Recently, I found I was not very clear about the difference between GNU GCC and Clang. It affected implementation and learning somethings, so I spent free time in some days to research it carefully.

During research process, I found many problems actually from language (not the programming language) and loose conceptual understanding.

If you search `clang` online, some people will tell you that it is a frontend, and then copy some introductions about compiler from books, list a bunch of tables for comparison, without any explanation of the principles and mechanisms. So more questions will pop up:

- Why `clang` is a frontend? Isn't it a complete compiler? If `clang` is a complete compiler, why is it called a front-end? If it's not complete, what is the backend?
- What exactly is the definition of a compiler? I feel that the definition of compiler in the book is different from the actual `gcc`.

I need to explain something: `gcc` here refers to the command you can use directly in Linux distributions such as Ubuntu (from the GNU software group). If I mean GNU GCC project, it will be written as "GNU GCC". If I mean `llvm-gcc`, it will not be abbreviated to `gcc`.

This article will gradually answer this series of questions. In the process, it will not only let you understand what `clang` is, but also let you know more about the compilation process, compiler, `gcc` and LLVM.

A modern program called "compiler" (like `gcc`) is a collection of tools

First you need understand one thing which is the answer to most of questions above: **A modern program called "compiler" (like `gcc`) is a collection of tools, including a preprocessor, a compiler, and calls multiple tools such as assembler, linker or loader, rather than a single compiler** (this conflict of terms and nouns is also one of the important reasons for misunderstanding).

Giving the answer first so that readers can understand the explanation better with the answer.

What exactly is compiler (or what is the compilation process)

As mentioned above, "compiler" is a term that often has conflicting expressions: **In many speeches, blogs, textbooks, and professional books, a compiler is described as "a program that converts source code into executable program" (For example, the "compiler" `gcc` can directly source code into an executable program)**. This sentence succinctly and accurately describes what happens when using the `gcc`, but it is not the definition of compiler.

Let us see the introduction of compiler in "Compiler: Principles, Techniques, and Tools Second Edition" (AKA "Dragon Book"), it also is the classic definition of compiler:

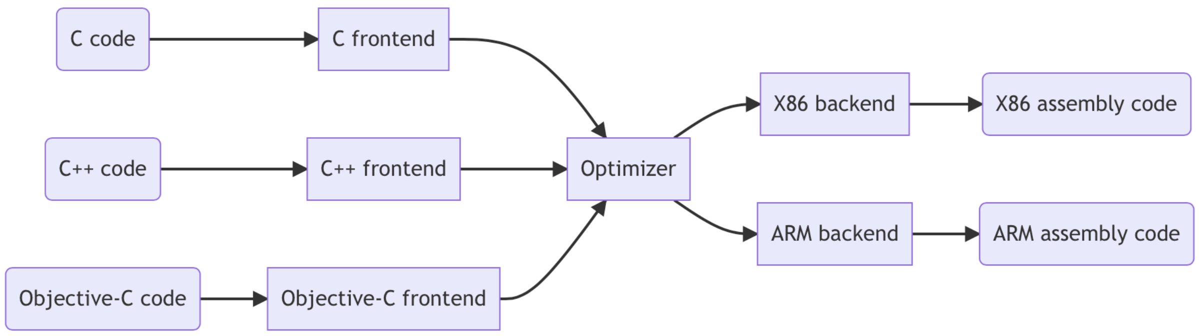
> Simply stated, a compiler is a program that can read a program in one language - the source language - and translate it into an equivalent program in another language - the target language.
> If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.

It actually means: the source code we wrote will be converted by compiler to get the code in another language, and if the converted code is in machine language, then the target code is an executable program. But if there are multiple source code files or libraries, it may be a Shared Object.

In other words, a compiler is actually a program that converts one language into another.

However, according to the standard compilation process in recent decades, the compiler refers to the program that converts source code files such as `.c` into `.s` assembly files. For the convenience of explanation, "compiler" in the following is based on this definition unless special stated,.

