In-depth exploration of LLVM

ZiYao Yin
12011524@mail.sustech.edu.cn
Southern University of Science and Technology
Shen Zhen, China

He Zhu
12012506@mail.sustech.edu.cn
Southern University of Science and Technology
Shen Zhen, China

ZhiXin Wang
12012502@mail.sustech.edu.cn
Southern University of Science and Technology
Shen Zhen, China

Keywords: LLVM, SSA, GCC

ACM Reference Format:
ZiYao Yin, He Zhu, and ZhiXin Wang. 2023. In-depth exploration of LLVM. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

LLVM (Low Level Virtual Machine) is a compiler infrastructure that was developed at the University of Illinois at Urbana-Champaign. It was created to provide a modern, portable, and highly optimized compiler infrastructure that can be used to develop compilers, programming tools, and other utilities. LLVM was first released in 2003, and has since become a popular choice for compiler development due to its modular design and extensive optimization capabilities. LLVM is written in C++ and is designed to be used as a library, allowing developers to easily incorporate LLVM's capabilities into their own programs. In this article, we will look at the original design of LLVM and see what is clever about llvm compared to a highly integrated compiler like GCC from a system design perspective.

2 Design principle

LLVM is a collection of libraries to support compiler development and related tasks. Each library supports a specific component of a typical compiler pipeline (lexicography, parsing, type-specific optimization, architecture-specific machine code generation, etc.). What makes it so popular is that its modular design allows its functionality to be very easily adapted and reused. It is convenient when you develop a compiler for an existing language to target a new hardware architecture, where you only need to write the specific hardware component and all the lexicography, parsing, machine independent optimization, etc. is handled for you, or when you develop a compiler for a new language and all the back-end stuff is handled for you. In general, LLVM was designed with a number of principles

1. Modularity: LLVM is highly modular, with a clear separation of concerns between different components. This makes it easy to understand and maintain the codebase, and allows developers to reuse components in different contexts.
2. Portability: LLVM is also designed to be portable, and can be used on a wide variety of platforms and architectures.
3. Efficiency: LLVM is designed to generate highly efficient code, and includes a number of optimization passes to improve the performance of the generated code.
4. Generality: LLVM is also a general-purpose compilation framework, and can be used to build compilers for a wide range of programming languages.
5. Adaptability: Extensiblility is also of great importance, and it includes a number of hooks and extension points that allow developers to customize and extend its behavior.

Next, we will explore some details of the implementation of llvm to gain a deeper appreciation of these design concepts and further understand this excellent compiler design.

3 SSA-based instruction set

LLVM was originally positioned as a relatively low-level virtual machine. It was designed to solve the problem of compiler code reuse, and LLVM came up with the LLVM IR, an intermediate code representation language that takes into account various application scenarios, such as calling LLVM in the IDE for real-time code syntax checking, compilation and optimization of static and dynamic languages, etc. The LLVM representation is language independent, allowing all code of a program, including system libraries and parts written in different languages, to be compiled and optimized together. The LLVM representation is language independent, allowing all code of a program, including system libraries and parts written in different languages, to be compiled and optimized together.

3.1 SSA-form

Static Single Assignment (SSA) is also recomended in class, It is called "static" because it does not change during the execution of the program, and "single assignment" because each variable is assigned a value exactly once. In SSA form, each variable is replaced with a unique version that represents a specific assignment to that variable.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

SSA form is particularly useful for optimizing code because it makes it easier to perform optimization passes on the code. For example, the compiler can easily identify and eliminate redundant computations, or reorder computations to take advantage of processor features such as out-of-order execution.

3.2 Implementation of virtual registers

The LLVM instruction set captures the key operations of ordinary processors but avoids machine-specific constraints such as physical registers, pipelines. LLVM provides an infinite set of typed virtual registers which can hold values of primitive types such as boolean, integer, floating point, and pointer. The virtual registers are in SSA form.

