CS3320: Compilers - I

Mini-Assignment #1 - Compilers and their Options

# Soumi Chakraborty

ES19BTECH11017

# 1 Compiler Options in GCC and LLVM

Compiler options are expressions/instructions that can be given in the command line to alter the default options of a compiler’s operation.

Compilers usually preprocess, compile, assemble and link when invoked. However, situations may arise when you need to stop the compiler before it has finished all the steps, or need only a particular set of steps, or need it to do something extra. This is where compiler options come into play.

## 1.1 GCC Compiler Options

Compiler options in GCC can usually be used for both C and C++ code, however, some of them are specific to each language.

The command line expression to run gcc with it’s default options is: gcc <filename.c>

Command line options may or may not need arguments, and they are usually passed right after the option either separated by a space or an equal to sign. Here are a few commonly used compiler options for gcc:

* **-o**: gcc -o helloWorld helloWorld.c  
  Allows us to set the name of the executable file to something of our choice.
* **-E**: gcc -E helloWorld.c > helloWorld.i

Gets only preprocessor output.

* **-S**: gcc -S helloWorld.c > helloWorld.s

Gets only assembly output.

* **-C**: gcc -C helloWorld.c

Gets only compiler output.

* **-Wall**: gcc -Wall helloWorld.c

Produces warnings thrown when a code with warnings is run.

* **-l**: gcc helloWorld.c -lm

Links libraries. The above example links math.h to the .c file.

* **-save-temps**: gcc -save-temps helloWorld.c

Generates the preprocessor, compiler and assembly output along with the executable file.

## 1.2 LLVM Compiler Options

LLVM can be used to create the frontend and the backend options for any programming language. For example, the Clang frontend was developed using LLVM which can be used to compile C/C++ code. For this short note, we will stick to command line options for Clang. A few of them have been listed below:

* **-o**, **-E**, **-S**, **-C** and a few other options work with LLVM/Clang as well.
* **-Werror**: converts warnings into errors. It can also accept arguments to turn a particular warning into an error.

# 2 Frontends supported by GCC and LLVM

The frontend part of a compiler is the thing that converts the source code to a form that can be read by the backend compiler to convert it to machine code. The frontend compiler takes the source code and is in charge of parsing, scanning and creating a symbol tree. It differs from language to language, and it doesn’t have much to do with the compiler backend.

## 1.1 GCC

GCC currently supports frontends for C, C++, Fortran, Go, D, Objective C and Ada.

These are the languages which are officially supported by GCC. However, there are some other popular languages with frontends that GCC supports. Thus, anyone can develop a language frontend that is supported by GCC without having to worry too much about the backend.

## 1.2 LLVM

LLVM is written in C++ and was originally meant to be compatible with all the GCC supported frontends. However, LLVM became very popular and supports a large number of frontends for languages like Rust, Swift, Julia, Haskell alongside the languages mentioned for GCC. LLVM is also widely used by people to develop their own frontends.

# 3 Various Architectures and Backends

Upon compiling a very simple C source code file for different architectures (on x86\_64, aarch64, ARM (32 bit)) on a x86\_64 machine, it was noticed that only the x86\_64 executable file gave the desired output, while the other two threw errors.

This makes it clear that the machine level executable code is backend dependent, even if the frontend used remains the same.

If we were to successfully run the executable files of a different architecture on this machine, we would have to cross compile.

# 4 Optimisation Levels

Compiling a few large C++ executable files (the codes had a time complexity of O(n2) and had to work with large 2D matrices) without any optimisation, and then with the levels O3, O2, O1, and O0, gave the following running times (on an average):

| **Optimisation Level** | **Running time (seconds)** |
| --- | --- |
| None | 0.195038 |
| O0 | 0.174782 |
| O1 | 0.162873 |
| O2 | 0.159612 |
| O3 | 0.128033 |

From the above table, it becomes clear that the execution time was as follows:

O3 < O2 < O1 < O0 < no optimisation.

The optimisation levels -Os and -Oz are mainly for optimising the size rather than the execution time. Moreover, these only work with LLVM, and not GCC. Upon reading up about these two optimisation levels online, it appears as though Oz has a slight edge over Os when it comes to optimising the size. However, I wasn’t able to verify if this indeed was the case because I do not have an appropriately large source code file which might show significant size changes upon optimisation.