Under this definition, the internal workflow of the compiler is roughly as follows:

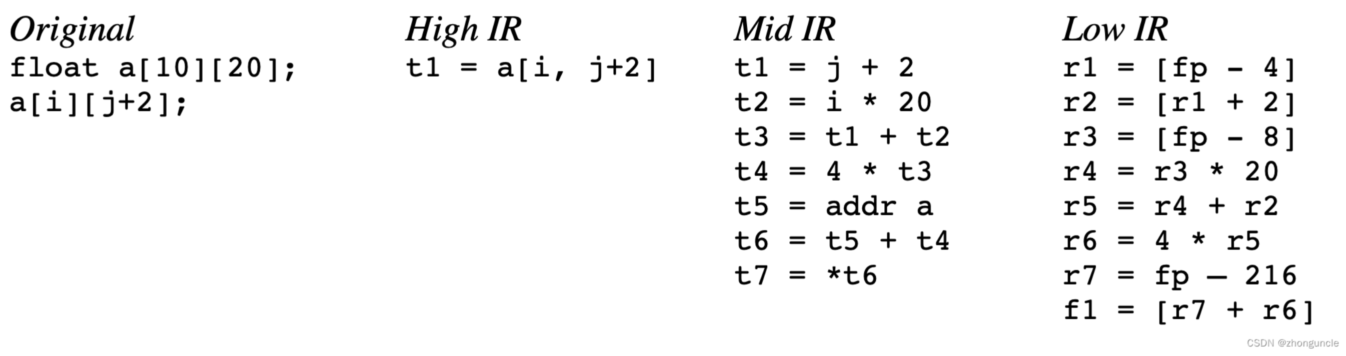


The compiler can generate platform-specific assembly code. Then the assembler converts the assembly code into machine language, and finally the linker links it into an executable program.

There are some additions:

1. The source code here have been preprocessed;
2. A language front-end generally refers to the lexical parser (Lexer) and the syntax analysis program (Parser). The front-end will convert the source code step by step (from high-level to low-level) into the intermediate expression (IR) required by the optimizer. front-end is implemented by multiple analyzers.
3. Generally, there is a "AST (Abstract Syntax Trees)" listed before the optimizer. This is a high-level intermediate expression, which is basically a reorganization of the source code.
4. Sometimes, optimizer is called the middle end. The optimizer not only improves performance, but as a middle end, it can better separate the front and back ends, increasing the possibility of cross-compilation.

The process from source code to high-level intermediate expression, and then to low-level intermediate expression as follows:

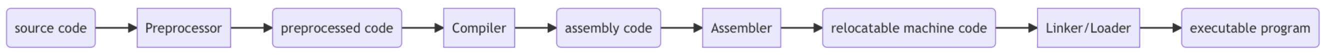


```
Original             High IR          Mid IR          Low IR
float a[10][20];     t1 = a[i, j+2]   t1 = j + 2      r1 = [fp - 4]
a[i][j+2];                            t2 = i * 20     r2 = [r1 + 2]
                                      t3 = t1 + t2    r3 = [fp - 8]
                                      t4 = 4 * t3     r4 = r3 * 20
                                      t5 = addr a     r5 = r4 + r2
                                      t6 = t5 + t4    r6 = 4 * r5
                                      t7 = *t6        r7 = fp – 216
                                                      f1 = [r7 + r6]
```

Here is an article for details: 《Intermediate Representation》

The process of converting source code into an executable program

The complete process of converting source code into an executable program is what we usually call the "compilation process." Over the past few decades, this process like (rounded rectangle represents code, rectangle represents various processors):



You can seen the source code go through a preprocessor, a compiler, an assembler, a linker or a loader to become executable program. The compiler is only responsible for converting the source code into corresponding assembly code.

