International Journal of Internet, Broadcasting and Communication Vol.11 No.1 69-74 (2019)
http://dx.doi.org/10.7236/IJIBC.2019.11.1.69

IJIBC 19-1-9

Development of a Prototyping Tool for New Memory Subsystem

Jungseok Cho, Doosan Cho*

Sunchon National University, EE
dscho@scnu.ac.kr

Abstract

The compiler is the key of the prototyping framework for the new memory system. These compiler-centric prototyping tools have several components, including compiler, linker, assembler, and standard libraries. It takes a lot of cost and man power to develop it all at zero base. Therefore, developer usually use a development framework to develop these prototyping tools efficiently. These development frameworks should be free of licensing issues when considering the commercialization of development results. Thus, developer should investigate the development framework, which is free from licensing issues and that provides all of the development environment to enable actual execution. There are three representative compiler-centric development frameworks: GCC, Clang (LLVM), and MS visual studio. There are some differences depending on the release version among them. And, there are some limitations to the freeware and commercial use. We chose LLVM here to explain the development of prototyping tools. This information will help accelerate the development of prototyping tools and will help reduce system development costs.

Keywords: *memory system, cache memory, low power design, optimizing compiler, embedded system, prototyping tool*

1. Introduction

The situation in which the system speed does not improve at the same rate as CPU speed develops is called memory wall [1]. Memory system related researches are continuing as technology to improve the system speed. Prototyping tools for evaluating new memory structures are also becoming important continuously. The development of these prototyping tools is also time consuming. In this paper, based on the experience of developing prototyping tool, we present the experience of compiler development which is a core part.

A compiler is the key of the prototyping technology for the new memory system. A compiler is developed, targeted to a new memory system, and executed using the generated code to evaluate the completed system. These compiler-centric prototyping tools have several components, including the linker, assembler, and standard libraries. It takes a lot of cost to develop it all from zero base. Therefore, developer will usually use

Manuscript Received: Jan. 15, 2019 / Revised: Jan. 19, 2019 / Accepted: Jan. 27, 2019
Corresponding Author: dscho@scnu.ac.kr
Tel: +82-61-750-3577, Fax: +82-61-750-3570
Sunchon National University, EE

such framework to develop these tools. These development frameworks should be free of licensing issues when considering the commercialization of development results. You should first investigate the development framework, which is free from licensing issues and that provides all of the development environment to enable actual execution. There are three compiler platforms: GCC [2], Clang (LLVM) [3], and MS visual C ++ [4]. There are some differences depending on the version. However, there are some limitations to the freeware and commercial use. The following are the licensing and support operating systems and whether they support C ++ grammar standards (which is generally used to build system program).

Table 1. Comparison of compiler frameworks

Compiler	windows	Unix-like	license	C++11	C++14	C++17
GCC	MinGW	YES	UoI/NCSA	YES	YES	YES
Clang	YES	YES	GPLv3	YES	YES	Partial
Visual C++	YES	YES	Partial free	YES	YES	Partial

The table shows representative compiler development frameworks, supports both Windows and Linux operating systems, and supports most grammar standards. Licenses can be freely distributed by default, but there are some restrictions on commercialization. The most popular open source license is the GNU GPL [5], which allows commercial modification and distribution patents. However, GCC is an ancestor of the compiler system and it is very difficult and time-consuming to develop based on it.

Clang (LLVM) has both Illinois State University open source licenses and NCSA licenses [6]. It is simply a more relaxed version than the GPL license. The name LLVM is called the compiler platform brand, not the abbreviation. LLVM is a brand that applies to LLVM-related projects, LLVM intermediate representation (LL), LLVM debugger, and C ++ standard library implementation of LLVM. LLVM is managed by LLVM Foundation. The company's president is the compiler engineer Tanya Lattner. LLVM is a very powerful compiler infrastructure framework designed for compile-time, link-time, and run-time optimization of programs written in your favorite programming language. LLVM works on multiple platforms, and its main purpose is to generate fast-running code. LLVM is basically the core of the intermediate language representation (IR) generator, which simplifies the creation of custom compilers with the full flow of the compiler development (front end analyzer + IR generator + LLVM backend), along with a front end to connect the desired language. The LLVM compiler framework is straightforward compared to GCC, which consists of C ++ code and has reuse and development in C.

