LLVM/CLANG the new C-compiler

A new star shines on the sky

**Overview of CLANG, its code quality tools,
the GPL-free license model
and its implementation for Microware OS-9**

Kei Thomsen, MicroSys Electronics GmbH

Introduction

C/C++ compilers have long been available for various processors and operating systems. Some compilers are very old, but maintained and up to date. Others are simply old compilers. So it was time to launch a new modern compiler, one that has been created with the latest programming methods and can do everything from the beginning that was cumbersomely added to the old compilers. Is it just a new compiler? Then it is uninteresting and boring. If it also includes code analysis tools and other tools for improving code quality, it is unrivalled in its value. For OS-9 a new compiler with the latest C/C++ standards is needed.

With LLVM/CLANG something completely new has been created.

The master thesis in 2000 by Chris Lattner and Vikram Adve at the University of Illinois on a Low Level Virtual Machine (LLVM) quickly turned into a completely new approach to a compiler for many current processor architectures. The name "Low Level Virtual Machine" sometimes causes confusion. LLVM is the name of the actual project for the LLVM-IR intermediate language, the LLDB debugger, the C and C++ libraries and the CLANG compiler. CLANG (C-Lang or "klæŋ", both is okay) is the frontend for many programming languages, especially C and C++. In 2005 Apple became aware of the CLANG and hired Chris Lattner as project manager to further develop this software technology. Since July 2008 the LLVM is now the standard compiler for Apple's development environment Xcode. Other sponsors are Qualcomm, Google, Fastly, ARM, MicroSoft, Intel, Cisco, Synopsys, Facebook, HSA Foundation, Huawei, Sony, Mentor and many more (https://llvm.org/foundation/sponsors.html).

What makes the CLANG so special? First the CLANG was used more as a frontend or driver plugin to the GCC and later it became a complete replacement for the GNU compiler. Since then, the compiler has been able to reach its full potential and shows how valuable a software development coordinated from the ground up is. The LLVM/CLANG has always placed great emphasis on a consistent style in the LLVM coding standard, code quality, repository and code reviews; unlike the GCC, which was created in 1987 and has been continuously expanded ever since. This left quite a mess in the code at times. For example, the global addressing of variables in GCC is created in about 40-70 places of the backend and the machine descriptions, whereas

in CLANG it is created in exactly one function of a class. The LLVM/CLANG and its tools are based on the most modern methods and technologies from the very beginning and thus perform many actions much more efficiently and coordinated. Hence, it is not surprising that today's standards like C11, C++17 and C++2x are not only created with the GCC, but increasingly also with the LLVM/CLANG.

Shortened compile time

The actual compile process is different between a classic compiler like GCC and the newer CLANG. The classic compiler takes the C/C++ file, inserts the header files and first generates an internal format from it. This result is then optimized and converted by a backend to target processor assembler. Depending on the compiler, it can happen 2-3 times that the intermediate data is written as a file or pipe and read in again. This means that a lot of time is spent formatting the output and scanning/parsing the input. The result is the assembler file. This file is read by an assembler in a further step and converted into an object file (.o), which the linker then assembles with other object files and libraries to form a program. Formatting and re-reading the output takes about 1/4 of the total compile time (Figure 1).

