What a Compiler Does - Introduction*

Charles Severance
Kevin Dowd

This work is produced by OpenStax-CNX and licensed under the
Creative Commons Attribution License 3.0†

1 What a Compiler Does

The goal of an *optimizing compiler* is the efficient translation of a higher-level language into the fastest possible machine language that accurately represents the high-level language source. What makes a representation good is: it gives the correct answers, and it executes quickly.

Naturally, it makes no difference how fast a program runs if it doesn't produce the right answers.[1] But given an expression of a program that executes correctly, an optimizing compiler looks for ways to streamline it. As a first cut, this usually means simplifying the code, throwing out extraneous instructions, and sharing intermediate results between statements. More advanced optimizations seek to restructure the program and may actually make the code grow in size, though the number of instructions executed will (hopefully) shrink.

When it comes to finally generating machine language, the compiler has to know about the registers and rules for issuing instructions. For performance, it needs to understand the costs of those instructions and the latencies of machine resources, such as the pipelines. This is especially true for processors that can execute more than one instruction at a time. It takes a balanced instruction mix — the right proportion of floating-point, fixed point, memory and branch operations, etc. — to keep the machine busy.

Initially compilers were tools that allowed us to write in something more readable than assembly language. Today they border on artificial intelligence as they take our high-level source code and translate it into highly optimized machine language across a wide variety of single- and multiple-processor architectures. In the area of high performance computing, the compiler at times has a greater impact on the performance of our program than either the processor or memory architecture. Throughout the history of high performance computing, if we are not satisfied with the performance of our program written in a high-level language, we will gladly rewrite all or part of the program in assembly language. Thankfully, today's compilers usually make that step unnecessary.

In this chapter we cover the basic operation of optimizing compilers. In a later chapter we will cover the techniques used to analyze and compile programs for advanced architectures such as parallel or vector processing systems. We start our look at compilers examining how the relationship between programmers and their compilers has changed over time.

*Version 1.3: Aug 25, 2010 11:23 am -0500
†http://creativecommons.org/licenses/by/3.0/
[1] However, you can sometimes trade accuracy for speed.