

# Optimization

## <sup>1</sup>GCC/Clang Optimizations for Embedded Linux



---

<sup>1</sup> Tiwari Abhinav Ashok Kumar

## Introduction to clang

What is **clang**? Clang is the front end for the **framework LLVM**. This front end was designed to compile **C, C++, and Objective-C++** to machine code. The **LLVM** framework "**is a set of reusable and modular compiler and toolchain technologies**," which means you can use the LLVM to create a compiler for any language, even the language invented by you.

*LLVM consists of the library which consists of things such as code generation for the number of processes and optimization purposes.*

Clang is a compiler used instead of GCC to create the executable from your source file.

Clang is a completely open-source compiler for the C/C++ and other C family of programming languages

The AIM of clang is to be the best in compiling the C family program language. That is why clang is the front end for the LLVM and usage LLVM optimizer and code generation, which allows clang to provide high-quality optimization and code generation support for many targets.

, You can see **Clang Web Site** or the **LLVM Web Site** to get more information about them..

The clang is

1. **Native front end to the LLVM.**
2. **Support C/C++ and other C family Program.**
3. **It don't have a backend (clang uses LLVM as a backend.)**
4. **Developed by Apple in 2007**

## History of Clang

During the start of the 20th century(2000) LLVM project started. This was the University of Illinois at Urbana-Champaign. This project was under the direction of Vikram Adve and Chris Lattner.

LLVM was designed to be a research infrastructure whose aim is to investigate dynamic compilation techniques for static and dynamic programming languages.

It was the year 2005 when, [Apple Inc.](#) started using the LLVM in their several commercial projects. Some of these projects are the [iOS SDK](#) and [Xcode 3.1](#). One of the first uses of LLVM was an [OpenGL](#) code compiler for [OS X](#) that converts OpenGL calls into more fundamental calls for [graphics processing units](#) (GPU) that do not support certain features. This allowed Apple to support OpenGL on computers using [Intel GMA](#) chipsets, increasing performance on those machines.<sup>[1]</sup>

Initially, Apple thought to use GCC as a front end to the LLVM. But here the main problem was apple's usage of Objective-C. GCC doesn't have good compatibility with Apple's IDE(integrated development environment). For GCC developers Objective-C was on lowest priority. GCC source code is even not modular.

Finally, GCC's license agreement, the [GNU General Public License \(GPL\) version 3](#), requires developers who distribute extensions or modified versions of GCC to make their [source code](#) available, but LLVM's [permissive software license](#) is devoid of such impediment.

Due to all these and some of the other hurdles Apple thought to develop a new compiler front end Clang which supports C, C++, and Objective-C.

The project received permission in 2007 to become an **open-source** project.

## Introduction to GCC

**What is GCC?**(GCC GCC abbreviation is GNU C compiler). This **GNU C compiler was developed by Richard Stallman**. Richard Stallman is the founder of the GNU project. He founded the GNU project in 1984.

The main objective of this project is to create a complete **Unix-like operating system whose aim was to provide freedom and cooperation between computer users and programmers**. To create a complete Unix-like operating system as free software, to promote freedom and cooperation among computer users and programmers.

GCC as time passes is growing and started supporting many languages such as C/C++, ObjectiveC/C+, Java(GCJ), Fortran, Ada, Go, OpenMP, etc. Initially, it was known as Gnu C Compiler but now it is known as "**GNU Compiler Collection**"

You can get more information about the GGC at <http://gcc.gnu.org/>

GCC is one of the important parts of the "**GNU Toolchain**".

The GNU Toolchain consist —

1. **GNU Compiler Collection (GCC):** compiler which supports many languages.
2. GNU Make: Automated build System.
3. GNU Binutils: set of tools such as a linker, assembler, debugger, disassembler etc.
4. GNU Bison: parser generator.