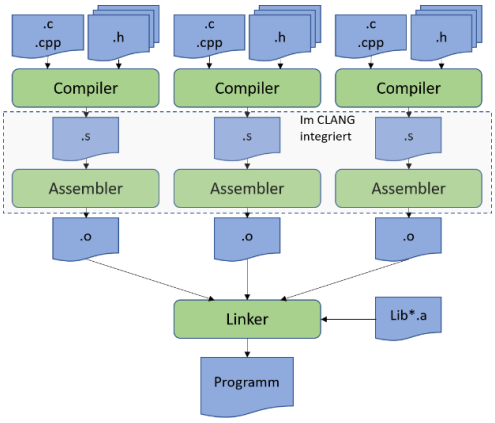


Figure 1: Classical Compiler with assembler

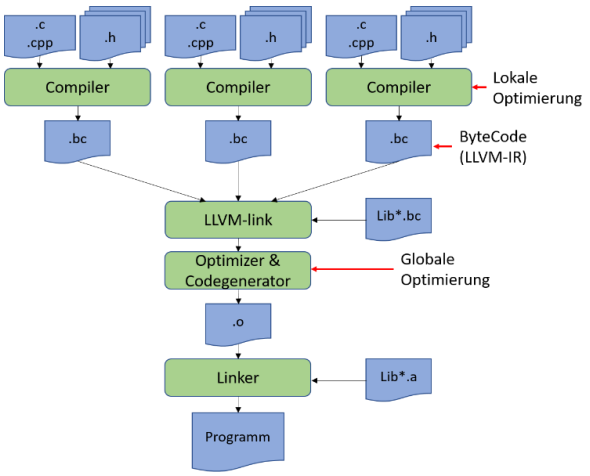


Figure 2: CLANG with alternative ByteCode Linking

To speed up and simplify this, the CLANG takes a different approach. An internal LLVM-IR format is also generated, but it is not output or formatted, but passed internally as an object/structure up to the object file (.o). This saves a lot of runtime, because at least the formatting and parsing of the assembler file is not necessary. Instead, a lot of information (debug, sections, types, etc.) are passed on directly, which up to now only appeared in the assembler files as elaborate additional information. Of course, an output can be forced at any point of the compile process, e.g. to continue working with byte code (LLVM-IR) or assembler code.

Performance improvement by linking on a higher level

Another big plus is the alternative of being able to link several sources and libraries together on the LLVM-IR level and then have them optimized as a whole. This allows small functions from other source files to be "inlined" directly. A feature that can improve the performance of programs significantly, because the optimization is done over the whole program and not only over the single source files (Figure 2). A very nice paper about this can be found at "TU-Dresden LLVM". This LLVM-IR is like the OS-9 UCC compiler with I-Code linking, therefore nothing really new for OS-9 developers, but for all others.

Attractive licensing model

What makes the LLVM/CLANG especially interesting for companies is the licensing. Many companies shy away from the GNU Public License (GPL) and would like to do without GNU compilers, tools and libraries because they are afraid of not fulfilling the requirements and thus becoming suable. The LLVM/CLANG environment offers a much more open and company-friendly licensing model. Originally it was the University of Illinois/NCSA Open Source License (UI/NCSAOSL), in 2019 it changed to Apache 2.0 License with LLVM exceptions:

Apache 2.0 License with LLVM exceptions (simplified summary)

- You may freely use, modify and distribute software under this license in any environment
- A copy of the license must be included in the package.
- Changes to the source code of the software under the Apache license do not have to be returned to the licensor.
- Own software using software under the Apache License does not have to be under the Apache License.

Original text: https://www.apache.org/licenses/LICENSE-2.0 plus exception from http://llvm.org/docs/DeveloperPolicy.html#new-llvm-project-license-framework

In order to become independent of the GPL problem with a C and C++ library, the LLVM has also been used to create the libc (MUSL), libc++, OpenMP, polly and other libraries from scratch. Under this license also the own LLD linker and the LLDB debugger are available. The LLVM libraries can also be used with the GCC, so they are not directly bound to the CLANG.

Easy changeover from GCC to CLANG

Since the CLANG has almost all command-line options like the GCC, including the -f... and -m... options, switching from the GCC to the CLANG is quite easy. Just replace the gcc or cc with clang in the Makefile. For ./configure scripts, you only need to set the environments CC=clang and CXX=clang++ and the Makefile for the CLANG is created. This makes the changeover particularly easy.

Changeover from UCC to CLANG

As seen, the CLANG uses the same options as the GCC. These are totally different to the UCC options. Here some work is needed if using makefiles. For example, the -d is now -D, -v is now -I and the library naming is different. Also the file endings have changed from the classical OS-9 naming to standard UNIX/Linux naming. The object format is no longer compatible, therefore old libraries cannot be used as binary.

Quality and speed comparison

At this point I would like to refrain from a speed and quality comparison between GCC and CLANG. The GCC and CLANG sources change several times a day, because both compilers are constantly being worked on and therefore changes occur all the time. Compared to the UCC, the output of the CLANG is typically between 5 and 20% faster.

