VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Sampo Hytönen

Replacing GCC compiler with Clang/LLVM

Technology
2018

VAASAN AMMATTIKORKEAKOULU
Tietotekniikka

TIIVISTELMÄ

Tekijä	Sampo Hytönen
Opinnäytetyön nimi	GCC kääntäjän korvaaminen Clang/LLVM kääntäjällä
Vuosi	2018
Kieli	englanti
Sivumäärä	57
Ohjaaja	Timo Kankaanpää

Tämä on tutkimustyö, joka keskittyy tutkimaan mitä hyötyä yritykselle antaisi siirtyä käyttämään GCC kääntäjän sijasta Clang kääntäjää. Työ on toteutettu ensin vertailemalla kääntäjiä ja tutkimalla, mitkä asiat on syytä ottaa huomioon, kun kääntäjää ollaan vaihtamassa. Viimeiseksi, yksi yrityksen projekteista käännetään Clang kääntäjällä, jotta nähdään, löytääkö uusi kääntäjä projektista virheitä tai parannusehdotuksia.

Ensimmäisessä osassa yritys ja työn tausta esitellään.

Toisessa osassa kerrotaan käännösprosessin teoriasta ja kääntäjän toimintaperiaatteista.

Kolmannessa osassa selitetään työssä käytetyt teknologiat.

Neljännessä osassa kääntäjiä vertaillaan.

Viides osa keskittyy siihen, mitä on syytä ottaa huomioon, kun GCC kääntäjästä Clangiin ollaan siirtymässä ja onko hyötyä käyttää molempia kääntäjiä rinnakkain.

Kuudennessa osassa yksi yrityksen projekteista konfiguroidaan Clangille, ja tulokset tästä prosessista paljastetaan.

Viimeinen osio yhteen vetää projektin.

Avainsanat GCC, Clang, LLVM, kääntäjä, käännösprosessi

VAASA UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Sampo Hytönen
Title	Replacing GCC with Clang/LLVM compiler
Year	2018
Language	English
Pages	57
Name of Supervisor	Timo Kankaanpää

This is a research project focusing on what advantages it would give for a company to change using Clang compiler instead of GCC. It is done by comparing the compilers and finding out what is the necessary things one should consider when changing a compiler. Lastly, one of the company's own projects is compiled with Clang to find out will it find out new bugs or suggestions for improvement from the code.

In the first section, the company and the purpose of the project are presented.

In the second section, the theory of the compilation process and the principles of the compiler design are explained.

The third section introduces to the technologies used in the project.

In the fourth section comparison of the compilers are done.

The fifth section focuses on what one should consider when migrating from GCC to Clang and would there be any advantages to use both compilers in parallel.

In the sixth section one of the company's own projects is configured for Clang and the results of the process are revealed.

The last section concludes the project.

Keywords GCC, Clang, LLVM, compiler, compilation process

CONTENTS

LIST OF FIGURES AND TABLES

1 INTRODUCTION

This study project is done for Wärtsilä Oyj on behalf of student of Vaasa University of Applied Sciences (VAMK). The project focuses to research what advantages it would give for the company to change using Clang compiler instead of GCC. It also describes what steps are needed to take in order to get the Clang compiler work and what kind of problems may occur during the process.

1.1 Wärtsilä Oyj

Wärtsilä is a Finnish corporation and one of the global leaders in the marine and energy markets. It is founded in 1834 in Finland. It employs approximately 18,000 employees in more than 70 countries and over 200 locations around the world. Wärtsilä is listed on Nasdaq Helsinki and its net sales was 4.8 billion in 2016. /52/

2 BASICS OF THE COMPILATION PROCESS

A computer is a sophisticated system that combines software and hardware seamlessly together. Hardware understands only electronic signals that are routed through semiconductor circuits to do some calculations and store them in a memory. These signals are converted into binary language used in software programming. They are simply a set of ones and zeros. For human it would be inconvenient to program using binary code and due to that, higher-level languages such as C programming language, have been created to write more complex programs. Therefore, compilers are needed. They are a set of tools and operating system components used to get the desired binary code that can be run on hardware to do the things a programmer wants.

In a compilation process, a human written code is translated to machine-readable commands. *The language processing system* is a set of various tools used in compilation process. It takes care of several tasks such as making code more efficient by optimizing it. Compiler also finds errors from code and prevents a user to run faulty programs. It warns about defects and mistakes a programmer may have written. For example, if a programmer has created a variable that is not used in the code, a compiler warns about it if this functionality is allowed in the compiler.

2.1 The Main Components of the Compilation Process

Usually, when referring to a compiler, one means the set of tools of the whole compilation process. In more detail, a compiler is just one piece in the entire process of compilation. There is a picture below (Figure 1) to depict the language processing system.

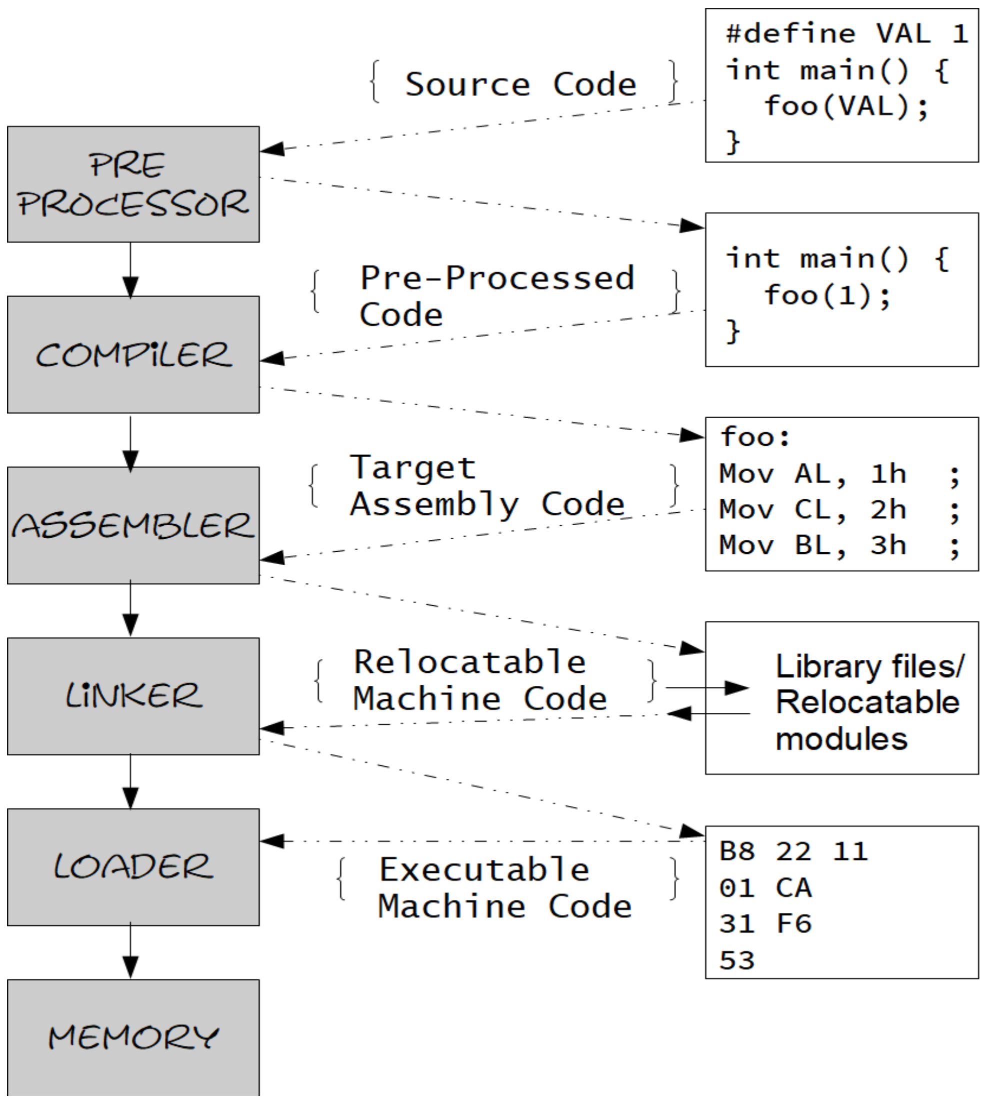


Figure 1 *The Language Processing System*

2.1.1 Preprocessor

Preprocessor prepares code for the actual compiler. For instance, it takes care of file inclusions, language extensions and maps predefined values i.e. macros into the code. In the picture above can be seen that preprocessor has mapped the macro definition *VAL* for the function call as a parameter. The tasks preprocessor needs to take care of varies between different preprocessors. /2/

2.1.2 Compiler

A compiler translates source code written in one language to target code that can be the same or some other language. Generally, it is used to translate high-level language like C to low-level symbolic machine language i.e. assembly language. It gets pre-processed code from a preprocessor gives its output for an assembler. /1, 2/

There are many kinds of compilers. Compilers that compile source code, which is run on different operating system or hardware, are called *cross-compilers*. *Source-to-source* compilers translate between two high-level languages. A compiler that does not change the language of its source code is called a *bootstrap compiler*. There is also *decompilers* to compile from a low-level to a high-level language. /1, 2/

Some languages such as JavaScript or Python use *interpreters* instead of compilers. While compilers read the whole source code and translate it at once, an interpreter reads and executes it in segments. /1, 2/

2.1.3 Assembler

Assembler translates assembly language or *symbolic machine code* in other words, to *machine code*. Machine code is a series of sequential machine-readable commands in a binary form to be stored in device's memory. /2/

2.1.4 Linker

Linker finishes an executable file by including external libraries and modules consisting of one or many object files translated to machine code into compiled code. Those object files can be linked dynamically or statically. In static linking a linker combines all the object files in one executable file while in dynamic linking the external object files are combined on runtime. Linker also determines where to store codes and references in a memory. The Figure 2 below depicts how the linker works. /2/

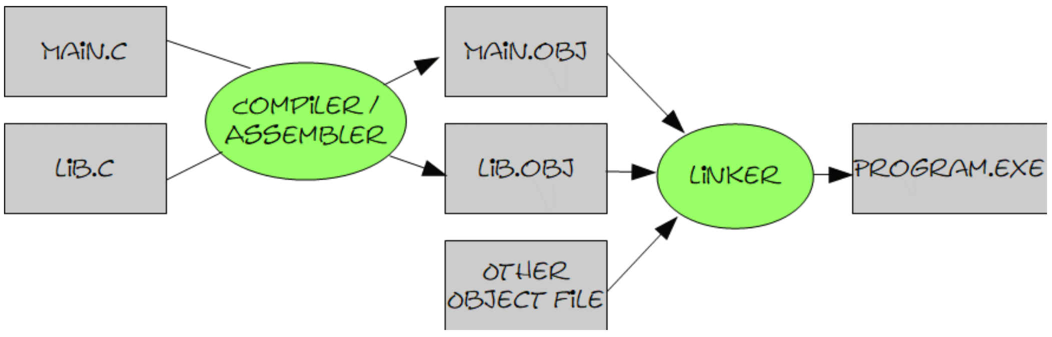


Figure 2 *Linker*

2.1.5 Loader

Loader calculates the size of a program allocates memory for it and loads it on the allocated memory. It also handles the various registers used in the program and links the program with the dynamically allocated libraries it needs. /2/

2.2 The Architecture of Compiler Design

A compiler is divided into several phases and those phases can be split to front-end, also known as *analysis phase*, and *synthesis phase* that refers to back-end. There is an *intermediate code generation* between the front- and the back-end. The phases are run in sequence and each phase get its input from an output of the previous phase. The way phases are implemented differs lightly between compilers, but in general, the architecture consists of the following phases (Figure 3). /1, 2/