GCC is

1. **The GNU compiler Collection**
2. **Native and cross-compilation**
3. **Multiple language front end**
4. **Developed by Richard Stallman.(1984).**

## **History of GCC**

The founder of the GCC project is **Richard Stallman**. We had a discussion earlier. This project was developed to create a Unix-like **operating system** as free software. Whose AIM was to promote freedom and cooperation among computer users and programmers.

All the **OS** which are based on the **UNIX required a C compiler**, and during that time there was no free compiler of C was available. This project **GNU Project** was being developed from scratch and it was funded by the Ngo's donation and individuals and companies to the Free Software Foundation. Which setup to support the **GNU**.

In **1987** first release of the GNU came into the picture. It was the first toolchain with a portable **ANSI C** optimizing compiler and which was released to the open source. Since after that time GCC has become one of the most important tools in the development of free software.

Later in **1992** GNU came up with the **2.0** series with additional features to compile **C++**. In **1997** additional experimentation started with the name **EGCS** to improve optimization and **C++** support.

After this **EGCS** became the new main branch for **GCC** development.

As there was no competitor to GCC over time GCC has extended to support many additional languages.

Nowadays **GCC is referred to as GNU Compiler Collection** and its development is guided by the **GCC Steering Committee**.

# Optimization Flag

*Here we are going to discuss about the flags which control the optimization and why optimization is important.*

**1. The main goal of the compiler is to convert a source file into assembly language (object code) with correct functionality and semantics to create an executable which will get executed on the target machine.**

**2. One of the main roles of the compiler is also to reduce the cost of compilation, decrease the size of the machine code, improve performance and create an object code and executable with debug info which will be beneficial for debugging.**

In order to achieve the goal mentioned in point 2, the compiler turns on the optimization. When the compiler turns on the optimization flags then the compiler is able to boost the performance, reduce the size of the executable and increase the chances of debugging the program.

***How does the compiler perform optimization?***

The compiler starts its optimization by first taking knowledge of the program. When we compile multiple files at once we get output in a single file. These output in a single file permits the compiler to use information gained from all of the files when compiling each of them individually.



Some of the optimization techniques are:-

1. **Local optimizations**
2. **Global optimizations**
3. **Loop optimizations**
4. **Prescient store optimizations**
5. **Interprocedural, whole-program, or link-time optimization**
6. **Machine code optimization and object code optimizer**
7. **Programming language-independent vs. language-dependent**
8. **Machine-independent vs. machine-dependent and many more.**

**Optimization can be further classified into machine-dependent and machine independent which is not the goal of this documentation. The documentation goal is to discuss the various optimization flags compiler usage.**

Every optimization in the compiler runs as a pass and most of these are controlled by the flags. But not all optimizations are directly controlled by the flags. Some of the optimizations depend on other optimization.

This document only covers the optimization that has a flag.

When we use **-O0** at this point most of the optimizations are disabled similarly when we use debug flags such as **-g** or **-Og** most of the optimization gets disabled.

One of the factors which affect the optimization is the way the compiler is configured for the particular architecture. Depending upon the way the compiler is configured for the particular architecture we can see a little bit of difference in each level of the **-O<number>** (here the number is replaced with 0,1,2,3).

## Optimization flag in GCC

In order to see various optimization enabled at each level of optimization in GCC use flag **-help=optimizers**.

```
-> % gcc --help=optimizers
```

The following options control optimizations:

- O<number> Set optimization level to <number>.
- Ofast Optimize for speed disregarding exact standards compliance.
- Og Optimize for debugging experience rather than speed or size.
- Os Optimize for space rather than speed.
- faggressive-loop-optimizations Aggressively optimize loops using language constraints.
- falign-functions Align the start of functions.
- falign-functions= This option lacks documentation.
- falign-jumps Align labels which are only reached by jumping.
- falign-jumps= This option lacks documentation.
- falign-labels Align all labels.