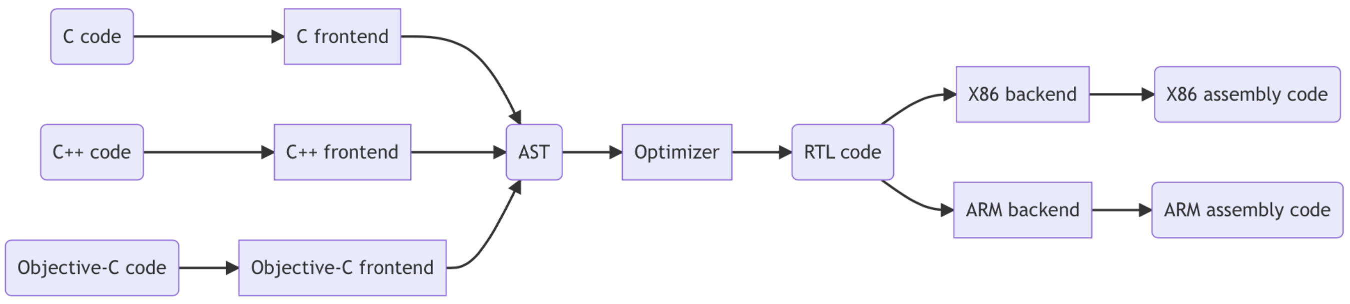
Process of gcc

You may often hear about compiler and assembler, the other three tools above are probably rarely heard of. The following uses C language and `gcc` to introduce this process. `gcc` contains the preprocessor `cpp`, and also calls the assembler `as` and the linker `ld`.

For an introduction to the three tools and the detailed process of each step in the compilation process, you can read my other article "Use gcc to show the complete compilation process", this article also introduces some `gcc` operation methods through practical operations. **It is highly recommended to read this article now, otherwise you may only understand the surface of content. The content of the article was originally intended to be placed here, but the number of word will over 10,000 words, which would take too long to read**.

gcc internal workflow

The internal workflow of gcc is as follows, the preprocessing process is ignored here:



Clang internal workflow

With the times, the traditional compilation process is no longer enough:

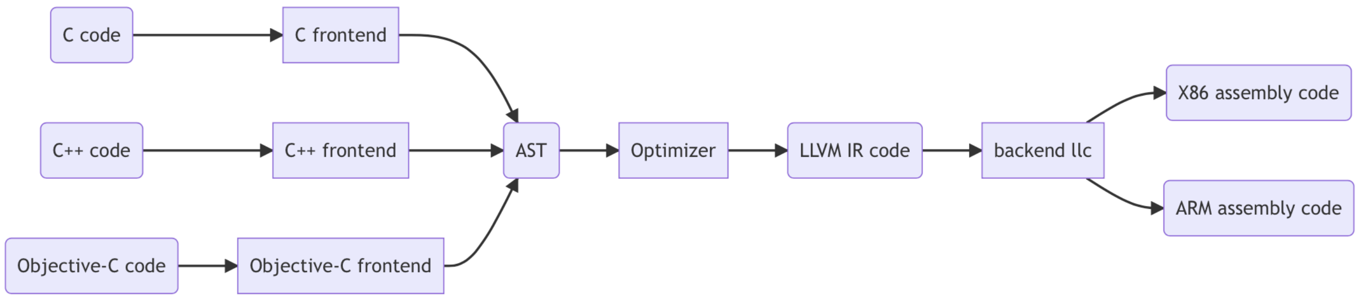
1. Performance optimization requires too much time and resources (today's assembly language is much more complex than before. The classic PDP-11 manual only has less than 30 pages of instructions, but now the Intel X86 instruction manual only has 2500 pages.);

2. Development for each machine platforms is expensive (for example, compiling the same program on ARM and X86);
3. The compiler's "plug-ins" are not enough (sometimes new optimization or processing is needed).

Now you meybe make sense that "front end" here means the front end of the compilation process of the entire C language family, not the front end of a compiler. So clang is a complete compiler that can convert `.c` into `.s` files, but can call the assembler and compiler to generate the final executable file.

As a compiler, clang can convert the C family language into LLVM IR (a low-level language), then convert and output a `.s` file, and then call the assembler in the LLVM project (or other assembler) to convert it into a `.o` object file (this is the "assembly stage" mentioned above), and finally call the linker to link and output an executable program.