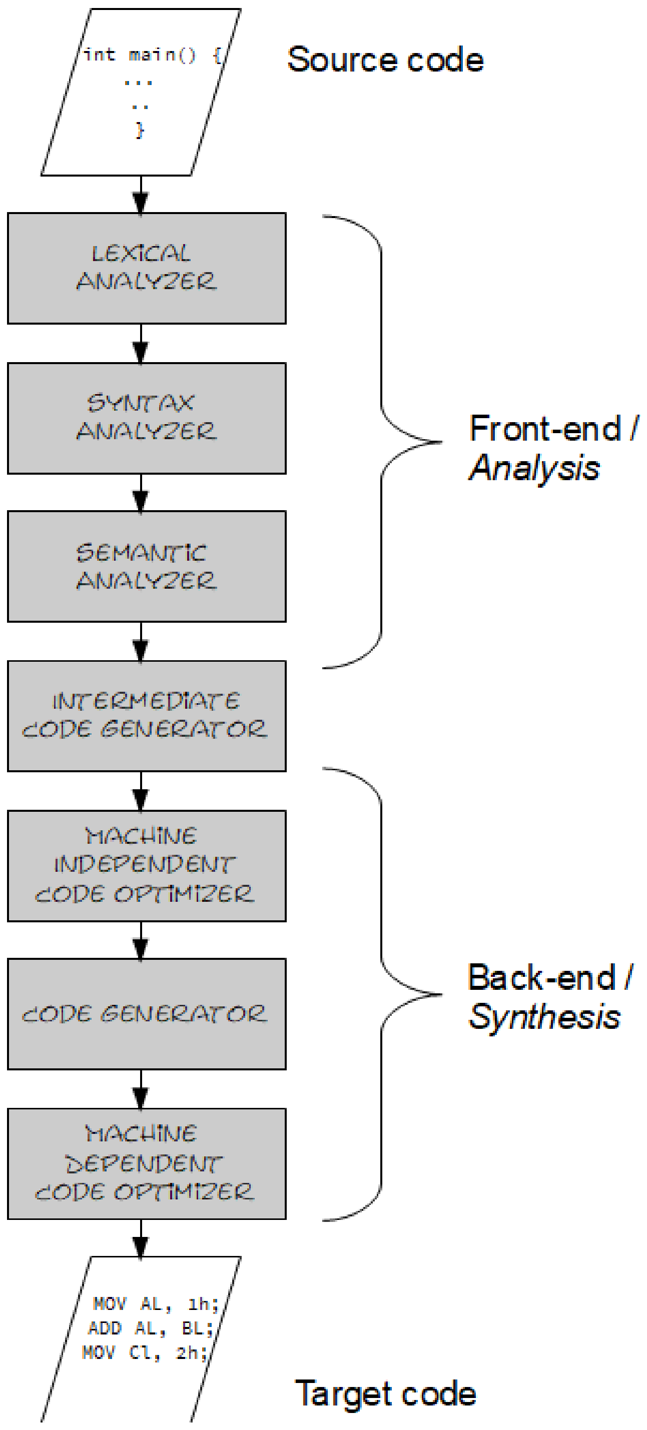


Figure 3 *Compiler Design Architecture*

2.2.1 Lexical Analysis

This is the first phase where code is read and divided into lexemes or *tokens*. To give an example, there is a variable declaration below written in C programming language and it is divided into tokens (Table 1). /1, 2/

```
int value = 50;
```

Table 1 *Tokens*

int	Keyword
value	Identifier
=	Operator
50	Constant
;	Symbol

2.2.2 Syntax Analysis

Syntax analyzer takes a list of tokens and generates a *syntax tree* of them. It also checks the expressions made of tokens for syntax errors. However, if a variable above was declared to store a string type for an integer variable, syntax analyzer would not throw an error for that. The following declaration would pass this phase: /1, 2/

```
int value = “Hello World”;
```

2.2.3 Semantic Analysis

Semantic analyzer runs type checking for given expressions. It ensures that variables are declared before using them, and that a program will not accept a string or boolean types for an integer variable and so on. /1, 2/

2.2.4 Intermediate Code Generation

Intermediate code is a language between target and source code. The benefits of using it are to make a compiler more generic. It allows one code to be compiled for many machine architectures by changing the back-end of the compiler. For the same reason, it makes it easier to compile multiple high-level languages by changing only the front-end. It also provides for using an interpreter by using a small program written in machine code instead of translating the whole code into target code. /1, 2/

2.2.5 Machine Independent Code Optimization

In the code optimization phase, the speed of code is improved, and it is made to consume less resources. It is done for instance, by deleting extra code lines and arranging the sequence of statements to speed up the program without changing the program behavior. /2/

2.2.6 Code Generation

Code generator generates target code of intermediate code. Typically, it is assembly language for a specific machine architecture. /1, 2/

2.2.7 Machine Dependent Code Optimization

In a similar way as in the Machine Independent Code Optimization phase, machine code given is optimized to run more effectively on a target device. In this phase, the high-level programming language is replaced by efficient low-level code. /1, 2/

2.3 Automating the Compilation Process

It would take a lot of time for programmers to use a compiler from command line and deal with all the relationships. To ease this job, there are tools to automate the routine of compiling source files into an executable. The most common tool is called *GNU Make*. It is a part of the larger GNU project, but the tool itself is compiler independent. /4/

GNU Make defines a language for dealing with relationships between source code, intermediate files and executables. It can be used also for managing alternate configurations, implementing reusable libraries, and parameterizing a process with macros defined by user. /4/

3 USED TECHNOLOGIES

This section introduces to the main technologies used in the project. The technologies can be combined in three sections. First is the GNU toolchain that combines the tools needed to compile with GCC. The LLVM project includes tools needed to compile with Clang. In the third section, there is CMake, that is used to configure larger projects.

3.1 GNU Toolchain

GNU Toolchain is developed to be a part of the larger GNU Project that is started in 1984 by Richard Stallman to provide a complete Unix-like operating system as free software. The toolchain is a collection of programs such as compiler, assembler and linker, aiming to develop other software and operating systems. It has been playing a vital role developing embedded systems software, and as an example, the Linux kernel.

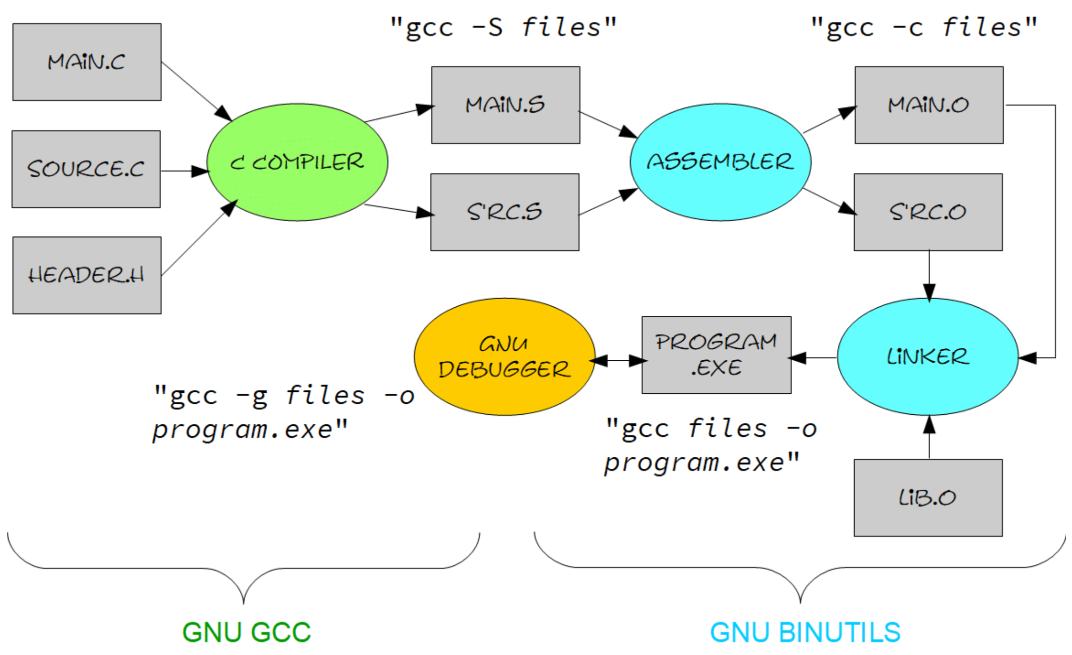


Figure 4 *GNU Toolchain*

3.1.1 GNU GCC

GNU Compiler Collection is a toolset for pre-processing and compiling code. GCC also invokes GNU Binutils that generates machine code of an output of the compiler. /3/

Previously GCC was defined as GNU C Compiler as it was developed to be only a C compiler but changed after it has extended for other languages as well. One can also write his or her own frontend for GCC to use it with a language not yet supported. GCC is a portable compiler that can be run on almost every device nowadays and enables also cross-compiling that is used widely with many kinds of embedded devices. /3/

3.1.2 GNU Binutils

GNU Binutils is set of binary tools for assembling and linking compiled code into an executable binary file. The main programs are GNU linker *ld/gold* and GNU assembler *as*, but it also comprises tools, for instance, to handle a symbol table list and to build libraries. /5/

3.1.3 GNU Debugger

GNU Debugger *GDB* is a mature debugger for programs compiled with GCC. It has a capability to stop a single thread at a break point. GDB also supports remote debugging that can be used, for instance, to run the debugger in a more powerful system. /6/

3.1.4 GNU Build System

While GNU Make’s purpose is to automate the compilation process, GNU Build System consists of tools designed to automate distribution of software for many different platforms. Makefiles need to be different for every different platform and GNU Build System, also known as *autotools,* is to generate proper makefiles for them. The main components of autotools are *autoconf* and *automake.* Autoconf is to create a configuration for automake that generates makefiles of these files. /7/

3.2 LLVM

The LLVM is an umbrella project consisting of modular compiler and toolchain technologies used for compilation process, similar to GNU Toolchain. The LLVM project is started in 2000 in University of Illinois by Professor Vikram Adve and first year graduate Chris Lattner. The original goal was to investigate compilation techniques that support both dynamic and static programming languages. The original acronym stood for *Low-Level Virtual Machine* but after the project grew and spread widely, the acronym was removed as it became misleading. /8/

LLVM differs from GCC by the way it is designed. While GCC is a complicated static compiler that is difficult for new developers to grasp, LLVM’s architecture is designed for reusable libraries with well-defined interfaces. It can be used as a static or a runtime compiler. /9/

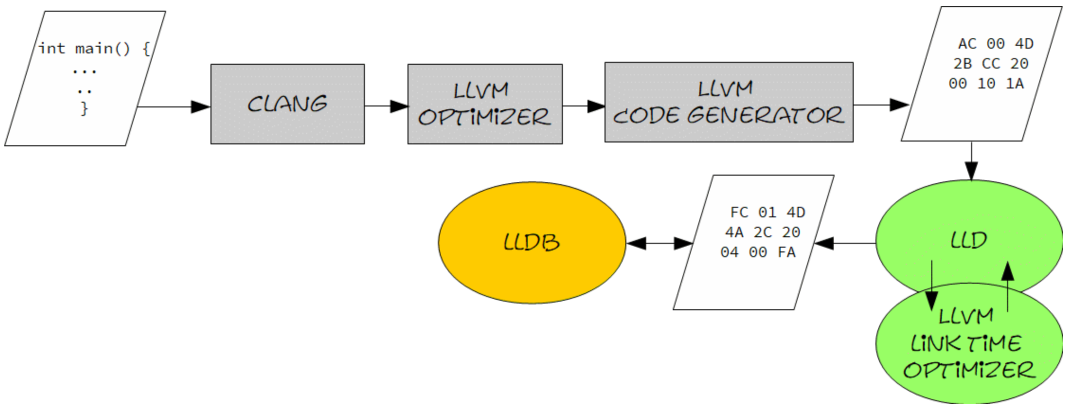


Figure 5 *Clang/LLVM Toolchaing*

3.2.1 LLVM Core

LLVM Core represents the intermediate code generator between source code and target code. In this phase, source code is translated into machine independent form *LLVM IR (Intermediate Representation).* LLVM Core libraries also includes a machine independent code optimizer. /8, 9/