LLVM is a load/store architecture: programs transfer values between registers and memory solely via load and store operations using typed pointers. In LLVM, all addressable objects ("lvalues") are explicitly allocated. Global variable and function definitions define a symbol which provides the address of the object, not the object itself. This gives a unified memory model in which all memory operations, including call instructions, occur through typed pointers. There are no implicit accesses to memory, simplifying memory access analysis, and the resentation needs no "address of" operator.

Most instructions of llvm, including all arithmetic and logical operations, are in three-address form: they take one or two operands and produce a single result. And the represent form of these registers are also SSA form, from example, each virtual register is written in exactly one instruction, and each use of a register is dominated by its definition.

3.3 Language-independent type system

3.3.1 Type define. One of the fundamental design features of LLVM is the inclusion of a language-independent type system. Every SSA register and explicit memory object has an associated type, and all operations obey strict type rules. This type information is used in conjunction with the instruction opcode to determine the exact semantics of an instruction. This type information enables a broad class of high-level transformations on low-level code.The LLVM type system includes:

1. primitive types: such as void, bool, signed/unsigned integers of different length and floating-point types.
2. four derived types: pointers, arrays, struc- tures, and functions.

Because most high-level language data types are eventually represented using some combination of these four types in terms of their operational behavior, so that this makes it possible to write portable code using these types.

3.3.2 Type safety and DSA. Because LLVM is language independent, declared type information in a legal LLVM program may not be reliable, for example, a wrong type pointer target may cause error in memory access. So pointer analysis algorithm must be used to distinguish memory accesses for which the type of the pointer target is reliably or not. LLVM use a flow-insensitive, field-sensitive and context- sensitive points-to analysis called Data Structure Analysis (DSA), As part of the analysis, DSA extracts LLVM types for a subset of memory objects in the program. It does this by using declared types in the LLVM code as speculative type information, and checks conservatively whether memory accesses to an object are consistent with those declared types.

3.3.3 Address calculate. Address calculation is also a problem for preserving type information design in LLVM. However LLVM use the **getelementptr** instruction to perform pointer arithmetic in a way that both preserves type information and has machine-independent semantics. Given a typed pointer to an object of some aggregate type, this instruction calculates the address of a sub-element of the object in a type-preserving manner.

Another advantage of this design is that making all address arithmetic explicit makes it be exposed to all LLVM optimizations, which is quiet useful in reassociation and redundancy elimination.

4 Compilation workflow and optimization

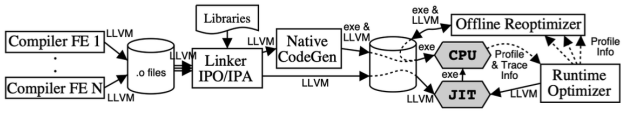

Figure 1. Compilation workflow.

4.1 Language-dependent optimization

The LLVM compiler front-end translates the source language program into the LLVM virtual instruction set. Each static compiler can perform a number of key operations, such as performing language-specific optimizations; translating the source program into LLVM code, synthesizing as much useful LLVM type information as possible, especially exposing pointers, structures, and arrays; and calling LLVM channels at the module level for global or inter-program optimizations. LLVM optimizations are built into the library, which makes it easy for the front-end to use them .

Many of the high-level optimizations in this process are not really language-dependent, and are often special cases of more general optimizations performed on LLVM code. In such cases, it is better to extend the LLVM optimizer to perform the conversion rather than to perform poorly reusable optimizations on the code for a specific front-end. And such optimizations can be performed at any stage of the compiler, which is one of the key design concepts of LLVM.

4.2 Linker time optimization and IPO

Interprocedural optimization (IPO) is a technique used to improve the performance of compiled code by analyzing and optimizing the relationships between different functions in a program. IPO analyzes the relationships between functions and optimizes them as a group. It is intended to reduce or eliminate repeated computations, inefficient use of memory, and simplify iterative sequences such as loops. If another routine is called in a loop, IPO analysis may determine that it is best to inline it. In addition, the IPO may reorder the routine to obtain a better memory layout and location.