Supported CPU-Architectures

The fact with the CLANG is that it does not support all previously known CPU platforms, but focuses on current CPUs, such as X86, AMDGPU, ARM, Hexagon, NVPTX, PowerPC, Sparc, SystemZ, XCore, MIPS, eBPF, RISCV. There used to be a backend for older CPU architectures like the 68k years ago, but currently there is no support anymore for it in the master repository.What is special about the CLANG is that it has all or alternatively only a selection of CPUs in a single binary and can also generate the output for different operating systems. You always use exactly the same compiler for all your CPUs and operating systems, so you do not have to deal with different compilers. In one configuration, code can be compiled for PowerPC, ARM and X86 for Linux, Windows, Free/Open/NetBSD and OS-9 for example, all with a single CLANG binary. It only depends on which libraries and headers were built and installed. This makes it a self-hosted and cross compiler in one.

Meanwhile even the Linux kernel can be compiled not only for ARM and since CLANG-9 also for X86. https://www.golem.de/news/compiler-llvm-9-baut-x86-linux-kernel-1909-143986.html .

Tools for high code quality

Static Code Analysis

With the CLANG, a static code analysis tool has been built directly into the compiler. The advantage is: you don't have a compiler X and a static code analysis tool Y, where you must tell Y how X works, which defines are sets, where the headers come from, etc. Within CLANG, the compiler uses exactly the same settings and code parts as the Analysis Tool, because the Analyzer is part of the compiler. The Analyzer knows exactly how the compiler processes certain things and can therefore perform very precise analyses. This is especially important when using the CLANG as a cross compiler for other CPU platforms and even another operating system (e.g. the RTOS Microware OS-9), because here completely different header files are used than with Linux or Windows. By specifying -analyze -analyzer-output html the CLANG is instructed not to compile as usual, but to subject the source code to a static code

analysis and write the result to a directory <sourcefile>.plist. For each hint a separate HTML file is created with the problem found and its conditions. It is even easier with the scan-build tool. This is a Python script that changes the environment CC and CXX for the common build tools and thus additionally calls the CLANG with the analysis function. So you get the usual compiler output of "your" compiler and also the information of the code analyzer. In addition, the scan-build generates an index.html with a summary of all findings and includes a stylesheet and JavaScript so that the display in the browser is formatted correctly.

scan-build make -j 4
starts Makefile (4 parallel runs), additionally to the static code analyzer.

scan-build ./configure
scan-build –keep-cc make
starts configure with the analyzer

scan-build gcc -O3 myprog.cpp -o myprog
regular compile by hand (here with GCC) using the scan-build.

See http://clang-analyzer.llvm.org/ for more information about the analyzer.

In the following example, for "libedit"

scan-build ./configure
scan-build –keep-cc make -j 6

a result list is created (Figure 3). 29 problems were discovered, of which 4 use-after-free positions were selected here. It is very nice to see that the analysis is not limited to one function only, but is also detected beyond subfunctions. Whether one follows up on the messages now or trusts that this case will never occur after all, must be considered on a case-by-case basis.