3.2.2 LLVM IR

LLVM IR is the machine independent code between source code and target code. It is generated by LLVM Core libraries. /8, 9/

3.2.3 Clang

Clang is a modular frontend for LLVM that supports C, C++ and Objective C languages. It is claimed that it can be significantly faster in comparison to GCC. It is designed as a drop-in replacement for GCC. In practical it means that many of the same command line options can be used what was used with GCC. /8, 10/

Clang aims to be user friendly by providing expressive diagnostics about warnings and errors. It includes a static analyzer that finds bugs from source code. /8, 10/

3.2.4 DragonEgg and llvm-gcc

In the early stages of the LLVM project development, there was no frontend implemented for it. Thus, the GCC frontend was used with the LLVM backend. DragonEgg is the newest tool to use GCC frontend with LLVM backend. Using it may facilitate the migration from GCC to LLVM in a context where Clang frontend cannot be used. /15/

DragonEgg has not had a lot of attention from developers in the recent years, but it works with newest version of LLVM and GCC. Older versions of GCC can be used with *llvm-gcc* that is a project focusing on the same goal. Llvm-gcc is not supported on the newer versions of GCC though. /15, 16/

3.2.5 lld

Lld is a linker for LLVM. It is claimed to be more than twice faster than GNU linker. /11/

3.2.6 LLDB

LLDB is a debugger provided by LLVM and Clang. It supports C, C++ and Objective C languages. LLDB is claimed to use memory more effectively compared to GDB. LLDB supports also remote debugging but lacks the ability to stop a single thread at a break point. /12, 39/

3.2.7 LLVM Link Time Optimizer

LLVM supports Link Time Optimization (LTO) that is intermodular optimization executed during the link stage. /13/

3.3 CMake

CMake is a set of tools to automate a distribution of a program for different platforms. It is licensed with a BSD-3 open source license and works on multiple platforms. Similar to GNU Build System, it generates Makefiles, but can be used also for testing purposes. /14/

CMake is used by creating a *CMakeLists.txt* file that is written using *cmake-language*. After that, calling *cmake <path/to/CMakeLists.txt>* will generate files needed to build the application. /14/

4 COMPARISON OF THE COMPILERS

In basic usage, both compilers work similarly and there are no big differences between them. This means that most of the same command line parameters can be used and the output is also similar. There are some differences though that will be looked more closely in this chapter.

GNU GCC has been the standard compiler for many Unix-like operating systems, but later some of them such as FreeBSD and macOS have changed to use LLVM instead. /35, 48/

At the moment, future seems promising for LLVM as it is supported by Apple. GCC on the other hand, has a huge user population and it is easy to get help with it. From the graph of active developers (Figure 6) can be seen that Clang and LLVM (the blue lines) are getting more active authors all the time, while GCC (the green line) is being steady where it has been since 2004. This means that GCC is not going to die for a long time albeit Clang and LLVM has been more attractive for new developers. /31, 40/

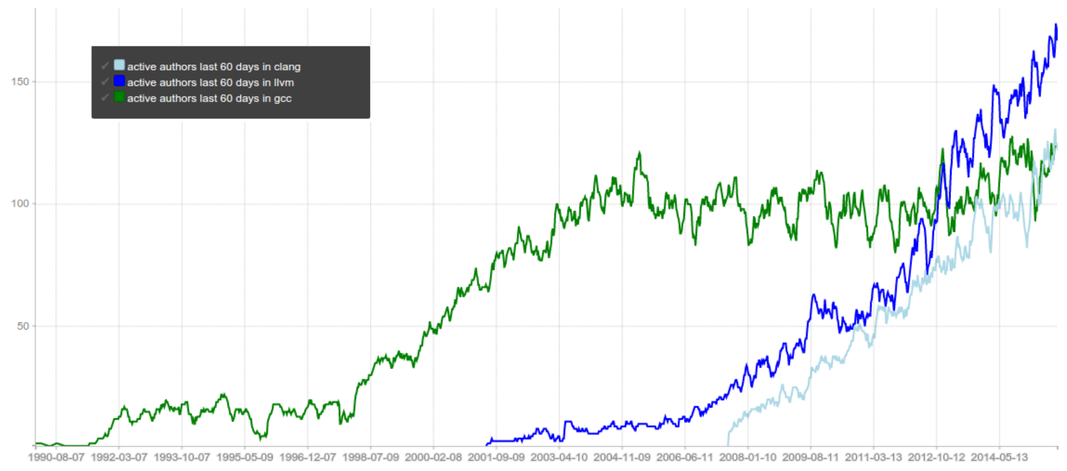


Figure 6 *Active Developers of the Compiler Repositories /40/*

4.1 Usability

This section focuses on what languages and target architectures are supported. It also compares features, diagnostics and differences between licensing of the compilers.

4.1.1 Supported Languages

Both of the compilers support a wide variety of languages. While GCC is a static compiler, LLVM supports also runtime compilations. However, GCC frontend can be used with some of the interpreted languages such as Java. /9, 17/

The table below (Table 2) describes the supported languages of the compilers. There may be some third-party libraries that support other languages as well but are not in the table. LLVM supports also many lesser known languages that are not in the table.

Table 2 *Supported languages by GCC and LLVM /9, 17, 18/*

Language	**GCC**	**LLVM**
C	Y	Y
C++	Y	Y
Objective-C	Y	Y
Objective-C++	Y	Y

Fortran	Y	Y
Java	Y	Y
Ada (GNAT)	Y	Y
Go	Y	Y
Pascal	Y	Y
Mercury	Y	Y
COBOL	Y	Y
Ruby	N	Y
Python	N	Y
Haskell	N	Y
D	N	Y
PHP	N	Y
Pure	N	Y
Lua	N	Y
Rust	N	Y

4.1.2 Supported Target Architectures

GCC supports wider variety of supported target architectures. The detailed list can be found from the references. LLVM supports the same basic architectures, such as *X86, X86-64, ARM, ARM64, AARCH64, PPC64, PPC32, XCORE* and many more. /19, 20/

The question what target architectures are supported becomes relevant if one is working with an unusual embedded architecture. In that case, GCC may be the only option to choose.

4.1.3 Diagnostics

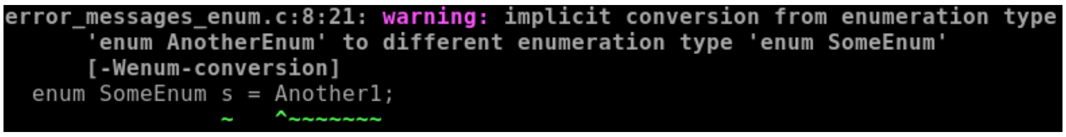
Clang claims to be a frontend that has better diagnostics for error and warning messages. Whether it is true or not will become clearer in this chapter. /22, 23/

Clang has colored and highlighted output diagnostics by default. GCC also supports colored output for versions 4.9 and later, but requires that *-fdiagnostics-color=[auto|never|always]* flag is given. /22/

There are examples below to show more details about the differences. The examples are run with GCC version 7.0 (Released in May 2017) and Clang version 5.0 (Released in September 2017). The first example is about implicit enumeration conversion. Clang warns of implicit enumeration conversions by default, but GCC does not, even if all warnings are enabled (Figure 7).

```
enum SomeEnum    { Some1    = 0, Some2    = 1 };
enum AnotherEnum { Another1 = 0, Another2 = 1 };

int main()
{
  enum SomeEnum s = Another1;
  return 0;
}
```



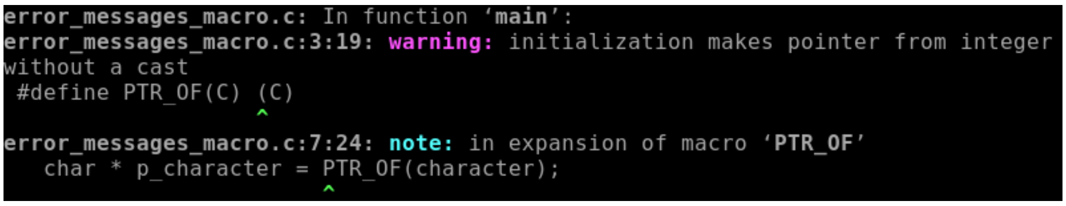
```
error_messages_enum.c:8:21: warning: implicit conversion from enumeration type
      'enum AnotherEnum' to different enumeration type 'enum SomeEnum'
      [-Wenum-conversion]
  enum SomeEnum s = Another1;
                ~   ^~~~~~~~
```

Figure 7 *Clang warning of implicit enumeration conversion*

Another example of error messages is faulty macro definition. Both compilers give clear diagnostics, but Clang output also gives an advice what is needed to do in order to correct the mistake (Figure 8 and Figure 9). This is an advantage for Clang as it speeds up the work of error correction.

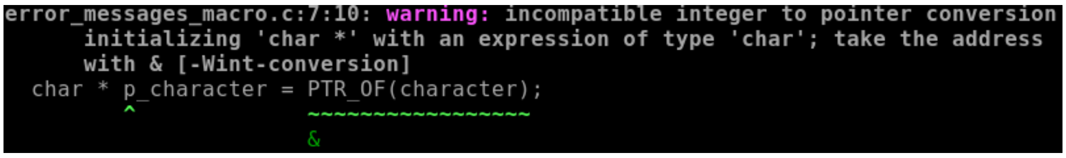
```
#define PTR_OF(C) (C)

int main()
{
  char character;
  char * p_character = PTR_OF(character);
  return 0;
}
```



```
error_messages_macro.c: In function ‘main’:
error_messages_macro.c:3:19: warning: initialization makes pointer from integer
without a cast
 #define PTR_OF(C) (C)
                   ^
error_messages_macro.c:7:24: note: in expansion of macro ‘PTR_OF’
   char * p_character = PTR_OF(character);
                        ^
```

Figure 8 *GCC output of macro expansions error*



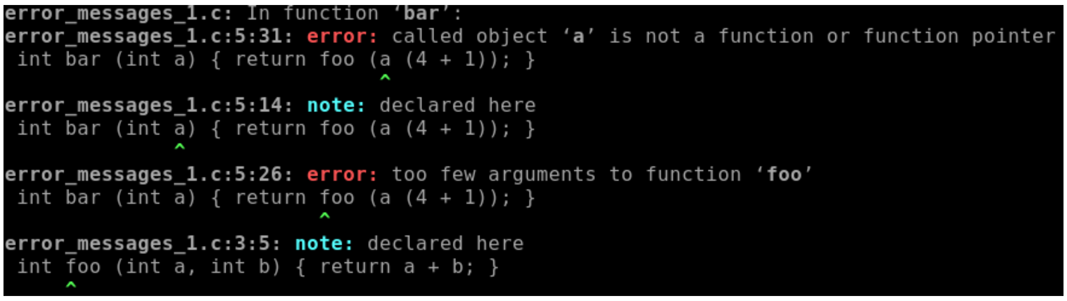
```
error_messages_macro.c:7:10: warning: incompatible integer to pointer conversion
      initializing 'char *' with an expression of type 'char'; take the address
      with & [-Wint-conversion]
  char * p_character = PTR_OF(character);
         ^             ~~~~~~~~~~~~~~~~~
                       &
```

Figure 9 *Clang output of macro expansion error*

The next example has a missing comma in a function declaration. It can be seen that GCC error messages shows clearly that there are too few arguments in function call foo (Figure 10). Clang, on the other hand, gives shorter and clearer output, it points to correct place in the code, but the error message itself is weird (Figure 11).

