GCC Optimization Primer

Ratnadeep Bhattacharya

August 19, 2019

Generally Used GCC flags and their uses

-Wall - all warnings
-Wextra - extra warnings
-g - debugging information
-Wpedantic - standard conformation
-Werror - turn warnings into errors
-std=c99 - asks the compiler to use the C99 standard

Basics and References

The GCC compiler is used to streamline code and speed it up. However, in the process it may render the code somewhat unreadable.

Compiler optimizations will generally make the code faster but probably more difficult to understand.

A list of GCC optimization options:
https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

A discussion on GCC optimization options
https://www.rapidtables.com/code/linux/gcc/gcc-o.html

Generally speaking most of the time execution time is traded off with compile time and vice versa.

A dated article that explains the optimizations well:
https://www.linuxjournal.com/article/7269

A brief overview of the optimization levels of GCC

A large variety of optimizations are provided by GCC at one of three levels but some at multiple levels. Some optimizations reduce the size of the resulting machine code while others reduce execution time.
Optimizations can be done by:

1. Mentioning the level of optimization
2. Specifically mentioning the options with the -f switch
3. A combination of 1 and 2

4. Eg.
   gcc -O1 -fno-defer-pop -o test test.c
   Enable O1 level optimization but disable the defer-pop option. This option basically allows the compiler to let function arguments accumulate on the stack for several function calls and pops them all at once. By disabling this option, the compiler is forced to pop function arguments from the stack as soon as the function returns.

** The default is level 0 (-O0 or -O). This reduces compile time and make debugging produce expected results.

Level 1 (-O1)

This level generally tries to reduce both size of the code and execution time. The optimizations done at this level do not increase compile time significantly.

Level 2 (-O2)

At this level, generally all optimizations, supported by that architecture, that does not involve a speed-size tradeoff are performed. With these optimizations, compilation time will be significantly increased.

Level 2.5 (-Os)

This is a special level at which all level 2 optimizations are enabled excepting the ones that increase code size – these are the alignment options (align-loops, align-jumps, align-labels, and align-functions). These optimizations skip space to align loops, jumps, labels and functions to an address that is a power of 2 in an architecture dependent manner. Skipping to these boundaries can increase speed of execution but also increases code size. Furthermore, all reorder block optimizations and prefetch-loop-arrays are also disabled.

Level 3 (-O3)

At this level, the focus is completely on speed. Several optimizations, further to the ones turned on at O2, are turned on that increase speed of execution but increases code size significantly. Although, this level produces fast code, the size of the image can be detrimental of execution time. For example, if size of code image is greater than size of instruction cache then severe performance penalties will be imposed.

-Ofast

Disregard strict standard compliance. The program will not be standard compliant anymore.

-Og

This should be the standard choice for the edit-compile-debug cycle. This level only enables optimizations that will not hamper debugging.

Note

It is also possible to optimize the program with a specific target architecture in mind with the “-march=Type” option.