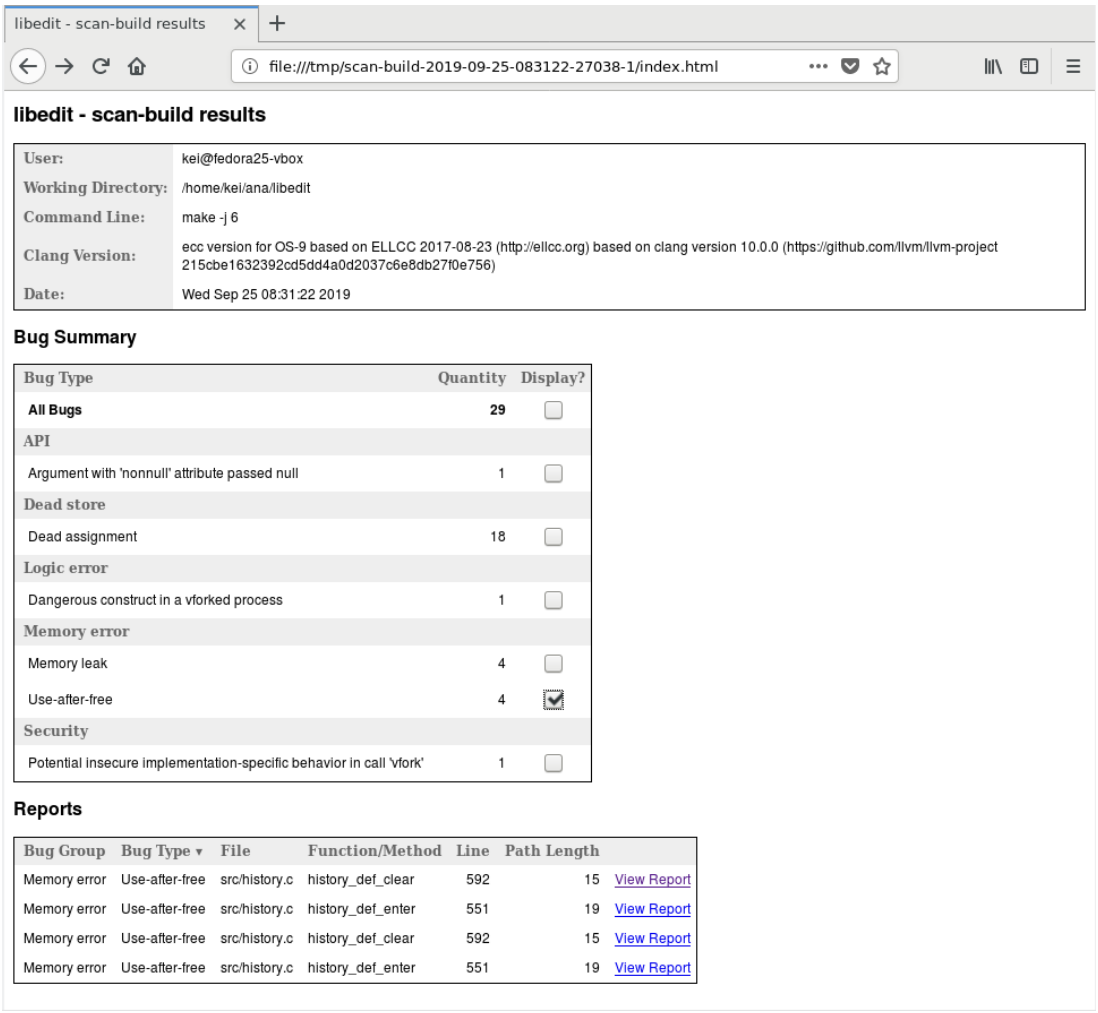


Figure 3: Overview of the static code analysis for libedit

The message in this example was viewed, discussed and decided by two of us for about 1/2h, yes this could really happen (Fig. 4). We did not pursue whether it could also occur in connection with the calling library. Here is an example of how the analyses look like. In Figure 4 on the left you can click backwards through the conditions starting from "15 Use of memory after it is freed" as the last entry.