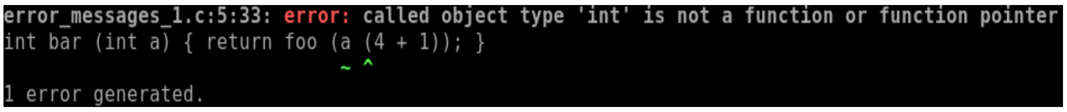
```
int foo (int a, int b) {
    return a + b;
}
int bar (int a) {
    return foo (a (4 + 1));
}

int main()
{
    printf("Result: %d \n", bar(1));
    return 0;
}
```



```
error_messages_1.c: In function ‘bar’:
error_messages_1.c:5:31: error: called object ‘a’ is not a function or function pointer
 int bar (int a) { return foo (a (4 + 1)); }
                               ^
error_messages_1.c:5:14: note: declared here
 int bar (int a) { return foo (a (4 + 1)); }
              ^
error_messages_1.c:5:26: error: too few arguments to function ‘foo’
 int bar (int a) { return foo (a (4 + 1)); }
                          ^
error_messages_1.c:3:5: note: declared here
 int foo (int a, int b) { return a + b; }
     ^
```

Figure 10 *GCC messages of missing comma*



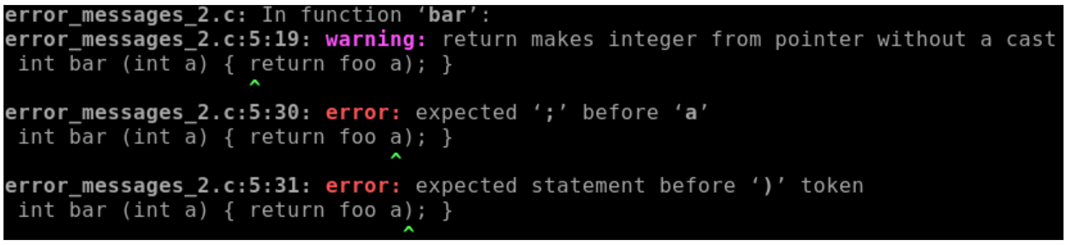
```
error_messages_1.c:5:33: error: called object type 'int' is not a function or function pointer
int bar (int a) { return foo (a (4 + 1)); }
                              ~ ^
1 error generated.
```

Figure 11 Clang *message of missing comma*

In the next example there is a missing opening parenthesis in a function call. Both compilers give quite similar error messages and point to the correct place in the code (Figure 12 and Figure 13). In both cases the message is not very descriptive.

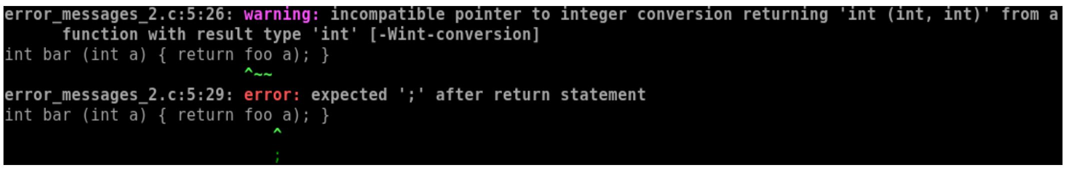
```
int foo (int a, int b) { return a + b;  }
int bar (int a)        { return foo a); }

int main()
{
    printf("Result: %d \n", bar(1));
    return 0;
}
```



```
error_messages_2.c: In function ‘bar’:
error_messages_2.c:5:19: warning: return makes integer from pointer without a cast
 int bar (int a) { return foo a); }
                   ^
error_messages_2.c:5:30: error: expected ‘;’ before ‘a’
 int bar (int a) { return foo a); }
                              ^
error_messages_2.c:5:31: error: expected statement before ‘)’ token
 int bar (int a) { return foo a); }
                               ^
```

Figure 12 *GCC error messages of missing parenthesis*



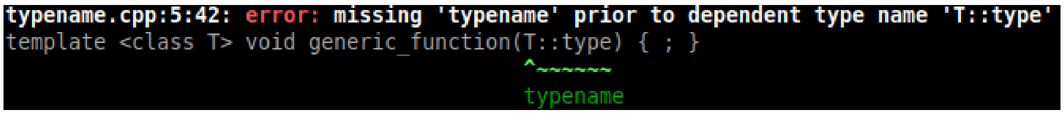
```
error_messages_2.c:5:26: warning: incompatible pointer to integer conversion returning 'int (int, int)' from a
      function with result type 'int' [-Wint-conversion]
int bar (int a) { return foo a); }
                         ^~~
error_messages_2.c:5:29: error: expected ';' after return statement
int bar (int a) { return foo a); }
                            ^
                            ;
```

Figure 13 *Clang error messages of missing parenthesis*

The last example has a missing *typename* word in front of the template function parameters. Clang shows the error clearly (Figure 14), while GCC error message is not as obvious (Figure 15).

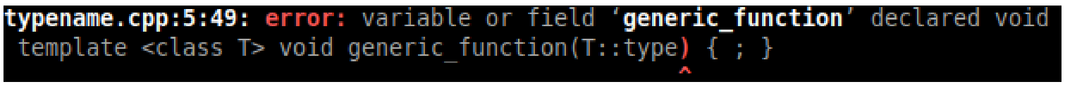
```
template <class T> void generic_function(T::type) { ; }
struct ClassA { };

int main()
{
    ClassA a;
    generic_function<ClassA>(a);
    return 0;
}
```



```
typename.cpp:5:42: error: missing 'typename' prior to dependent type name 'T::type'
template <class T> void generic_function(T::type) { ; }
                                         ^~~~~~~
                                         typename
```

Figure 14 *Clang Output of Missing Typename*



```
typename.cpp:5:49: error: variable or field ‘generic_function’ declared void
 template <class T> void generic_function(T::type) { ; }
                                                 ^
```

Figure 15 *GCC Output of Missing Typename*

Table 3 *Results of Diagnostics Comparison*

Test case	**GCC**	**Clang**
Implicit enumeration conversion	Missing	Good
Faulty macro definition	Ok	Good
Missing comma in a function declaration	Good	Ok
Missing opening parenthesis	Ok	Ok
Missing typename	Ok	Good

According to the Table 3, Clang has slightly better diagnostics in these test suites. In most cases, there are no big differences after all.

4.1.4 Features

GCC is much older and larger project so naturally it has some features that are not yet supported by Clang. Some of them are listed below. These are not in the C standard but extensions in GCC. /21/

- Clang does not support nested functions.
- Variable-length arrays in structures are not supported.
- Clang does not accept some constructs GCC accepts where a constant expression is required. Those are called *fold-expressions*.
- Clang does not support variable types *_Decimal32* for floating point and *_Fract* for fixed-point. /21/

4.1.5 Licensing

Both compilers are published with a license that is free of charge for users. There are some differences though. GCC is licensed with a GPL license which is, according to Free Software Foundation, a free software license that preserves the justice of software users, and it should not be confused with an open source license. GPL requires that every change made to code needs to be revealed public as well. LLVM, in turn, is licensed with a BSD open source license. It allows users to modify it as they want and do not require the modifications to be revealed for others. For most people it does not make a difference which license is used, but for some it may be the decisive question. /8, 37/

“In the free software movement, we campaign for the freedom of the users of computing. The values of free software are fundamentally different from the values of open source, which make "better code" the ultimate goal. The Clang and LLVM developers reach different conclusions from ours because they do not share our values and goals. They object to the measures we have taken to defend freedom because they see the inconvenience of them and do not recognize (or don't care about) the need for them. I would guess they describe their work as "open source" and do not talk about freedom. They have been supported by Apple, the company which hates our freedom so much that its app store for the i-things requires all apps to be non-free. The existence of LLVM is a terrible setback for our community precisely because it is not copylefted and can be used as the basis for non-free compilers. So that all contribution to LLVM directly helps proprietary software as much as it helps us.” -Richard Stallman, the original developer of GCC and the launcher of the GNU project. /37/

The GPL license is better for free software developers who value the ideology over productivity. But for a company doing software development, the restrictions of the license may be obstructing in some cases. For example, in the case the company’s developers would want to create some enhancements or extensions into GCC, it is too time consuming to release the new version of GCC for the free software community. In these cases, the LLVM licensing would be a better option.

4.2 Compilation Time

In this section, compilation times are compared. There are tables below (Figure 16 and Figure 17) to depict the average proportional compilation time. The values are not seconds, but a percentage value of the difference between compile times of the compilers. Calculations are done by taking proportional values of compilation times of every test in a test suite. The average proportional value is then calculated by adding up the values and dividing the sum with the number of tests in a test suite.

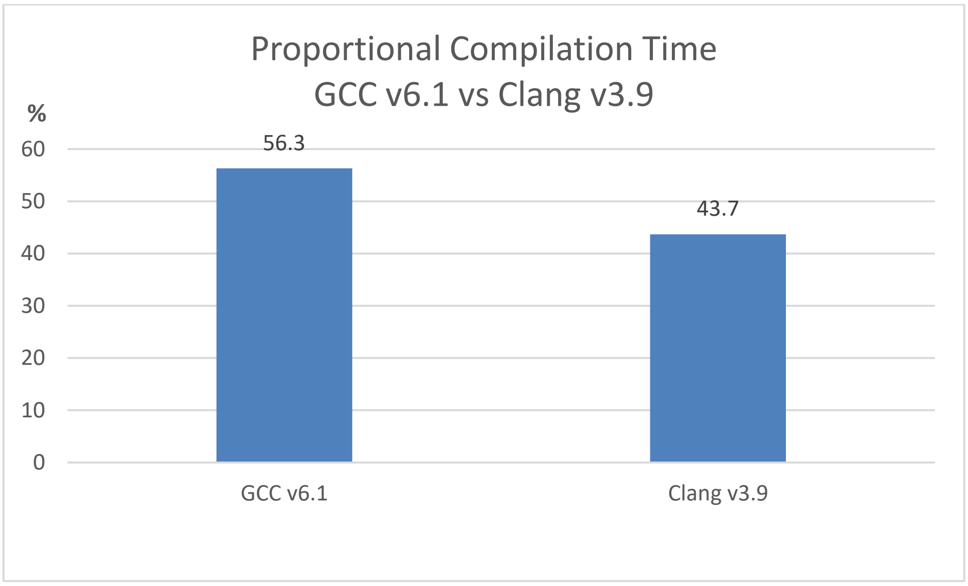


Figure 16 *Proportional Compilation Time GCC v6.1 vs Clang v3.9 /27/*

The first test suite compares Clang version 3.9 and GCC version 6.1. It can be seen from Figure 16 above, that in this test Clang is significantly faster in comparison to GCC. /27/

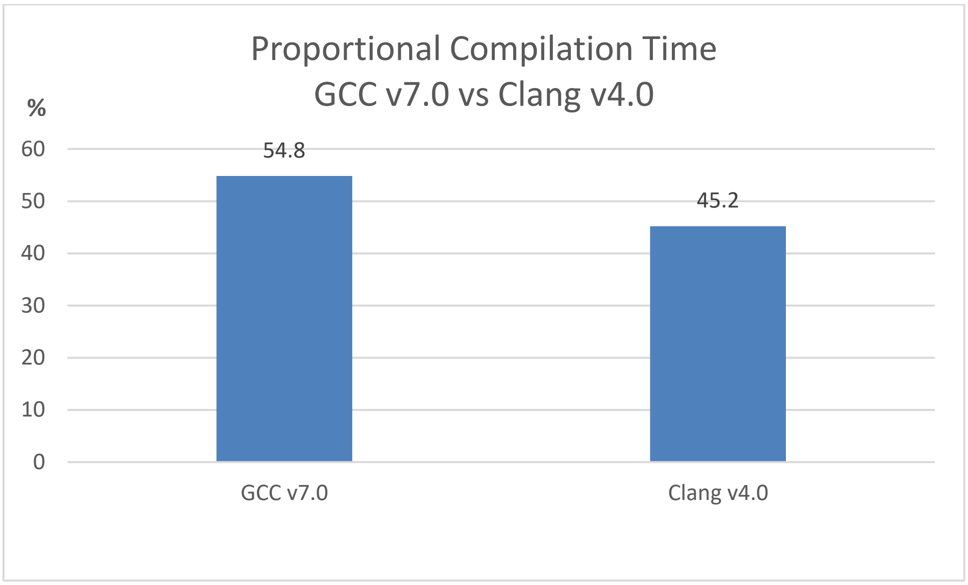


Figure 17 *Proportional Compilation Time GCC v7.0 vs Clang v4.0 /28/*

The next test suite compares GCC version 7 and Clang version 4.0. According to the Figure 17, the difference between compilation time of the compilers has decreased, but Clang version 4 is still clearly faster. /28/

It seems that Clang compiles the code a bit faster while newer versions of GCC are catching up the difference. There are test suites in which GCC outperforms Clang, but Clang is faster in the average. /27, 28/

4.3 Performance of Produced Program

The comparison of performance is done by calculating average proportional value of tests in various test suites. There are tables below (Figure 18, Figure 19 and Figure 20) to depict the average proportional performance of different versions of the compilers. The tests are run on Linux platform with the optimization flag -O3. More details of them can be found from the references. The tables are built by emphasizing every test with an integer value 100 and multiplied by the proportional percentage value. The values given in the tables are then summed up together and divided by the number of tests.

