Sravan Tumuluri
skt3rt
Post-Lab 9
11/22/13

The -O2 flag in the clang++ compiler is used, to put it simply, to optimize the runtime/execution of the code or program that is being compiled. This flag results in a change in the code at the assembly level to make it more efficient/fast. The optimizations that occur from this flag range from the basic reduction in instructions (combining steps, using a couple different instructions to shorten instruction count) to more complex optimizations such as inter procedural optimizations or locality optimizations to reduce memory access times or times between linking the program together. When comparing the assembly code of an optimized code snippet to that of a non-optimized piece, there are some clear differences.

```
.Ltmp5:
        .cfi_def_cfa_register rbp
        sub     RSP, 16
        lea     RDI, QWORD PTR [_ZStL8__ioinit]
        call    _ZNSt8ios_base4InitC1Ev
        lea     RDI, QWORD PTR [_ZNSt8ios_base4InitD1Ev]
        lea     RSI, QWORD PTR [_ZStL8__ioinit]
        lea     RAX, QWORD PTR [__dso_handle]
        mov     RDX, RAX
        call    __cxa_atexit
        mov     DWORD PTR [RBP - 4], EAX # 4-byte Spill
        add     RSP, 16
        pop     RBP
        ret
```

The above code is found in the non-optimized assembly file. All of this occurs before the definition of the the method (in this case it is the collatz conjecture problem described in the pre-lab). The optimized assembly file does not contain this set up code until later in the file and it contains fewer lines. The optimized file starts the code off with the declaration of the Collatz method.

```
.Ltmp13:
        .cfi_def_cfa_register rbp
        sub     RSP, 16
        mov     DWORD PTR [RBP - 8], EDI
        cmp     DWORD PTR [RBP - 8], 1
        jne     .LBB1_2
----------------------------------------
        .cfi_def_cfa_register rbp
        cmp     EDI, 1
        je      .LBB0_5
# BB#1:
        test    DIL, 1
        jne     .LBB0_3
```

Sravan Tumuluri
skt3rt
Post-Lab 9
11/22/13

The optimized code is on the bottom and as shown, it is clear that the optimized code does 2 comparisons in the space that it took the non-optimized code to do 1 comparison. Also, the optimized code uses different registers and assigns parameters to registers instead of accessing and loading into memory addresses directly. This type of optimization is about locality and the idea behind this is that, the compiler would like memory or values that are frequently accessed together to be close in memory so as to reduce the time it takes to travel between addresses (http://en.wikipedia.org/wiki/Optimizing_compiler).

Another interesting topic to look at is dynamic dispatch. This is the process by which the compiler/program decides which implementation of a method is chosen at the runtime of the program. This is used for when the program cannot determine which implementation to run at compile time due to command line parameters affecting which implementation or due to the run time type of said parameters. Single dispatch is when the selection of the method is only based on the type or class of the object of the program. Multiple dispatch is when there are multiple factors in deciding which implementation of the method to run.

Implementation of dynamic dispatch is often done through virtual functions in C++ and these virtual functions are functions that can be overridden through inheritance, giving an underlying basis to polymorphism in object oriented programming. When done in C++ to show dynamic dispatch, the overridden is chosen based on which specific object or class calling the method, based on the command line parameters given.