Visual C ++ also recently released the source code of the old version compiler and released it for free. Modifications are commercially available for distribution, but should not be specified as an MS asset. Given the licensing issues and compiler development costs, we developed a C ++ compiler with the LLVM compiler platform. The uniqueness of LLVM is that it provides only the compiler backend, so the components needed to configure a real software development environment, such as the front end and libraries / linkers, should use other packages together. To begin with, the compiler is software that translates advanced programming languages such as C ++ into machine code that computers can execute.

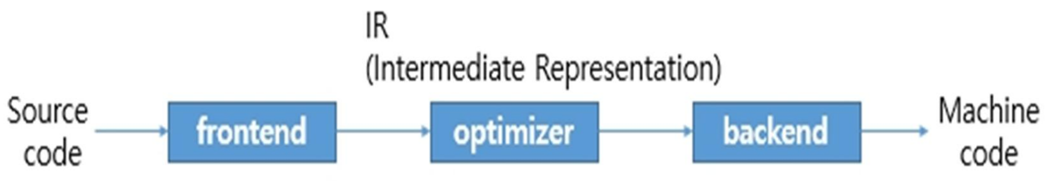


Figure 1. Compiler Construction [7]

As shown in the figure, when the source code is input, the front end parses and parses the intermediate language expression generated as a result of optimization, and the optimized intermediate language expression is mapped to the machine code to be generated as the final machine code is completed. Normally, the front-end parser uses the existing parser by parsing languages that do not change in a grammar such as C ++ or C, and we use Clang for this. Clang is a compiler front-end that interacts with LLVM to generate intermediate language expressions used as input to LLVM. In addition, the GCC front end can also be used with LLVM, and research has shown that Clang performance is superior to GCC, and LLVM mainly uses Clang.

Even Visual C ++ can use Clang as a front end. It is also supported by Microsoft with outstanding performance. LLVM is a set of libraries for building compilers, allowing you to make optimizers and backends more sophisticated. This part allows each of them to build their own compiler by implementing their own features for commercial purposes. Actual compiler tools require C ++ standard libraries, linkers, and loaders in addition to these three. Libraries are essential when using basic functions such as vector, cout, etc., and machine code generated in the backend requires tools such as a linker / loader to integrate with the library. Developing all of this within a period of about 10 months is virtually impossible at a time / cost level, so I have configured it to work with the free distributed standard C ++ libraries and the MinGW linker / loader [8]. Details will be discussed in the next chapter.

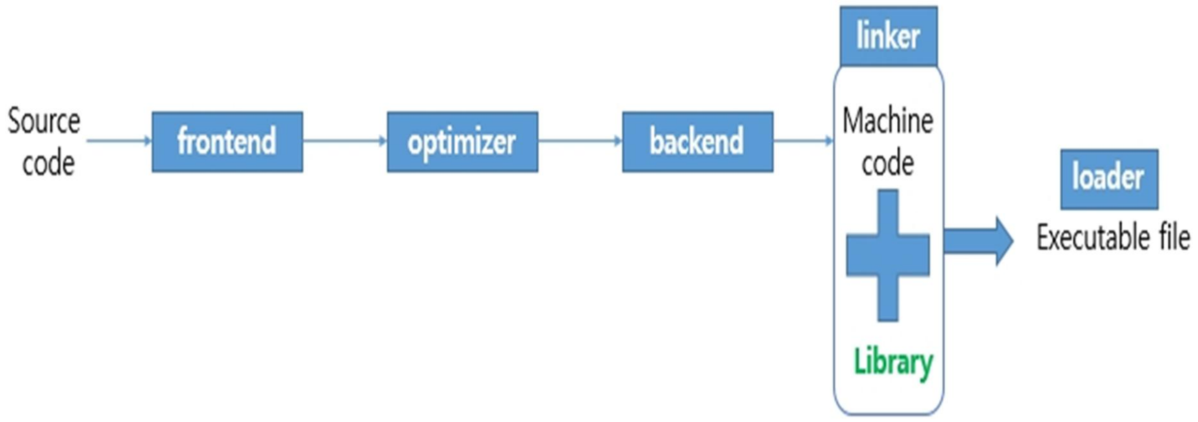


Figure 2. Compiler, library and linker flow for generating executable code

2. Components of LLVM

LLVM can take the intermediate representation (IR) code from the front end (another name is a programming language parser) to provide the intermediate code of the complete compiler system, generating an optimized IR. This new IR can then be transformed to generate machine-dependent assembly language code for the target platform (Intel, AMD, MIPS, etc.). LLVM accepts IR from the GNU compiler collection

(GCC front-end) toolchain (or clang) so it can be used with a variety of existing compilers written for this project. LLVM can generate machine code that can be relocated in binary machine code at compile time, link time, or runtime.