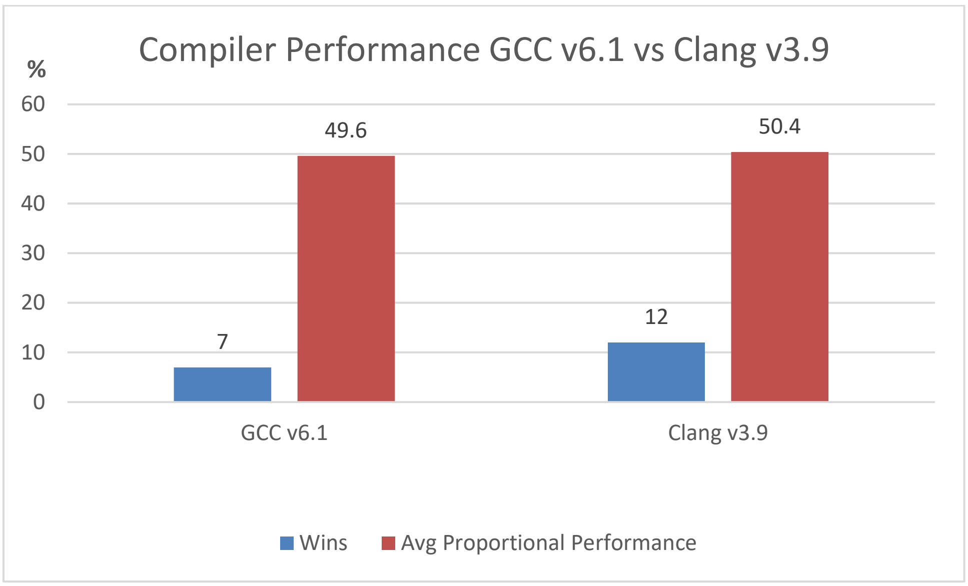


Figure 18 *Compiler Performance GCC v6.1 vs Clang v3.9 /27/*

The first test suite compares GCC version 6.1 and Clang version 3.9 (Figure 18). The suite consists of 19 tests. Clang got 12 wins over GCC. The performance was very even. /27/

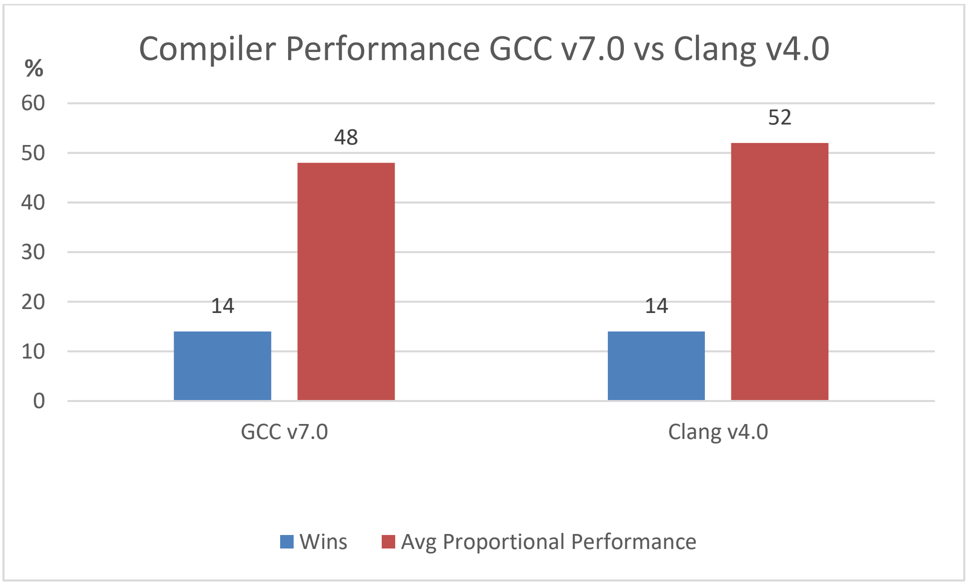


Figure 19 *Compiler Performance GCC v7.0 vs Clang v4.0 /28/*

The next test suite consists of 27 tests. It compares GCC version 7.0 and Clang version 4.0 (Figure 19). Both compilers took 13 wins and there was one draw. Though, Clang got 4 % better performance than GCC. /28/

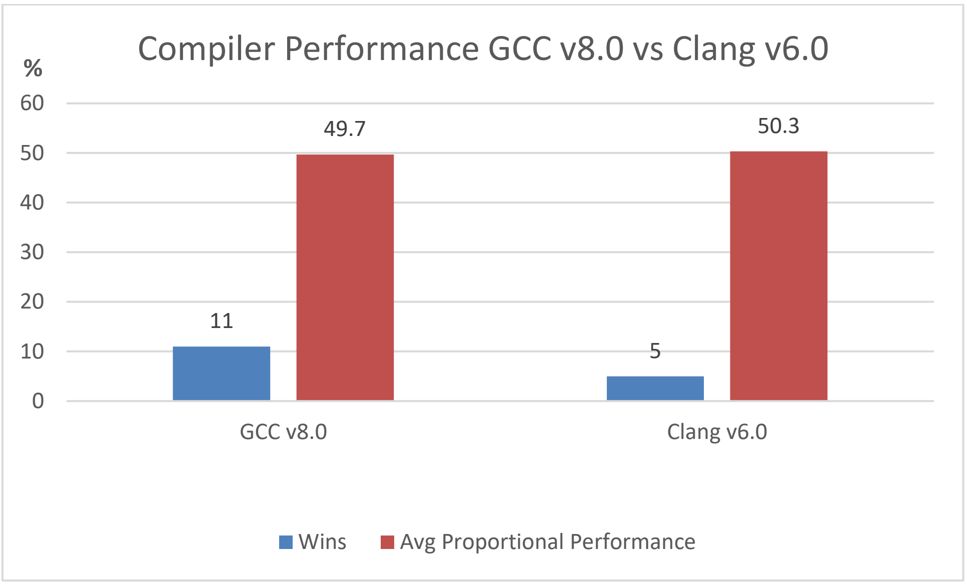


Figure 20 *Compiler Performance GCC v8.0 vs Clang v6.0 /29/*

In the last test suite (Figure 20), GCC version 8 took 11 wins over Clang version 6, which took 5 wins. Clang produced much better performance in one of the tests and that is why the average proportional performance is a little bit higher with Clang. /29/

The results of performance tests show that both compilers have their own pros and cons. Before GCC version 8, Clang has produced better performance, but the difference has equalized between the newest versions of the compilers (Figure 21 and Figure 22). The comparison of the newest versions of the compilers shows, that most of the times GCC is performing better. On the other hand, Clang has better total performance value, because in those cases where it won, it did it sovereignly. It could be assumed, that the further the development of the compilers go, the less there will be difference between them. One reason for this is the fact that the compiler developers can learn from each other's as the code is open source in both sides. /27, 28, 29/

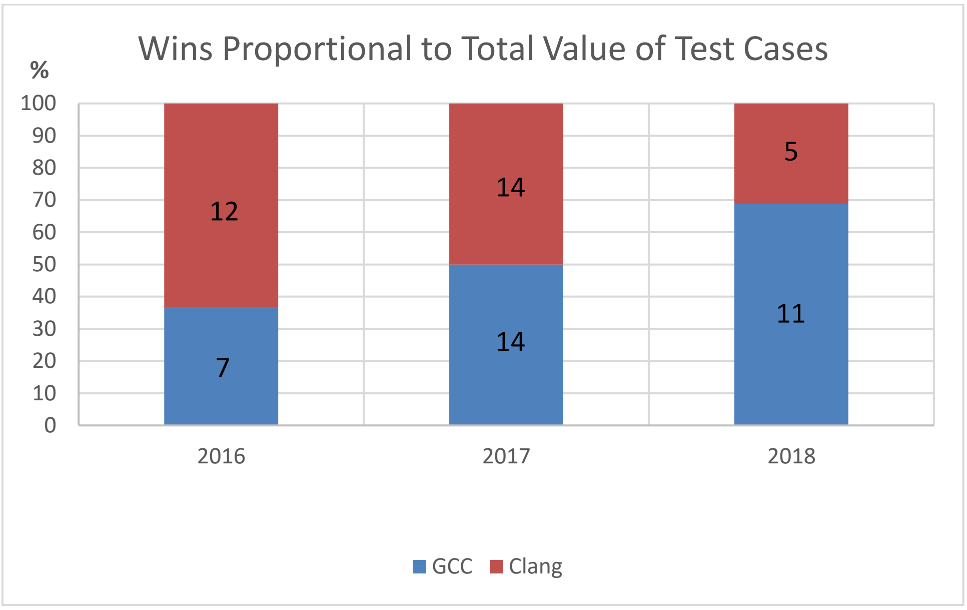


Figure 21 *Wins Proportional to Total Value of Test Cases /27, 28, 29/*

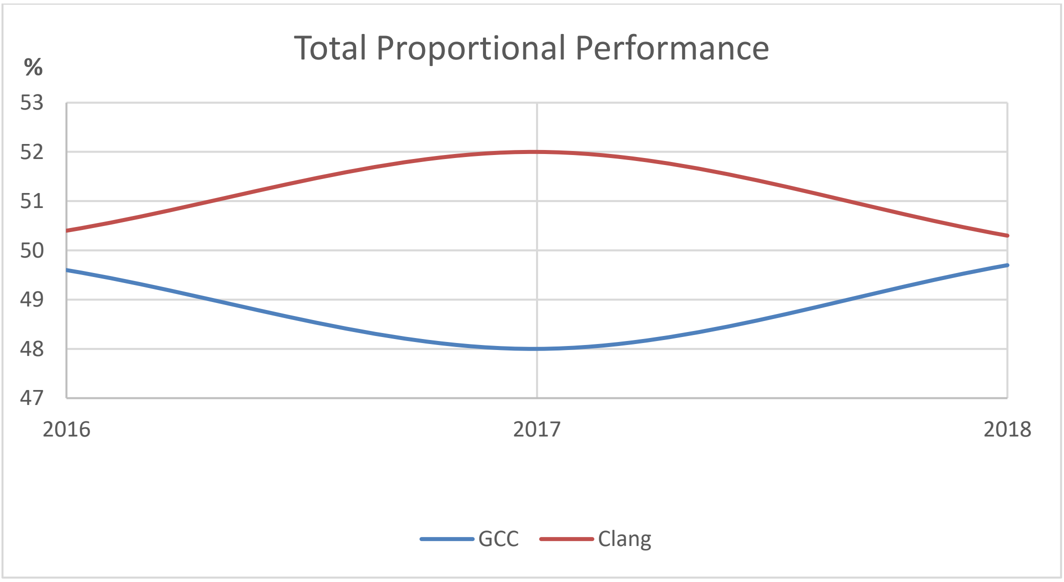


Figure 22 *Total Compiler Performance /27, 28, 29/*

4.4 Optimizations

Both compilers have the same optimization levels -O0, -O1, -O2 and -O3. Optimization level zero (0) is the default. It reduces compilation time and makes debugging produce the expected results. Level three (3) takes all of the optimizations in use and thus increases a compilation time, but also improves performance. Other levels are between the two. There are also some special optimizations. -Os optimizes for size, -Ofast is pretty much the same as -O3, and -Og optimizes for debugging experience. Clang has optimization flag -Oz, that is close to -Os, which optimizes for size, but reduces the code size even further. Clang has also optimization flags that are higher than -O3, but currently they are equivalent to -O3. /32, 33/