.....  
.....  
**Run above command in linux to see its complete output.**

When we use option **-Q** with **-help** it gives a description whether that particular flag is [**disabled or enabled**].

```
-> % gcc -Q --help=optimizers
```

The following options control optimizations:

- O<number>
- Ofast
- Og
- Os
- faggressive-loop-optimizations [enabled]
- falign-functions [disabled]
- falign-functions=
- falign-jumps [disabled]
- falign-jumps=
- falign-labels [disabled]

.....  
.....  
**Run above command in linux to see its complete output.**



## Optimization Flags Used

### -O<number>

#### -O0

This flag in gcc enable following optimization.

```
-> % gcc -Q -O0 --help=optimizer | grep enabled &>gcc_oo.enabled
-> % cat gcc_oo.enabled
faggressive-loop-optimizations      [enabled]
-fallocation-dce                    [enabled]
-fasynchronous-unwind-tables       [enabled]
-fauto-inc-dec                      [enabled]
-fbit-tests                         [enabled]
-fdce                               [enabled]
-fdelete-null-pointer-checks        [enabled]
-fearly-inlining                    [enabled]
-ffp-int-builtin-inexact            [enabled]
-ffunction-cse                      [enabled]
```

Run above command in linux to see its complete output.  
Or ([Click here](#))

This flag reduce the size and increase performance without running any optimization. By default if you enable **debugging** it goes for **-O0**. This flag also allow debugger to provide expected result.

#### -O1

When we optimize the compiler, compiler take a lot of memory and time for a large function. With **-O1** flags compiler put an extra effort to reduce the code size and execution and this optimization mostly focus on the size optimization.

```
-> % gcc -Q -O1 --help=optimizer | grep enabled &>gcc_o1.enable
```

```
-> % cat gcc_o1.enable
```

```
faggressive-loop-optimizations [enabled]
-fallocation-dce [enabled]
-fasynchronous-unwind-tables [enabled]
-fauto-inc-dec [enabled]
-fbit-tests [enabled]
-fdce [enabled]
-fdelete-null-pointer-checks [enabled]
-fearly-inlining [enabled]
-ffp-int-builtin-inexact [enabled]
-ffunction-cse [enabled]
```

**Run above command in linux to see its complete output.**

**Or ([Click here](#))**

## **-O 2**

Every optimization optimizes more than it previously. This optimization nearly all supported optimization. This optimization as compared to -O1 increases both compilation time and the performance of the generated code. This optimization provides a balance optimization between space and time. Let's take an example of, **loop unrolling(increases size) and function inlining( increase size)**, but these two optimizations decrease the execution time. What to do here?. This is confusing what to do and these types of conflicting optimization are not performed in this phase.

By using above mentioned command you can see all the optimization flag involve in these optimization stages


**([Click here to have a view](#))**

## **-O 3**

Third and the highest level of optimization which enables more optimization than O2. It is more important than size.

**O3 = O2 + reaming\_regisiter + optimization which\_gives\_importance\_to\_speed\_over\_size**

Here inlining is also enabled which can increase performance but increase the size of the executable also. That's the reason O3 can produce fast code but due to the increasing size of

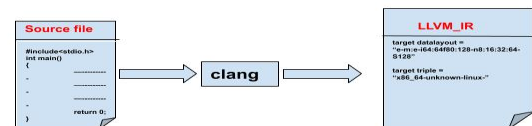


the code, we can see a performance penalty. Out of all three levels, O2 increases the chance of producing better code by balancing the size and speed.

## Optimization flag in Clang

Clang is the front end of the LLVM as we have discussed. Then how optimization flags are taken care of by the clang. For these, you need to have knowledge about the compiler's front-end and back-end.

Clang's work is to just convert the source code to the LLVM\_IR  
LLVM\_IR is nothing but a textual lower-level Representation of what our source code should do.