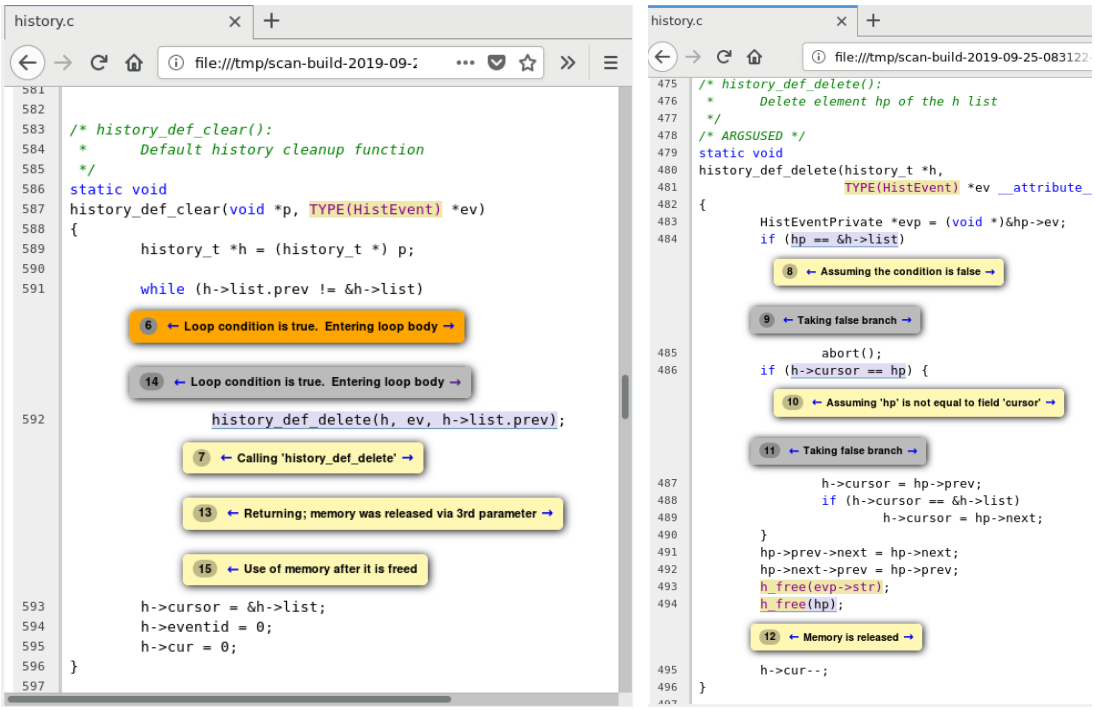


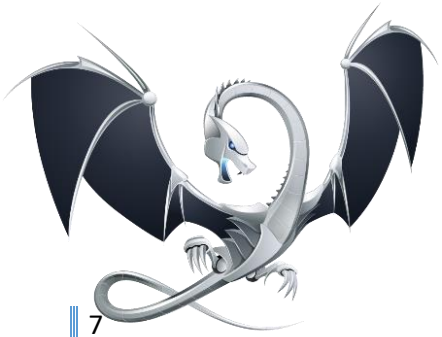
Figure 4: Detailed information in the browser about a detected problem.

Sanitizer

Further help is provided by the Sanitizer in the compiler. This is nothing new and is also available in GCC, but this is hardly known to the general public. Sanitizers are additional code parts that are inserted during compilation to perform various checks. There are different types of address, thread, memory or undefined behavior sanitizers, all of which are designed to help you identify different problems in the program that you might not easily come across or even know exist. Information about this in https://clang.llvm.org/docs for the different sanitizers. Implementation and examples for using with OS-9 should follow later this year.

Summary

The CLANG compiler with its complete toolchain is a really good replacement for the GNU compiler and now also for UCC OS-9 application. It has been developed according to the latest methods, is used by the very big players and masters the latest standards in C and C++. The faster processing saves time when compiling and at the same time it offers a lot of possibilities for stress-free use due to its more open license. Its built-in static code analyzer provides a cost-effective method for checking the source code.
My tip: just try the CLANG, it's worth it! Especially the static code analyzer can be really helpful to find and avoid errors an to be more productive.



Sources

The LLVM Dragon logo is © Apple Inc.

The detailed information in this text can be found on llvm.org and clang.llvm.org.

LLVM Coding Standard: https://llvm.org/docs/CodingStandards.html

TU-Dresden LLVM: https://tu-dresden.de/ing/informatik/ti/vlsi/ressourcen/dateien/dateien_studium/dateien_lehstuhlseminar/vortraege_lehrstuhlseminar/folder-2012-11-09-8491578029/LLVM.pdf?lang=de

Author

Dipl.Ing. (FH) Kei Thomsen has 31 years of experience in the field of embedded RTOS programming. Since 1997 he is responsible for the operating system Microware OS-9 as developer, support and trainer. One focus are hardware-related developments for customer-specific products based on PowerPC, ARM and X86 CPU architectures.

Contact Information:

Kei Thomsen, thomsen@microsys.de
MicroSys Electronics GmbH
D-82054 Sauerlach, Muehlweg 1,
www.microsys.de

Microware OS-9: Background

Since February 2013 the RTOS Microware OS-9 is owned by Microware LP (https://microware.com), a partnership of three companies, MicroSys, Freestation (Japan) and RTSI (USA).

Microware LP actively continues the development of OS-9. Recent enhancements provide support for e.g. Arm® Cortex® A8, A9 and A53/72 based CPUs.

MicroSys in Sauerlach near Munich takes care of customers in Europe and provides technical support, consulting and development services.