4.5 Memory usage

The competition which compiler use memory more effectively is tight. In the following test suite, there is eight different projects, and the proportional size of binaries are compared. Details about the case can be found from the references. The results of the test are shown in the graphs below (Figure 23 and Figure 24). Both compilers took four wins of total eight, but Clang beats GCC in total comparison. The test suite is run using GCC version 7.1 and Clang version 4.0. The projects are built with GCC using optimization flag -Os, and with Clang optimization flags -Os and -Oz. /34/

It can be seen from the Figure 23, that without one exception, the sizes of the binaries are close together. In the one exception case, Clang produced significantly better results and that causes the total comparison to be better as well (Figure 24). /34/

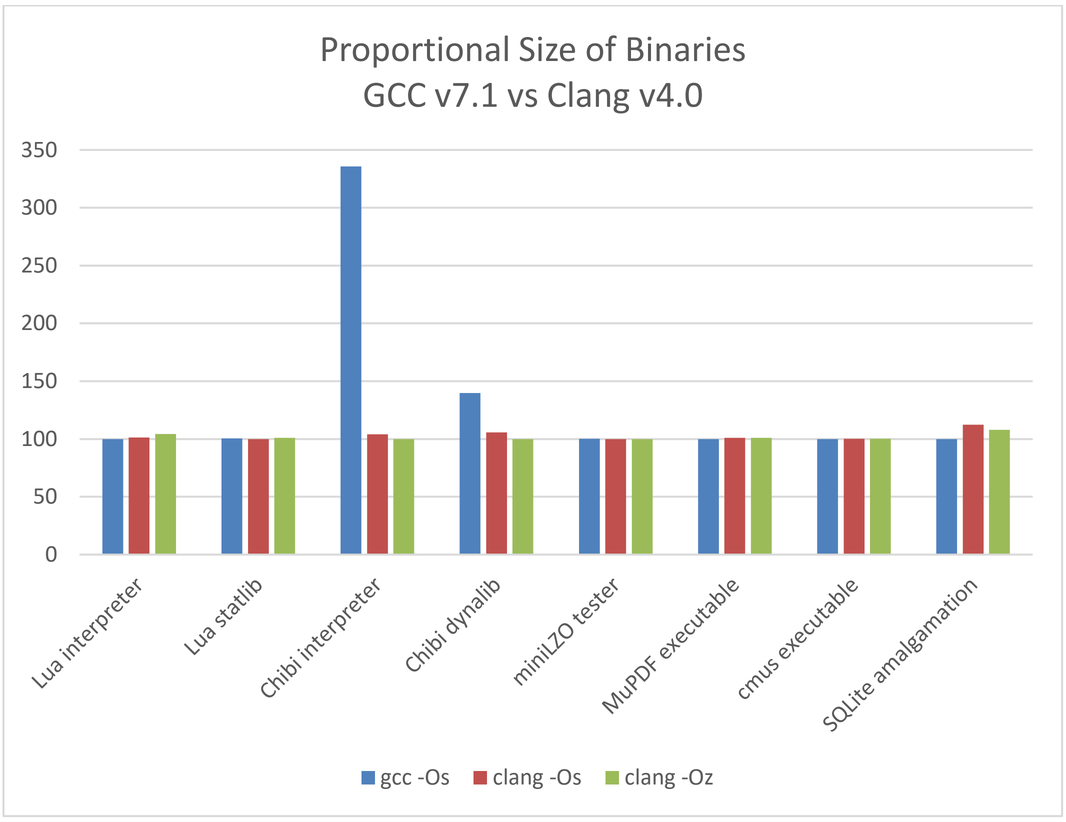


Figure 23 *Comparing Size of Binaries /34/*

From the total results in Figure 24 can be seen, that there is no significant difference between the outputs whether Clang -Oz or -Os flag is used.

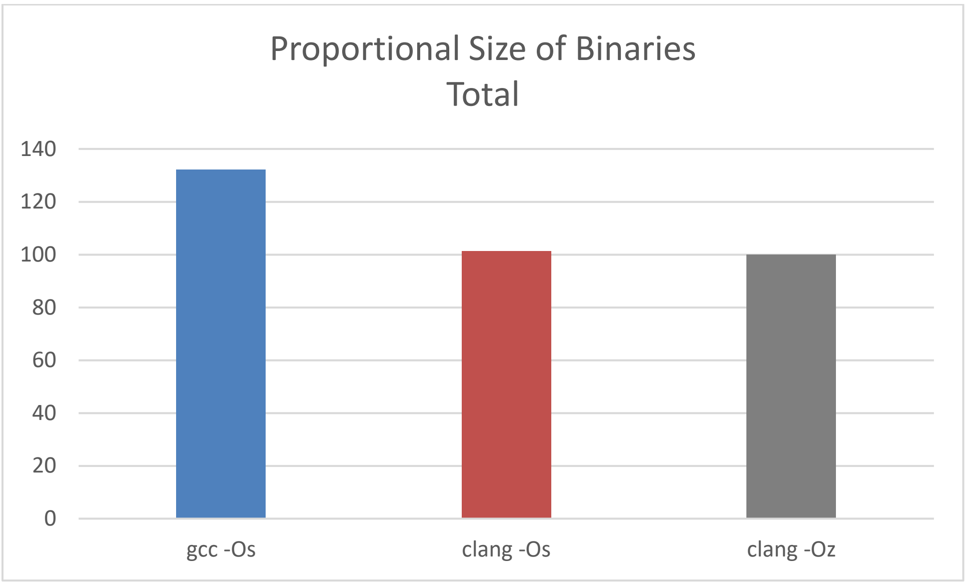


Figure 24 *Binary size comparison, total results*

4.6 Runtime Error Detection

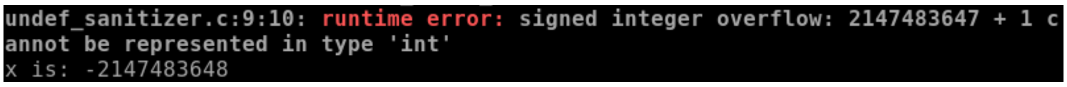
Runtime error detection mechanism, more generally known as a sanitizer, is a method to find failures in code during runtime that cannot be detected in the compilation phase. This feature becomes very handy, when testing code for errors. There are many kinds of sanitizers, for example AddressSanitizer, ThreadSanitizer, MemorySanitizer, UndefinedBehaviorSanitizer, DataFlowSanitizer and LeakSanitizer. By default, sanitizers are not enabled, but they can be enabled by giving a *-fsanitize=sanitizer* flag for a compiler. /24, 25/

To give an example, an example program is given, where a value of a variable is overrun.

```
int main()
{
    int x = 1;
    while(x > 0) {
        x++;
    }
    return 0;
}
```

The above program will finish after the value of *x* is less than zero. It should not really happen though, as the value is only incremented, not decremented. When the value exceeds its maximum and is incremented one more time, it will run over and become negative. Compiler is not able to warn about this situation, but the runtime error detection is. In some cases, overflowing a variable is a deliberate act, but this example is only to show what the runtime error detection is able to do.

After the above code is compiled with an appropriate sanitizer flag -*fsanitize=undefined* and run, the program detected an undefined behavior and gave an error of it (Figure 25). It worked with both compilers.



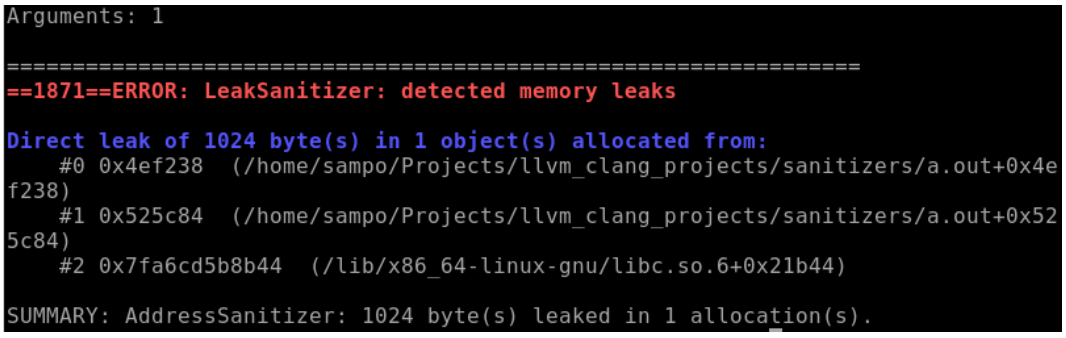
```
undef_sanitizer.c:9:10: runtime error: signed integer overflow: 2147483647 + 1 c
annot be represented in type 'int'
x is: -2147483648
```

Figure 25 *Output of clang compiled code*

Another example is about memory allocation that is not freed before program ends. The code below is compiled with -*fsanitize=address* flag.

```
int main(int argc, char** argv)
{
   char * buffer = malloc(1024);
   return 0;
}
```

The program gave an error of leaking memory with both compilers while the address sanitizer was enabled (Figure 26). The error message is not very descriptive though. It may be difficult to find the exact place of the memory leak from code.



```
Arguments: 1

=================================================================
==1871==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 1024 byte(s) in 1 object(s) allocated from:
    #0 0x4ef238  (/home/sampo/Projects/llvm_clang_projects/sanitizers/a.out+0x4e
f238)
    #1 0x525c84  (/home/sampo/Projects/llvm_clang_projects/sanitizers/a.out+0x52
5c84)
    #2 0x7fa6cd5b8b44  (/lib/x86_64-linux-gnu/libc.so.6+0x21b44)

SUMMARY: AddressSanitizer: 1024 byte(s) leaked in 1 allocation(s).
```

Figure 26 *AddressSanitizer program output*

4.6.1 Differences between GCC and Clang Sanitizers

Address-, Thread- and MemorySanitizers are originally developed as a part of the Clang project by Google but are later adopted for GCC as well. The basic usage of sanitizers works well with both compilers but there some differences. For instance, Clang supports wider range of AddressSanitizer features, as it can be seen from the Table 4 below. More information about the differences between the sanitizers can be found from the resources. /41, 26/

Table 4 *Differences between GCC and LLVM AddressSanitizer Features /41/*

AddressSanitizer Feature	**GCC v7.1**	**LLVM v5.0**
Std containers overflow detection	Yes	Yes
Dynamic allocation overflow detection	Compiler support missing	Yes
Using private aliases for globals	Yes	Optional, not safe
ODR violation detection	Yes	Yes
Symbol size changing for global variables	No	Yes
Adaptive global redzone sizes	No	Yes
KASan support	Yes	Limited

ASan_experiments support	No	Yes
Invalid pointer pairs detection	No	Yes
Default runtime library linkage	Dynamic	Static
Use explicit list of exported symbols	No	Yes
Asan_symbolize script	No	Yes
Support ASan blacklist file	No	Yes
Support sanitizer coverage	Limited	Yes
Support old Linux kernels (< 3.0)	Yes	No
Support no_sanitize attribute	No	Yes
Instrument function call arguments whose address is taken	No	Yes
Support dead stripping of globals on Linux	No	Yes

4.7 Fuzz Testing

The basic idea of fuzz testing, or fuzzing, is to run code with massive amount of random inputs trying to find vulnerabilities of the code. Fuzzing is done by creating a testbench for code, pairing it with a fuzzing engine that generates random inputs, and launching it to run on a server. After the code has being tested for hours, days or weeks, it can be seen from the testbench, which inputs have caused an error to occur.