[LLVM](#) is a set of libraries whose primary task is the following: suppose we have the LLVM IR representation of the following C++ function

```
int double_this_number(int num)
{ int result = 0; result = num; result = result * 2; return result; }
```

the core of the LLVM passes should **optimize** LLVM IR code:

Unoptimized		Optimized
LLVM_IR	-----> LLVM_PASSES ----->	LLVM_IR

LLVM\_IR is an intermediate between clang and LLVM framework. In some of the research papers, they have even mentioned that LLVM libraries are middle end.

**As LLVM is not just a library it is a set of data structure , utility , modules ,etc). Clang also take benefit of LLVM libraries during the front end processing**

For these reason clang is called as a driver

Opt tool is there in llvm where various transformation, analysis, and modular passes are run.

In llvm everything in the compiler runs as a Pass. To manage this Past there are two pass managers in —

1. Legacy Pass Manager
2. New Pass Manager (After LLVM 13.0.0.0 the legacy pass manager have been deprecated and the new pass manager is used by default)

This means that the previous solution for printing the optimization passes used for the different optimization levels in opt will only work if the legacy pass manager is explicitly enabled with **-enable-new-pm=0**. So as long as the legacy pass manager is around (expected until LLVM 14), one can use the following command.

This below-mentioned command so the different level of the optimization is

```
llvm-as < /dev/null | opt -optimization-flag -disable-output -debug-pass=Arguments -enable-new-pm=0
```

There is a user guide on how to use the new pass manager with opt on the [LLVM Website](#).

## Optimization Flags Used

### -O0 —

```
-> % llvm-as-14 < /dev/null | opt-14 -O0 -disable-output -debug-pass=Arguments -enable-new-pm=0 &>
clang_O0
-> %cat clang_O0
Pass Arguments: -tti -verify
Pass Arguments: -targetlibinfo -tti -assumption-cache-tracker -profile-summary-info
-annotation2metadata -forceattrs -basiccg -always-inline -barrier -annotation-remarks -verify
```

Run above command in linux to see its complete output.  
Or ([Click here](#))

### -O1 —

```
-> % llvm-as-14 < /dev/null | opt-14 -O0 -disable-output -debug-pass=Arguments -enable-new-pm=0 &>
clang_O0
-> %cat clang_O1
```

Run above command in linux to see its complete output.  
Or ([Click here](#))

### -O2 —

```
-> % llvm-as-14 < /dev/null | opt-14 -O2 -disable-output -debug-pass=Arguments -enable-new-pm=0 &>
clang_O2
-> %cat clang_O2
```

Run above command in linux to see its complete output.  
Or ([Click here](#))

## **-O3**

```
-> % llvm-as-14 < /dev/null | opt-14 -O3 -disable-output -debug-pass=Arguments -enable-new-pm=0 &> clang_O3  
-> %cat clang_O3
```

Run above command in linux to see its complete output.  
Or ([Click here](#))

There is a user guide on how to use the new pass manager with opt on the [LLVM Website](#).

Below mentioned are some of the optimization flags which are common in both **gcc and clang**.

Flag	When To Use
O0	When you need binaries without optimization useful while debugging
O1	Favour size over speed of code similar to (Og/Os )
O2	To maximize the speed and size. (default for most of the compiler)
O3	Turn on all the optimization which O2 do, but favour speed over size
Os	Similar to O2 but does not enable opt passes which increases size
Ofast	O3 plus some inaccurate math and enable all the passes that is not enable for all standard compilant.
Og	Optimization for the debugging info.



## Conclusion

We are aware of the things that all applications are different. This leads to a clear concept that there is no ideal configuration of optimization and option switches that yield the best result. so there's no magic configuration of optimization and option switches that yield the best result.

The easiest way to achieve good performance is to rely completely on the **-O2 optimization level**. If you are not interested in assumption then specify target architecture using the **-march=**. For **space-constrained applications**, the **-Os** optimization level should be considered first.

By **enabling and/or disabling** we can also achieve the best performance.

---