LLVM supports language-independent instruction set and type systems. Each instruction is created as a Static Single Assignment. That is, each variable, called a data type specified register, is allocated once and then fixed. This helps to simplify the analysis of dependencies between variables. With LLVM, you can compile code statically, as in an existing GCC system, or compile it later from the IR to machine code via just-in-time compilation (JIT) [9]. The data type system consists of five basic types, such as integer or floating-point numbers, and five derived types, such as pointers, arrays, vectors, structures, and functions. A class in C ++ can represent an array of structures, functions, and function pointers.

- Front End

LLVM was originally written to replace the existing code generator in the GCC stack, and many GCC frontends have been modified and used. LLVM now supports compilation of Ada, C, C ++, D, Delphi, Fortran, Haskell, Objective-C, and Swift using a variety of front ends. Some are derived from the GNU Compiler Collection (GCC) versions 4.0.1 and 4.2.

As interest in LLVM increased, efforts were made to develop new front ends for various languages. Most notable is Clang, a new compiler that supports C, C ++, and Objective-C. Clang, primarily supported by Apple, aims to replace the C / Objective-C compiler in the GCC system with a system that is more easily integrated with the integrated development environment (IDE) and supports multithreading more widely. Support for the OpenMP directive has been included in Clang since release 3.8.

- Intermediate representation

At the heart of LLVM is intermediate representation (IR), a low-level programming language similar to assembly. IR is a robust set of reduced instruction set computing (RISC) instructions that abstract the subject's details. For example, a calling convention is abstracted through the call and ret commands using explicit arguments. Also, instead of a fixed set of registers, IR uses a temporary set of types such as %0, %1, and so on. LLVM supports 3-address format IR, a human-readable assembly format.

A simple "Hello, world!" Example program in IR format:

```
@.str = internal constant [14 x i8] c "hello, world¥0A¥00"

declare i32 @printf(i8*, ...)

define i32 @main(i32 %argc, i8** %argv) nounwind {
entry:
   %tmp1 = getelementptr [14 x i8]* @.str, i32 0, i32 0
   %tmp2 = call i32 (i8*, ...)* @printf( i8* %tmp1 ) nounwind
   ret i32 0
}
```

Figure 3. Intermediate language expression example generated by LLVM

The LLVM Machine Code Project is a framework of LLVM for machine instruction translation between text format and machine code. Previously, LLVM converted assemblies to machine code using assemblies provided by the system assembler or toolchain. The integrated assembler for the LLVM Machine Code project supports most LLVMs, including x86, x86-64, ARM, and ARM64 [10].

- Linker

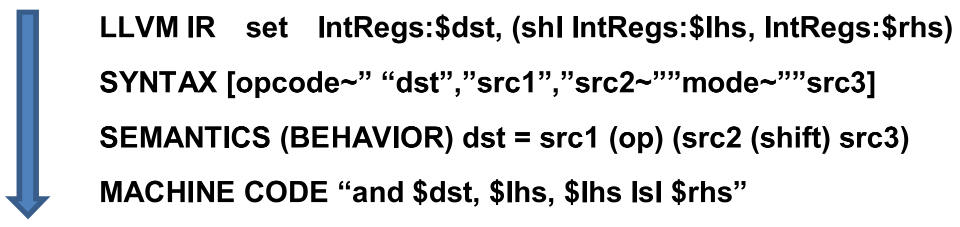
The lld subproject is an attempt to develop a platform-independent linker for LLVM. lld aims to remove dependencies on third-party linkers. Current lld supports ELF, PE / COFF and Mach-O in order of completion [11]. If lld is insufficient, you can use another linker such as GNU ld. Link-time optimization is possible using lld. When link-time optimization is enabled, the compiler generates the LLVM bit code instead of the source code, and the linker performs the source code generation.