LLVM includes a library for fuzz testing called libFuzzer. It is used by writing a testbench for the code and adding *fuzzer* sanitizer for the compilation call.

```
clang -g -O1 -fsanitize=fuzzer thecode.c
```

With GCC the fuzz testing can be done by external tools such as A*merican Fuzzy Lop.* In similar way, first a testbench is created for the project. Then the project is configured for American Fuzzy Lop, and the external tool is invoked.

LLVM way of fuzzing is easier in comparison to GCC.

5 CONSIDERATIONS OF MIGRATION TO CLANG

This section focuses on how the migration from Clang to GCC can be done and what one should consider when switching between the compilers.

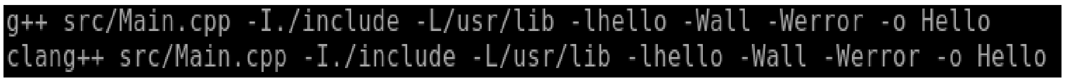
Many Linux distributions have already Clang built-in as an alternative to GCC and Microsoft supports it in Visual Studio, so the compatibility should not be an issue anymore. /46/

5.1 Using GCC Frontend with LLVM Backend

GCC frontend for LLVM backend can be used with the DragonEgg tool. Even though it has not been developed much in the recent years, it works with the newest versions of GCC and LLVM. Using the DragonEgg may be useful in the case that Clang frontend is not wanted to be used for some reason. /15/

5.2 Switching to Clang/LLVM

Clang is developed to be a drop-in replacement for GCC, so it is easy to change to use it. In the small example project below (Figure 27), only thing needed to do is to change command *g++* to *clang++* and the options do not need to be changed.



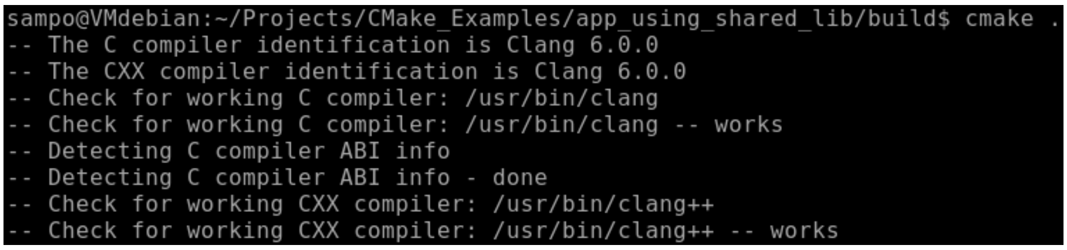
```
g++ src/Main.cpp -I./include -L/usr/lib -lhello -Wall -Werror -o Hello
clang++ src/Main.cpp -I./include -L/usr/lib -lhello -Wall -Werror -o Hello
```

Figure 27 *Switching between gcc and clang.*

5.2.1 Using CMake

If a project is configured with CMake, it makes it easier to switch between compilers. The compiler is chosen by setting *CMAKE_C_COMPILER* and *CMAKE_CXX_COMPILER* variables in a configuration file. In addition, all the compiler parameters set in the configuration file need to be checked that they are supported by the compiler going to be used.

After configuring the project, cmake will generate Makefiles for the compilation process (Figure 28).



```
sampo@VMdebian:~/Projects/CMake_Examples/app_using_shared_lib/build$ cmake .
-- The C compiler identification is Clang 6.0.0
-- The CXX compiler identification is Clang 6.0.0
-- Check for working C compiler: /usr/bin/clang
-- Check for working C compiler: /usr/bin/clang -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/clang++
-- Check for working CXX compiler: /usr/bin/clang++ -- works
```

Figure 28 *Calling cmake*

5.2.2 Differences on Binutils

There are some differences between LLVM and GNU binutils and that may be one concern when changing larger projects from GCC to Clang. LLVM provides alternatives for most of the GNU binutils, but the usage is not always the same. The most important differences are the usage of *strip* and *objcopy* tools from GNU binutils. *Strip* discards symbols from object files and *objcopy* copies contents from one object file to another. *Strip* tool is missing from LLVM, but similar tool is included in the linker *llvm-ld* and is used by adding a *–strip-all* or *–strip-debug* flag. LLVM has its own version of *objcopy* but it lacks for example *–only-keep-debug* flag that is used to strip a file while keeping the debugging section intact. /42, 43, 44/

In some cases, it may be reasonable to use some of the tools from GNU toolchain, for example *objcopy*, *strip* and *objdump* but change the others. This was done when compiled the company's own project using Clang (6.2 Creating a new Clang Based Toolchain). *Objdump* is to display information about one or more object files. /42/

5.3 Using Both Compilers in Parallel

It is a good practice to use both compilers in parallel to build a project. It will help finding errors more effectively. As a downside, it requires some extra work to maintain configurations for both compilers.

6 MIGRATING WÄRTSILÄ’S SOFTWARE TO CLANG

In this section, the process to migrate Wärtsilä’s own software to be built with Clang instead of GCC is described.

The software being migrated is developed on Linux based operating system. The GCC version currently in use is 4.8.2. It is released in October 2013. /50/

In this project, Clang and LLVM were compiled from source to configure them properly. The build files took a great slice of the memory of the development environment. Due to that, first the size of the development environment was expanded for it to be large enough for a new toolchain and both compilers.

After installing the new compiler, a new Clang based toolchain was created. It was done by first copying the existing GNU based toolchain and modifying it to use Clang instead of GCC.

Finally, the software was configured to use the new toolchain.

6.1 Building the Compiler

In order to set non-default configurations for Clang, it needs to be built from source. Configurations are needed for instance, to use *libc++* library, which is developed for Clang, instead of GNU’s *libstdc++*. *Libc++* includes Clang support for libraries *c++11* and *c++14*. In addition, by building the compiler from source, it makes it easier to use external tools from GNU toolchain. /47/

Step-by-step introduction to download, configure and build Clang from source is described below.

First, external binutils for LLVM are downloaded.

```
sudo mkdir /opt/llvm; cd /opt/llvm
sudo git clone --depth 1 git://sourceware.org/git/binutils-
gdb.git binutils_src;
sudo mkdir binutils_build; cd binutils_build
sudo ../binutils_src/configure --disable-werror
sudo make all-ld
```

Next, source code for Clang and LLVM are downloaded.

```
cd /opt/llvm
sudo git clone https://git.llvm.org/git/llvm.git/ llvm_src
cd /opt/llvm_src/tools
sudo git clone https://git.llvm.org/git/clang.git/
cd ../projects
sudo git clone https://git.llvm.org/git/compiler-rt.git/
sudo git clone http://llvm.org/git/lldb
sudo git clone https://git.llvm.org/git/libcxx.git/
sudo git clone https://git.llvm.org/git/libcxxabi.git/
```

After that, the dependencies for the project are installed and the build process is configured. This build is for 32-bit compiler because the target architecture is 32-bit. Another way is to build 64-bit version but giving a *-m32* flag for the compiler. In some cases, the build process can fail if not enough swap memory is allocated. This is why the allocation and swapping are done below.

```
sudo apt-get install libelf-dev
sudo apt-get install swig
sudo apt-get install python-dev
sudo apt-get install libtinfo-dev:i386
sudo apt-get install libffi-dev:i386
sudo apt-get install libelf-dev:i386

sudo fallocate -l 10g /mnt/10GB.swap
sudo chmod 600 /mnt/10GB.swap
sudo mkswap /mnt/10GB.swap
sudo swapon /mnt/10GB.swap

mkdir /opt/llvm/llvm_build; cd /opt/llvm/llvm_build
sudo cmake -G "Unix Makefiles" \
-DLLVM_ENABLE_PROJECTS="libcxx;libcxxabi;compiler-
rt;lldb;clang" \
-DLLVM_EXTERNAL_LIBCXX_SOURCE_DIR:PATH="/opt/llvm/llvm_src/
projects/libcxx" \
-DLLVM_EXTERNAL_LIBCXXABI_SOURCE_DIR=/opt/llvm/llvm_src/
projects/libcxxabi \
-DLLVM_EXTERNAL_COMPILER-RT_SOURCE_DIR=/opt/llvm/llvm_src/
projects/compiler-rt \
-DLLVM_EXTERNAL_LLDB_SOURCE_DIR=/opt/llvm/llvm_src/
projects/lldb \
-DLLVM_EXTERNAL_CLANG_SOURCE_DIR=/opt/llvm/llvm_src/
tools/clang \
-DLLVM_BINUTILS_INCDIR=/opt/llvm/binutils_src/include \
-DLLVM_ENABLE_LTO=ON \
-DLLVM_TARGETS_TO_BUILD="X86" \
-DLLVM_BUILD_32_BITS=ON \
-DLLVM_USE_SANITIZER="Address;Undefined" \
-DLLVM_INSTALL_BINUTILS_SYMLINKS=ON \
/opt/llvm/llvm_src/
```

The flags used to configure the project are described below. /51/

```
-G"Unix Makefiles" – For generating make-compatible paral-
lel makefiles

CMAKE_INSTALL_PREFIX=directory – Specify for directory the
full pathname of where you want the LLVM tools and librar-
ies to be installed (default /usr/local)

CMAKE_BUILD_TYPE=type – Valid options for type are Debug,
Release, RelWithDebInfo, and MinSizeRel. Default is Debug.

LLVM_ENABLE_ASSERTIONS=On – Compile with assertion checks
enabled (default is Yes for Debug builds, No for all other
build types).

LLVM_ENABLE_PROJECTS="project1;project2"
– Semicolon-separated list of projects to build, or all for
building all (clang, libcxx, libcxxabi, lldb, compiler-rt,
lld, polly) projects.

LLVM_EXTERNAL_PROJECTS - Semicolon-separated list of addi-
tional external projects to build as part of llvm.

LLVM_EXTERNAL_{CLANG,LLD,POLLY}_SOURCE_DIR=/PATH
– These variables specify the path to the source directory
for the external LLVM projects Clang, lld, and Polly, re-
spectively, relative to the top-level source directory.

LLVM_ENABLE_LTO - Add -flto or -flto= flags to the compile
and link command lines. On or Off. Defaults to Off.

LLVM_USE_SANITIZER="Adress,Undefined..." – Define the sani-
tizer used to build LLVM binaries and tests. Possible val-
ues are Address, Memory, MemoryWithOrigins, Undefined,
Thread, and Address;Undefined. Defaults to empty string.

LLVM_BUILD_32_BITS – Build 32-bit executables and libraries
on 64-bit systems.

LLVM_BINUTILS_INCDIR=/path/to/binutils/include – The cor-
rect include path will contain the file plugin-api.h. (To
use GOLD plugin)

LLVM_INSTALL_BINUTILS_SYMLINKS - Install symlinks from the
binutils tool names to the corresponding LLVM tools. For
example, ar will be symlinked to llvm-ar.
```

After the project is successfully configured, it is built and tested.

```
cd /opt/llvm/llvm_build
sudo make -j8
sudo make cxx
sudo make check
sudo make install
```

Finally, when all the phases above are executed without errors, the compiler is ready to use.