The internal process of Clang as following, and is also omitted here:



The assembler and linker used by `clang` can be from LLVM or GNU. For example, the linker can use `lld` from LLVM, or use `ld` or `gold` from GNU, and MSVC's `link.exe` in Windows. By default, it uses LLVM.

If you are curious about the more detailed Clang workflow and the operations of each step, such as what options to process some step in the compilation process, you can read this document "An Overview of Clang".

This compilation method is very convenient for adapting to different platforms. When developing for new platform, just map the machine instructions to the LLVM IR. There is no need for developers to write a new optimizer and code generator to convert the source code into assembly code, saving time and effort.

Why is `clang` a frontend? Isn't it a complete compiler? If `clang` is a complete compiler, why is it called a front-end? If it's not complete, what is the backend?

Clang is a complete compiler and also is a front-end. It's just a front-end that converts source code into an executable program flow, not a compiler's front-end. If it points front end of compiler, it is composed of the preprocessor, lexical analyzer (Lexer), and syntax analyzer (Parser).

The backend corresponding to `clang` is assembler, linker, and other tools included in LLVM or software groups such as GNU. These tools are responsible for assembling and linking the assembly code into the final executable file.

What is the definition of a compiler? I feel that the definition of compiler in the book is different from the actual `gcc`

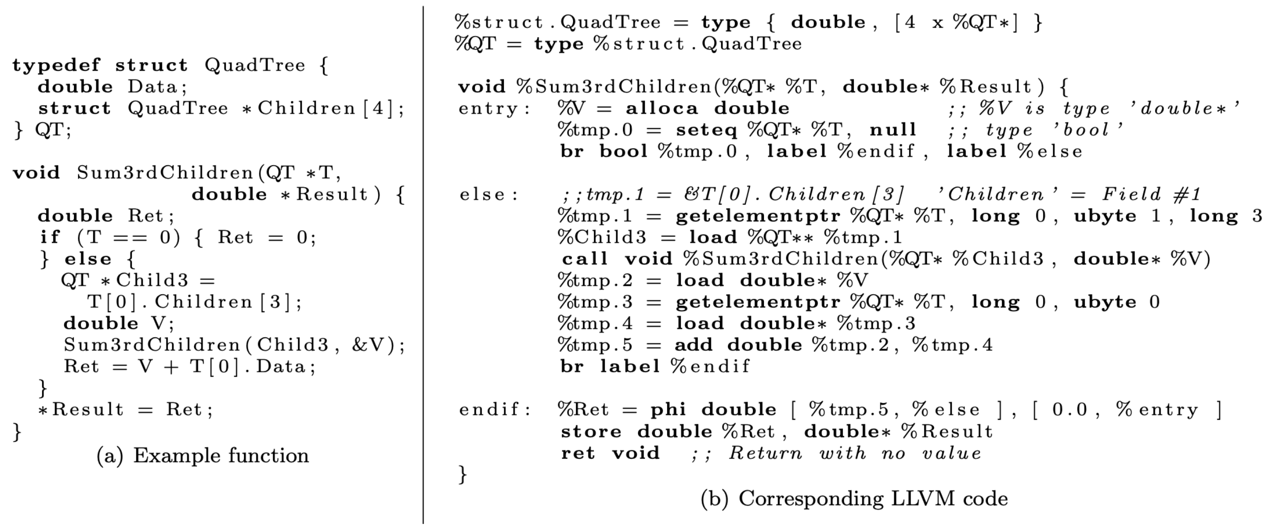
The definition of a compiler has been explained in detail in the previous. Now, generally speaking, "compiler" refers to a program that converts files such as `.c` into `.s` files.

In fact, compilers, such as `gcc`, include some tools (such as preprocessors) and also call other tools (assembler and linker), so they are different from the definition.

What is the LLVM project?