- C ++ Standard Library

The LLVM project includes a dual-licensed C ++ standard library under the MIT license and the Illinois State University license. If you cannot use lld because you need to work with the linker, use the distribution provided by the linker toolchain.

- Developed LLVM Compiler backend

How to configure the compiler with the LLVM framework is described in detail in the LLVM document. Using the LLVM application programming interface (API) [12], the LLVM tools llc and lli llvm-gcc can be used to generate LLVM IRs. Here is a brief overview of the overall compiler overall configuration.



```
LLVM IR    set    IntRegs:$dst, (shl IntRegs:$lhs, IntRegs:$rhs)
SYNTAX [opcode~” “dst”,”src1”,”src2~””mode~””src3]
SEMANTICS (BEHAVIOR) dst = src1 (op) (src2 (shift) src3)
MACHINE CODE “and $dst, $lhs, $lhs lsl $rhs”
```

Figure 4. Code generation example

Compiler is usually divided into parts that convert the code written in programming languages such as C / C ++ / Java into Intermediate Representation (IR) that is independent of the target architecture, and convert the IR into machine code of the target architecture. To create a compiler for the processor we want to develop, you can implement the part that converts LLVM IR to machine code. To do this, it is necessary to map each instruction of LLVM IR to appropriate instruction of target architecture as shown in Figure 4. It is necessary to understand the meaning of LLVM IR command, to compare the meaning and syntax of each instruction of target architecture, and to select appropriate command and perform mapping.

- Frame Lowering

When a function call is made in a program, memory is allocated in memory for the purpose of storing the local variables of the function and the arguments of the function. This is called the frame of the corresponding function. When the function is called, a frame is allocated. When the function is finished and returned to the original function, the frame must be released. The compiler must generate these sequences at the beginning and end of the function using the appropriate commands in the target architecture.

- Expand Pseudo Instructions

When mapping from LLVM IR to Machine code, all commands are not generated as actual commands. For example, a command that assigns a large immediate value to a register is generated as a pseudo code called LOAD_IMMEDIATE, and then converted to an actual instruction at the end of the compiler. Since the size of immediate that can be encoded in a command is limited, in order to allocate a larger immediate to a register, the upper bit and the lower bit must be allocated twice. This is not done in the mapping phase, but instead, the LOAD_IMMEDIATE command is replaced with the appropriate target architecture command at the end.

3. Conclusion

The compiler is one of the most sophisticated system software. It usually takes two to three years or more to develop, and development costs are also high. Using the LLVM based development method presented in this paper will help to save development period and development cost. The development of compiler-centric prototyping tools can be accelerated because the compiler development takes most of the time.

In this paper, we describe the compiler development process in developing prototyping tool using LLVM. Rapid development of prototyping tools can shorten overall system development time, which has a direct impact on cost savings. In the prototype tool, the compiler is a particularly time-consuming tool. Shortening the development time of the compiler is essential for system development. The results of this study will help to shorten compiler development time. In the next work, we will explain the development process of the compiler's optimization pass.

Acknowledgement

This work was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education (NRF - 2018R1D1A1 B07050054).

References

[1] Philip Machanick, "Approaches to Addressing the Memory Wall," Technical Report, School of IT and Electrical Engineering, University of Queensland, 22p, November, 2002.
[2] GCC, the GNU Compiler Collection, online: https://gcc.gnu.org/
[3] Clang: a C language family frontend for LLVM, online: https://clang.llvm.org/
[4] Visual Studio 2017, online: https://visualstudio.microsoft.com/ko/vs/features/cplusplus/
[5] GNU Operating System, online: https://www.gnu.org/licenses/gpl-3.0.html
[6] University of Illinois/NCSA Open Source License,
online: https://en.wikipedia.org/wiki/University_of_Illinois/NCSA_Open_Source_License
[7] Compiler, online: https://en.wikipedia.org/wiki/Compiler
[8] Home of the MinGW and MSYS Projects, online: http://www.mingw.org/
[9] Just-in-time compilation, online: https://en.wikipedia.org/wiki/Just-in-time_compilation
[10] ARM architecture, online: https://en.wikipedia.org/wiki/ARM_architecture
[11] Portable Executable, online: https://en.wikipedia.org/wiki/Portable_Executable
[12] Application programming interface,
online: https://en.wikipedia.org/wiki/Application_programming_interface