6.2 Creating a new Clang Based Toolchain

In this project, the toolchain was created by copying and modifying the existing one that is configured for GCC. The new toolchain used *objcopy, objdump* and *strip* tools from the GNU toolchain, but other tools from the LLVM project (5.2.2 Differences on Binutils).

GCC libraries and include files was removed from the new toolchain and they were replaced with corresponding Clang and LLVM files. It was important to remove the GCC files that they would not conflict with the similar include and library files from the LLVM project.

6.3 Configuring the Wärtsilä's Software

There was some changes needed to perform in order to configure the software for the new Clang based toolchain. In this section, the most important changes are revealed.

The compilation uses *ccache* to speed up recompilation. Ccache caches previous compilations and prevents same compilation from being run again. The software uses ccache compiler wrappers to detect which compiler is in use. The wrappers are simply pass-through shell scripts that invoke the actual ccache tool. In the development environment, there are ccache wrappers for gcc and g++ compilers in path */usr/bin*, and similar tools are created for clang and clang++ into the same path. /49/

The project is configured using CMake. Following changes need to be made in the CMake configuration files. The paths for tools in the new toolchain and the libraries and include files are given. Also the new compiler wrappers are taken into use.

```
set(COMPILER_WRAPPER_CLANG /usr/bin/ccache-clang)
set(COMPILER_WRAPPER_CLANGPP /usr/bin/ccache-clang++)

###################################
# Paths to Clang/LLVM
###################################
set(LLVM_EXT_BINUTILS_PATH
"/opt/llvm/binutils_build/binutils")
set(LLVM_PATH "/opt/llvm/llvm_build/bin")
set(LLVM_LIB_PATH "/opt/llvm/llvm_build/lib")
set(LLVM_INCLUDE_PATH "/opt/llvm/llvm_build/include")
set(CLANG_INCLUDE_PATH
"/opt/llvm/llvm_build/lib/clang/7.0.0/include")
set(CLANG_LIB_PATH
"/opt/llvm/llvm_build/lib/clang/7.0.0/lib/linux")

###################################
# Clang/LLVM Includes
###################################
include_directories("${LLVM_INCLUDE_PATH}/llvm/Support")
include_directories("${LLVM_INCLUDE_PATH}/llvm/Config")
include_directories("${LLVM_INCLUDE_PATH}/c++/v1")
in-
clude_directories("${LLVM_INCLUDE_PATH}/c++/v1/experimental
")
include_directories("${LLVM_INCLUDE_PATH}/c++/v1/ext")
include_directories("${CLANG_INCLUDE_PATH}")
include_directories("${CLANG_INCLUDE_PATH}/sanitizer")
include_directories("${CLANG_INCLUDE_PATH}/cuda_wrappers")
include_directories("${CLANG_INCLUDE_PATH}/xray")

###################################
# Clang/LLVM Libraries
###################################
link_directories("${W_LLVM_LIB_PATH} ${W_CLANG_LIB_PATH}")
```

List of the CMake parameters and corresponding tools from GCC and LLVM toolchains are shown in the Table 5.

Table 5 *CMake Parameters and Corresponding Tools from Toochains*

CMake Parameter	**GCC Tool**	**LLVM Tool**
CMAKE_C_COMPILER	ccache_gcc gcc	ccache_clang clang
CMAKE_CXX_COMPILER	ccache_g++ g++	ccache_clang++ clang++
CMAKE_AR	ar (GNU)	llvm-ar

CMAKE_STRIP	strip (GNU)	strip (GNU)
CMAKE_OBJCOPY	objcopy (GNU)	objcopy (GNU)
CMAKE_OBJDUMP	objdump (GNU)	objdump (GNU)
CMAKE_NM	nm (GNU)	llvm-nm
CMAKE_SIZE	size (GNU)	llvm-size
CMAKE_READELF	readelf (GNU)	llvm-readelf
CMAKE_LINKER	ld-linux.so (GNU)	libLLVMLinker.a

All the errors that occur during compilation process need to be resolved, but particular warnings can be omitted to get the compilation done. In the process, one error of nested function was fixed by moving the inner function out of another function and giving it more parameters. Another error was about redefined constant. The constant was removed from another place and the library file which included the first declaration was included the first place. Below is a list of the warnings that was omitted in the process.

-Wno-unused-command-line-argument
-Wno-implicit-function-declaration
-Wno-builtin-requires-header
-Wno-tautological-compare
-Wno-enum-conversion
-Wno-tautological-constant-out-of-range-compare
-Wno-duplicate-decl-specifier
-Wno-static-local-in-inline
-Wno-reserved-user-defined-literal
-Wno-varargs
-Wno-gnu-designator
-Wno-parentheses-equality

-Wno-mismatched-tags

-Wno-sometimes-uninitialized

-Wno-constant-conversion

-Wno-shift-sign-overflow

-Wno-undefined-inline

-Wno-unused-private-field

-Wno-logical-not-parentheses

-Wno-overloaded-virtual

-Wno-return-type

-Wno-unused-const-variable

-Wno-format

-Wno-comment

6.4 Results of the Compilation Process

There were some problems on the compilation process, that occurred more likely because of conflicts between GCC and LLVM include files. One thing that may have caused them was that the toolchain was only modified to support Clang instead of creating it similarly as the old toolchain was created. In the future, this is the first thing that needs to be done to get it work properly. Some other problems occurred when *libstdc++* was used but that was fixed by using *libc++* library (6.1 Building the Compiler).

One notice is that the GCC version currently used is starting to be old (released in 2013). It could be updated to get the new features and enhancements of GCC in use.

Quite many warnings occurred when the project was compiled using Clang. Of course, some of them occurred because GCC supports some functionalities that are not yet supported by Clang. Clang found for instance, redefined constants, functions without return type, implicit function declarations and variables that are sometimes, but not always, uninitialized (6.3 Configuring the Wärtsilä's Software). All of them are risk for bugs to occur.

The comparison tests of compilation time and performance for the software could not be run in this project. However, the more important thing is how well a compiler finds bugs and how are the error and warning messages displayed. It could be expected, that by finding potential bugs earlier and faster, the productivity of software development is increased measurably.

Using two compilers requires some extra work to configure project to support both of them. Also it means that not all of the functionalities of GCC can be used, but there are not so many of them after all. Despite these facts, it may be worth the effort, as it has been seen, that using two compilers is a good way to prevent bugs from occurring.

7 CONCLUSION

To answer a question, which compiler is better, it highly depends on the case. It depends on the architecture, requirements, structure and the way a project is coded, i.e. which pointer types are used and so on. Both compilers have their own pros and cons. Owing to these facts, it would be good practice to have both compilers in use to find out which one is better for each particular case. Also using both compilers can be a good way to prevent bugs from occurring. This is because it has been seen during the project, that Clang could find some warnings that GCC did not notice. Also Clang supports some features of sanitizers that are not yet working with GCC.

The work needed to migrate from GCC to Clang is proportional to the size of the project and complexity of configurations. With smaller projects the amount of work is minimal, but for instance in the Wärtsilä case, there is need for some redesign of the configurations of the compilation process to support both compilers comprehensively. After the work is done, switching between compilers is fast and easy.

In the comparison phase can be seen that Clang won GCC in almost every test suite. One thing to mention for the benefit of GCC is that it worked well without any compatibility problems, while with Clang there was some of them. It is also much easier to find help with problems using GCC than it is currently with Clang. This is due to the wide user population that GCC has.

"Even though LLVM is usually compared to GCC and introduced to be a better alternative for it, it does not try or plan to obsolete GCC. They are two different projects focusing on different goals, even though there is some overlapping between them."
-Chris Lattner, the original developer of LLVM and Clang /38/

REFERENCES

/1/ Basics of Compiler Design, Torben Mogensen (2010), University of Copenhagen, Denmark

/2/ https://www.tutorialspoint.com/compiler_design

/3/ An Introduction to GCC (2004), Brian Gough, Bristol, UK

/4/ Managing Projects with GNU Make (2004), O'Reilly, Robert Mecklenburg

/5/ https://www.gnu.org/software/binutils/

/6/ https://www.gnu.org/software/gdb/

/7/
https://www.gnu.org/software/automake/manual/html_node/index.html#SEC_Contents

/8/ http://llvm.org/

/9/ http://www.aosabook.org/en/llvm.html

/10/ http://clang.llvm.org/docs/UsersManual.html

/11/ http://lld.llvm.org/

/12/ http://lldb.llvm.org/

/13/ https://llvm.org/docs/LinkTimeOptimization.html

/14/ https://cmake.org/

/15/ https://dragonegg.llvm.org/

/16/ https://releases.llvm.org/

/17/ https://gcc.gnu.org/onlinedocs/gcc-7.2.0/gcc/#toc-Language-Standards-Supported-by-GCC

/18/ https://www.rust-lang.org/en-US/contribute-compiler.html

/19/ https://www.gnu.org/software/gcc/backends.html

/20/ https://stackoverflow.com/questions/15036909/clang-how-to-list-supported-target-architectures

/21/ https://clang.llvm.org/docs/UsersManual.html#gcc-extensions-not-implemented-yet

/22/ https://clang.llvm.org/diagnostics.html

/23/ https://gcc.gnu.org/wiki/ClangDiagnosticsComparison

/24/ https://clang.llvm.org/docs/

/25/ https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

/26/ https://github.com/google/sanitizers/wiki

/27/ https://www.phoronix.com/scan.php?page=article&item=gcc-61-clang39&num=1

/28/ https://www.phoronix.com/scan.php?page=article&item=gcc7-clang4-jan&num=1

/29/ https://www.phoronix.com/scan.php?page=article&item=epyc-compilers-nov&num=1

/30/ https://llvm.org/docs/GoldPlugin.html

/31/ https://web.archive.org/web/20111004073001/http://lists.trolltech.com/qt4-preview-feedback/2005-02/msg00691.html

/32/ https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

/33/ https://clang.llvm.org/docs/CommandGuide/clang.html

/34/ https://retro-freedom.nz/clang-vs-gcc-binary-size.html

/35/ https://lists.freebsd.org/pipermail/freebsd-current/2012-November/037610.html

/36/ https://gcc.gnu.org/gcc-6/changes.html

/37/ https://gcc.gnu.org/ml/gcc/2014-01/msg00247.html

/38/ https://www.youtube.com/watch?v=029YXzHtRy0

/39/ https://www.quora.com/What-is-the-difference-between-LLDB-and-GDB-And-why-dont-they-have-the-same-commands

/40/ https://mo.github.io/2015/11/04/browser-engines-active-developers-and-commit-rates.html

/41/ https://github.com/google/sanitizers/wiki/AddressSanitizerClangVsGCC-(5.0-vs-7.1)

/42/ http://lists.llvm.org/pipermail/llvm-dev/2017-June/113609.html

/43/ https://sourceware.org/binutils/docs/binutils/objcopy.html

/44/ https://sourceware.org/binutils/docs/binutils/strip.html

/45/ https://sourceware.org/binutils/docs/binutils/objdump.html

/46/ http://llvm.org/docs/GettingStarted.html#requirements

/47/ https://libcxx.llvm.org/

/48/ https://clang.llvm.org/get_started.html

/49/ https://ccache.samba.org/

/50/ https://www.gnu.org/software/gcc/gcc-4.8/

/51/ https://llvm.org/docs/CMake.html

/52/ https://www.wartsila.com/about