As mentioned above, many compilers require multiple intermediate expressions (IR). These intermediate expressions may be generated by lexical analyzers or semantic analyzers, which are inconsistent. It leads to update instructions and optimization becoming very difficult.

LLVM's full name is "Low-Level Virtual Machine", which is an implementation of architecture and intermediate expression. Originally, LLVM project was a set of tools surrounding LLVM code. The C language and corresponding LLVM code are as follows (Image from Chris Lattner's "Architecture for a Next-Generation GCC"):



```c
typedef struct QuadTree {
  double Data;
  struct QuadTree *Children[4];
} QT;

void Sum3rdChildren(QT *T,
                    double *Result) {
  double Ret;
  if (T == 0) { Ret = 0;
  } else {
    QT *Child3 =
      T[0].Children[3];
    double V;
    Sum3rdChildren(Child3, &V);
    Ret = V + T[0].Data;
  }
  *Result = Ret;
}
```

(a) Example function

```
%struct.QuadTree = type { double, [4 x %QT*] }
%QT = type %struct.QuadTree

void %Sum3rdChildren(%QT* %T, double* %Result) {
entry:  %V = alloca double                  ;; %V is type 'double*'
        %tmp.0 = seteq %QT* %T, null        ;; type 'bool'
        br bool %tmp.0, label %endif, label %else

else:   ;;tmp.1 = &T[0].Children[3]  'Children' = Field #1
        %tmp.1 = getelementptr %QT* %T, long 0, ubyte 1, long 3
        %Child3 = load %QT** %tmp.1
        call void %Sum3rdChildren(%QT* %Child3, double* %V)
        %tmp.2 = load double* %V
        %tmp.3 = getelementptr %QT* %T, long 0, ubyte 0
        %tmp.4 = load double* %tmp.3
        %tmp.5 = add double %tmp.2, %tmp.4
        br label %endif

endif:  %Ret = phi double [ %tmp.5, %else ], [ 0.0, %entry ]
        store double %Ret, double* %Result
        ret void  ;; Return with no value
}
```

(b) Corresponding LLVM code

Figure 2: C and LLVM code for a function

LLVM code have three usages:

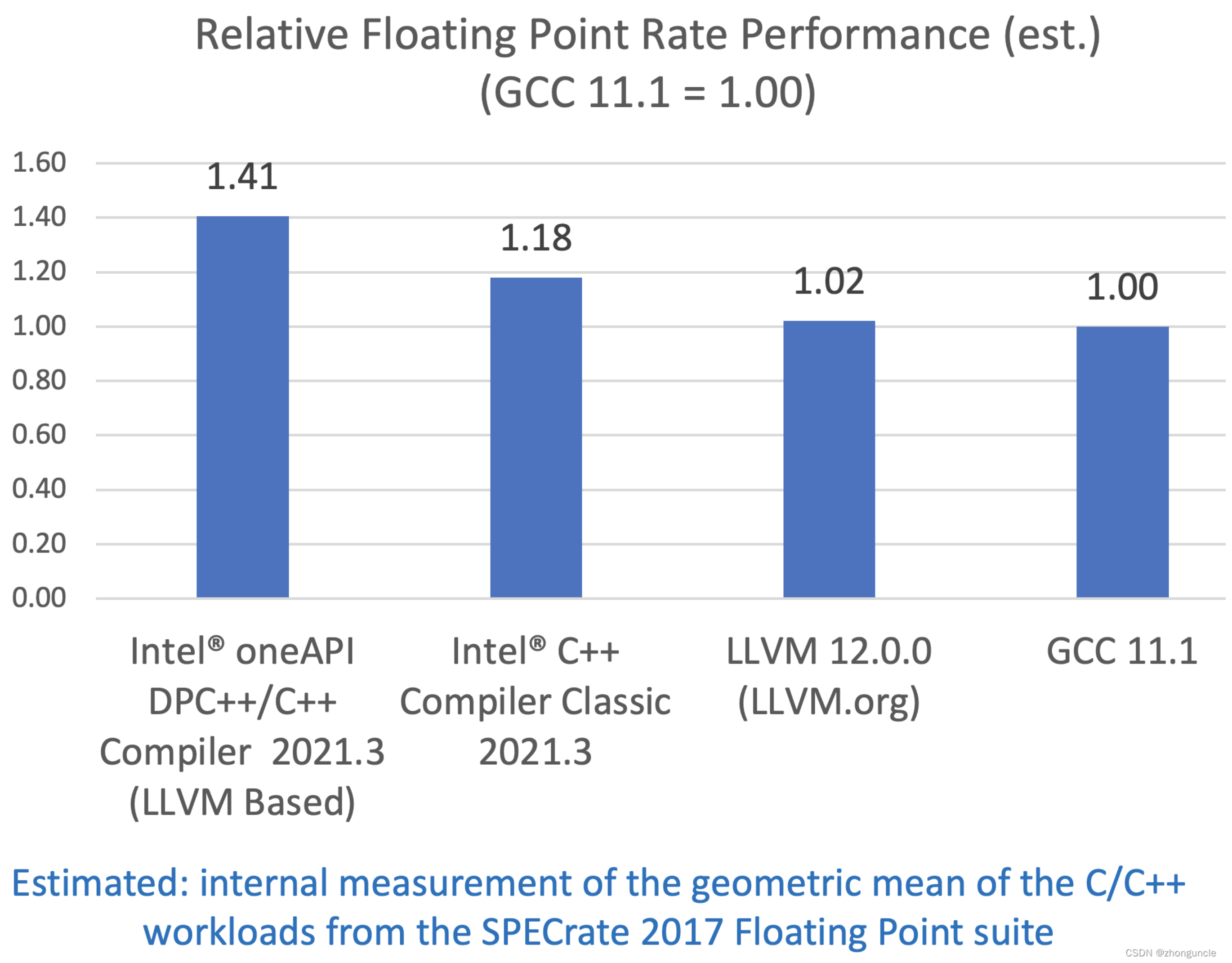
1. Intermediate expression of the compiler;
2. Bitcode stored in the hard disk;
3. Assembly language expression for human reading

These three usages are actually equivalent. They can either directly use, or have tools can convert them easily. This makes LLVM compatible with new machines, optimizing performance, developing new languages, and even disassembly much easier.

The core of the entire LLVM project is LLVM IR. The LLVM IR is intended to be a "universal IR", low-level enough that high-level code can be cleanly mapped to the LLVM IR (similar to how instructions used by processors are "universal IRs", allowing many different languages to be mapped to these assembly language). This brings great performance improvements to compilers using LLVM IR.

For a more detailed introduction to LLVM design, please refer to the document: "LLVM Language Reference Manual".

About the performance improvements brought by LLVM, you can read this article from Intel: "Intel® C/C++ Compilers Complete Adoption of LLVM"



What is the difference between gcc and clang?

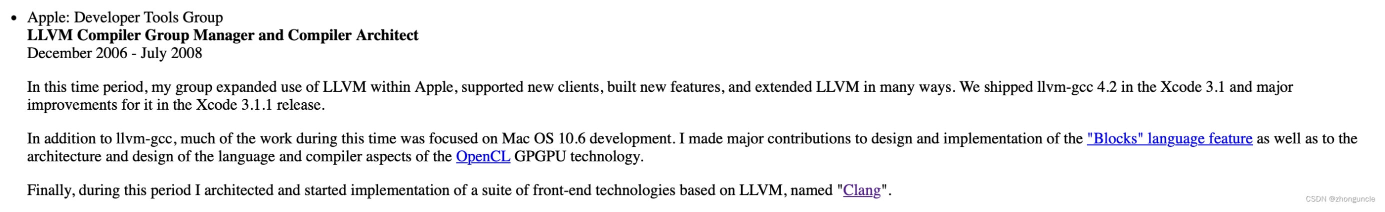
In the early days of LLVM, it starts a project called `llvm-gcc`. The biggest difference between it and GNU GCC is that: **`llvm-gcc` uses LLVM IR as the lowest-level intermediate expression at the end of the compiler, not GNU GCC uses RTL as the lowest level intermediate expression, so the last part of the `llvm-gcc` deals with LLVM IR instead of RTL (Register Transfer Language).**

In other respects, `llvm-gcc` will output an assembly file like `gcc`. However, you can have `llvm-gcc` output LLVM bytecode by using the `-emit-llvm` option.

