWORD PTR [ebp-8], 4
mov eax, DWORD PTR [ebp-16]
mov edx, DWORD PTR [ebp-20]
add eax, edx
mov DWORD PTR [ebp-4], eax
mov eax, 0
leave
```

I have five int parameters inside the main function, a, b, c, d, and add, where a, b, c, d, is initialized with integer, and add is adding a and b. Among five parameters, I did not use two parameters, c and d. When compared the original assembly code with the one optimized with -

O2 flag, the very first thing I could notice was that the assembly code was reduced to about half when optimized. The original assembly code, which is the far right screenshot above, pushed ebp, and assigned all parameters and as local variables. The one with optimized, the middle screenshot above, did not push ebp and just used offset from esp and only used two parameters that is needed.

b) Functions.1:

The second experiment was how assembly code optimizes when using a function. I created a simple add function as shown in screen shot on far left above. For the add function itself, the optimized code was not pushing ebp, but using the offset of esp. It was also using offset of esp inside the main function, since the compiler knows how many parameters will be used. Also, instead of calling the add function inside the main function, it dereferences the esp where the return value of add function is stored.

c) Functions.2:

```
int add(int a, int b){
    return a + b;
}

int main() {
    int i = 0;
    int a = 1;
    int b = 2;
    int c = 3;

    int d = add(a, b);

    return d;
}
```

```
main:
.LFB972:
    .cfi_startproc
    push    ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    mov ebp, esp
    .cfi_def_cfa_register 5
    sub esp, 40
    mov DWORD PTR [ebp-20], 0
    mov DWORD PTR [ebp-16], 1
    mov DWORD PTR [ebp-12], 2
    mov DWORD PTR [ebp-8], 3
    mov eax, DWORD PTR [ebp-12]
    mov DWORD PTR [esp+4], eax
    mov eax, DWORD PTR [ebp-16]
    mov DWORD PTR [esp], eax
    call    _Z3addii
    mov DWORD PTR [ebp-4], eax
    mov eax, DWORD PTR [ebp-4]
    leave
```

The third one is similar to the second one, but wanted to see the difference when I return the result of the add function instead of just using ‘return 0;’ . Since most of the optimized and original code was same, did not placed the screen shot here. However, there was one noticeable difference, even when comparing with the result from the second experiment. The optimized assembly code inside the main function was basically two lines of code, as shown on the screen shot to the left. This is because that the compiler can pre-calculate the value it will return. Then the compiler only need to move the value to the eax, which is the return value.

```
mov eax, 3
ret
```

d) For loop

```
int add(int a, int b){
    return a + b;
}

int main() {
    int i = 0;
    int a = 1;
    int b = 2;
    int c = 3;
    int d = 0;

    for (i = 0; i < 3; i++){
        d += 2;
    }
cout <<d;
    return 0;
}
```

To see how for loop can be optimized by the compiler, I used the code I created as shown on the screen shot to the left. Interesting thing that I noticed during comparing the optimized and original code is that the compiler is pre-calculating the return value and dereferencing the address where it is stored. This type of optimization did not just occur on the for loop code, but it also occurred on other experiments that requires calculations. When I asked how can compiler can

pre-calculate values to one of the TAs, he said that this can be done because the compiler has a several stage. Basically it creates a tree of the program and reduce the redundant part. When I searched it from Google, I found this type of optimization is called constant folding or constant propagation. Constant folding was determining constant expressions on compile time instead of runtime, so it knows what to do and what it will be on compile time. As the name of the optimization indicates, this type of optimization can only be done on constants.

```
mov DWORD PTR [esp+28], 0
mov DWORD PTR [esp+36], 1
mov DWORD PTR [esp+40], 2
mov DWORD PTR [esp+44], 3
mov DWORD PTR [esp+32], 0
mov DWORD PTR [esp+28], 0
jmp .L4
:
add DWORD PTR [esp+32], 2
add DWORD PTR [esp+28], 1
:
cmp DWORD PTR [esp+28], 2
jle .L5
mov eax, DWORD PTR [esp+32]
mov DWORD PTR [esp+4], eax
mov DWORD PTR [esp], OFFSET FLAT:_ZSt4cout
call    _ZNSolsEi
mov eax, 0
leave
```

```
mov DWORD PTR [esp+4], 6
mov DWORD PTR [esp], OFFSET FLAT:_ZSt4cout
```

The one above is the code inside of the main where the constant folding optimization was done. It does not do all of the steps what original code is doing and avoids branching which can significantly increase the performance.

source:

TAs

CS2150 slide (Advanced C++)

stackoverflow
http://reverseengineering.stackexchange.com/questions/4402/what-is-operator-newunsigned-int

wikipedia
https://en.wikipedia.org/wiki/Constant_folding