Colin Harfst

cph5wr

Lab Section 103

17 November 2016

Post-Lab 9

Dynamic Dispatch

To understand how dynamic dispatch works at the assembly level, we must first review what we know about static and dynamic dispatch. Static dispatch describes an instance when a class with inheritance calls upon a method that is defined for both the "parent" class and the "child" class. In static dispatch, the compiler evaluates which method should be used based upon the type of the pointer calling the method. This differs from dynamic dispatch. By using the keyword "virtual", dynamic dispatch waits until runtime to evaluate which method to use. This is done by following the pointer to the object and checking whether the object is an instance of a "parent" class or the "child" class.

This is done by the Virtual Method Table. When we dynamically allocate memory for an object, using a pointer and the "new" keyword, we have exactly that: a pointer to a new object. Within this object, there is a field that is a pointer to an array called the Virtual Method Table (VMT). This array contains all the methods that were declared with the keyword "virtual" to use dynamic dispatch. Then, when the virtual method is called, at runtime, the program will go to the VMT and find the proper instance of the method to be used.

To understand how this is seen at the assembly level I wrote C++ code using, as expected, a person class that is inherited by a student class. Both classes then have a method, called blah which is listed as virtual in the person class. We then have C++ code within a main method that attempts to use the virtual method as follows.

```
Person *p  = new Student();

p->blah();
```

When compiling, the VMT will initially hold the memory address of Person::blah(), but once it sees that p points to a Student object, it updates the VMT to hold the memory address of Student::blah(). So, in this instance, the method blah as defined in the student class will run. This is done explicitly at the assembly level. This is done by directly overwriting the method as it is stored at the assembly level.

When I looked at the assembly code generated without the "virtual" keyword, a subroutine called _ZN6Person4blahEv- that is Person::blah(). When the "virtual" keyword is used, rax is called instead. At this point, rax holds a mess of a pointer that points somewhere in memory. The mov commands preceding this call to rax is where the magic happens. This makes it so that rax contains the memory address of the subroutine corresponding to Student::blah(). What I have noticed here that seems of importance is that at the assembly level, it is a simple as calling a difference subroutine, but it is quite convoluted to do with just moving around the contents of registers and memory. It is nice that we can abstract this idea with C++, but I see that it does in fact function as explained.

Optimized Code

I wanted to understand what happens when we use the -O2 flag when compiling. I wrote some several simple programs and then compiled them into assembly with and without the -O2 flag. The code I wrote included loops, recursive calls, and objects. I made some specific and some general observations which I have listed below.

1. It is immediately clear that the optimized code uses memory far less often. In a simple hard-coded for loop, then unoptimized code has the following code

   ```
   mov [rbp-4], 0
   mov [rbp-8], 5
   mov [rbp-12], 0
   mov [rbp-16], 0
   ```

   it is unclear to me exactly what is happening (other than the five which is hard coded), but the optimized code clearly eliminated such useless accesses to memory by using registers when possible.
   I read that a result of this simple optimization is that debuggers lose some of their functionality or reliability as they cannot access memory where variables are stored (because they are in fact not stored in variables).

2. Another difference I noted was the additional use of titles that I am unfamiliar with. For example, I see lines more frequently that look like the following.

   ```
   mov edi, _ZNSt8ios_base4InitD1Ev
   ```

   It is not entirely clear what this jumble of character means to the novice assembly coder. This line is not seen anywhere else within the code. I suspect that this jargon (or some other similar looking line of jargon) defines some essential characteristic of my program such as managing loops and conditionals.

3. In an interesting case where I wrote a factorial function, the optimized code was longer in terms of length. This increase in length was a result of a variety of things, some of which I am sure I don't understand. It did appear that the optimized code hard coded some of the recursive steps rather than simply making a recursive call. This may optimize the code in that it need not jump around and call subroutines, but rather can just run through the lines. In general, I think it is just important to note at this point that it is interesting that the generated code could be longer.

4. A final difference I noticed was, when code I wrote had conditional statements, those conditional statements would appear as jump commands in the unoptimized code, but there would not be corresponding jump commands in the optimized code. The idea here is that minimizing jumps will increase efficiency, even if minimally. Of course, the conditionals are still controlled in some capacity. I also want it to be clear that jump commands are still being used in general, but the unoptimized code very clearly uses je when my main method would check an equivalence. Similarly, it uses jmp, when in an else command. These are not seen the same way in the optimized code

I hope this shows my effort in trying to understand some differences. I tried to explain phenomenon I saw generally, while also describing specific examples, but I also understand that the specific examples may not have been described so precisely. I think these four observations describe completely different things that I noticed and I hope that that is sufficient.