LLVM founder Chris Lattner created a C language family compiler that uses LLVM as an intermediate expression for all intermediate expressions at Apple, that is Clang.

Although `clang` has eliminated `llvm-gcc`, but `llvm-gcc` still live now, though the usage and performance are not as good as `clang`.

Here is the part of Chris Lattner's resume that mentions the birth of Clang (https://www.nondot.org/sabre/Resume.html#Apple):



- Apple: Developer Tools Group
  LLVM Compiler Group Manager and Compiler Architect
  December 2006 - July 2008

  In this time period, my group expanded use of LLVM within Apple, supported new clients, built new features, and extended LLVM in many ways. We shipped llvm-gcc 4.2 in the Xcode 3.1 and major improvements for it in the Xcode 3.1.1 release.

  In addition to llvm-gcc, much of the work during this time was focused on Mac OS 10.6 development. I made major contributions to design and implementation of the "Blocks" language feature as well as to the architecture and design of the language and compiler aspects of the OpenCL GPGPU technology.

  Finally, during this period I architected and started implementation of a suite of front-end technologies based on LLVM, named "Clang".

As a conclusion, the difference between `gcc` and `clang` is: intermediate layers of `clang` is in LLVM IR, while intermediate layers of `gcc` is TRL or some other things.

Notice: the `llvm-gcc` is not the compiler discussed in LLVM founder Chris Lattner's paper "Architecture for a Next-Generation GCC". This LLVM compiler also is not the same thing as Clang. The diagram of this LLVM compiler in the paper is as follows:

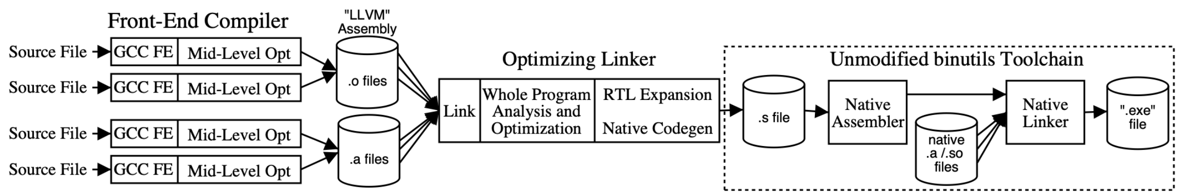
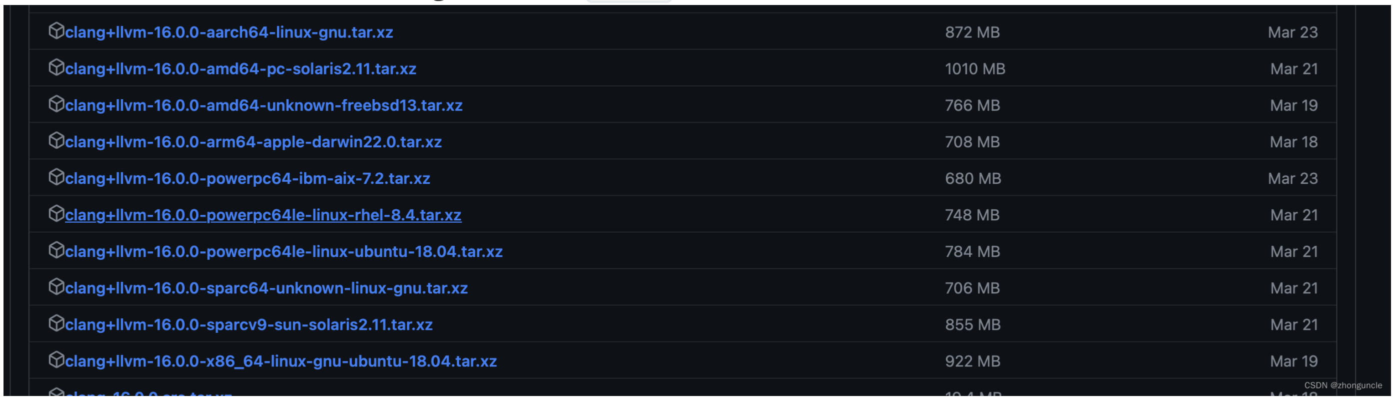


Figure 1: High-Level Compiler Architecture for Whole-Program Optimization

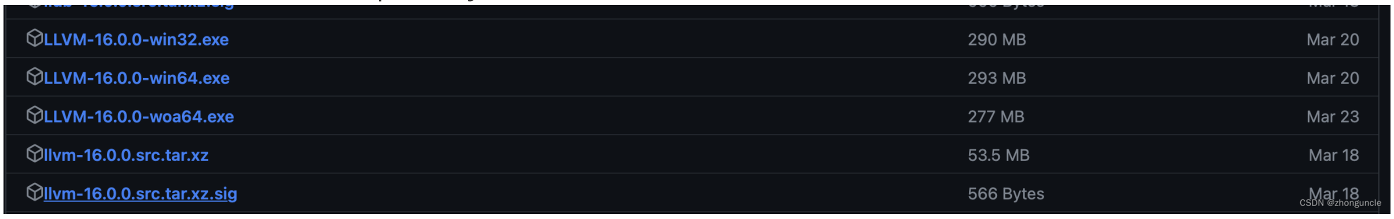
The difference is that a link layer is added in the middle, and happens two links in the entire compilation process. But according to Intel's data, it is obvious that performance and effect of the LLVM compiler are similar to those of GNU GCC. Now you still can download it on GitHub. The latest version is 16: https://github.com/llvm/llvm-project/releases/tag/llvmorg-16.0.0

You can choose to download it together with `clang`:



clang+llvm-16.0.0-aarch64-linux-gnu.tar.xz	872 MB	Mar 23
clang+llvm-16.0.0-amd64-pc-solaris2.11.tar.xz	1010 MB	Mar 21
clang+llvm-16.0.0-amd64-unknown-freebsd13.tar.xz	766 MB	Mar 19
clang+llvm-16.0.0-arm64-apple-darwin22.0.tar.xz	708 MB	Mar 18
clang+llvm-16.0.0-powerpc64-ibm-aix-7.2.tar.xz	680 MB	Mar 23
clang+llvm-16.0.0-powerpc64le-linux-rhel-8.4.tar.xz	748 MB	Mar 21
clang+llvm-16.0.0-powerpc64le-linux-ubuntu-18.04.tar.xz	784 MB	Mar 21
clang+llvm-16.0.0-sparc64-unknown-linux-gnu.tar.xz	706 MB	Mar 21
clang+llvm-16.0.0-sparcv9-sun-solaris2.11.tar.xz	855 MB	Mar 21
clang+llvm-16.0.0-x86_64-linux-gnu-ubuntu-18.04.tar.xz	922 MB	Mar 19

It can also be downloaded separately:



LLVM-16.0.0-win32.exe	290 MB	Mar 20
LLVM-16.0.0-win64.exe	293 MB	Mar 20
LLVM-16.0.0-woa64.exe	277 MB	Mar 23
llvm-16.0.0.src.tar.xz	53.5 MB	Mar 18
llvm-16.0.0.src.tar.xz.sig	566 Bytes	Mar 18

When wrote this blog, I have a deeper understanding of the `gcc` and `clang` compilers and usages. However, since this article is too long, mistakes is inevitable. If you find any mistakes while reading (mistakes, typos, things I forgot to delete, etc.), please leave a comment and let me know. Thanks!

I hope these will help someone in need~