For example, consider a program that consists of two functions: 'func1' and 'func2'. If 'func1' calls 'func2' many times, it may be possible to optimize the relationship between these two functions by inlining 'func2' into 'func1', eliminating the overhead of the function call. This optimization would not be possible if each function was optimized in isolation.

For languages that compile on a file-by-file basis, effective IPO across translation units (module files) requires knowledge of the "entry points" of the program so that a whole program optimization (WPO) can be run, this is implemented as a link-time optimization (LTO) pass in LLVM, because the whole program is visible to the linker.

In compiler optimization, link time optimization is a technique that allows the compiler to perform optimization passes across multiple object files at link time. This can enable the compiler to make more aggressive optimization decisions that would not be possible if the object files were optimized in isolation. And Link time is the first phase of the compilation process where most of the program is available for analysis and transformation. So link-time is a natural place to perform aggressive interprocedural optimizations across the entire program. The link-time optimizations in LLVM operate on the LLVM representation directly, taking advantage of the semantic information it contains.

4.3 Code generation and JIT

Before execution, a code generator is used to translate from LLVM to native code for the target platform in one of two ways. In the first option, the code generator is run statically at link time or install time. If the user decides to use the post-link (runtime and offline) optimizers, a copy of the LLVM bytecode for the program is included into the executable itself. Alternatively, a just-in-time (JIT) Execution Engine can be used which invokes the appropriate code generator at runtime, translating one function at a time for execution.

Here, JIT is a technique used by some compilers to compile code at runtime, rather than ahead of time (AOT). This allows the compiler to optimize the code for the specific hardware and operating system on which it is running, potentially resulting in better performance. In AOT compilation, the compiler processes the source code and generates machine code that can be directly executed by the processor. This machine code is typically stored in a standalone executable file or in a shared library. In contrast, JIT compilation involves compiling the source code at runtime, either as it is needed or in advance of when it will be executed. This allows the compiler to generate machine code that is optimized for the specific hardware and operating system on which the program is running.

4.4 Runtime path profiling

Runtime path profiling is a technique used to gather data about the execution of a program at runtime. This data can be used to guide the optimization process, potentially resulting in better performance.

There are several different ways to gather data for runtime path profiling. One common technique is to use instrumentation, which involves inserting special code into the program that records information about its execution. This can include data such as the frequency with which different code paths are taken, the values of variables at different points in the program, and the order in which instructions are executed. Once the data has been gathered, it can be used to reoptimize the program, a process known as reoptimization. This can involve recompiling the program with different optimization options, or modifying the program's machine code at runtime to reflect the data gathered from the profiling process.

In LLVM, thanks to the modularity of the LLVM design and the exposure of a large amount of detailed process information, runtime path profiling is efficietly carried out in LLVM. As a program executes, the most frequently executed execution paths are identified through a combination of offline and online instrumentation. The offline instrumentation identifies frequently executed loop regions in the code. When a hot loop region is detected at runtime, a runtime instrumentation library instruments the executing native code to identify frequently-executed paths within that region. Once hot paths are identified, LLVM will duplicate the original LLVM code into a trace, perform LLVM optimizations on it, and then regenerate native code into a software-managed trace cache. Then it will insert branches between the original code and the new native code.

4.5 Profile-guided optimization

Profile-guided optimization (PGO) is a technique used in compiler optimization to improve the performance of compiled code. It works by using data gathered from the execution of a program to guide the optimization process, resulting in more efficient machine code. It can improves application performance by shrinking code size, reducing branch mispredictions, and reorganizing code layout to reduce instruction-cache problems. PGO provides information to the compiler about areas of an application that are most frequently executed. By knowing these areas, the compiler is able to be more selective and specific in optimizing the application.

PGO is particularly effective at optimizing code that exhibits a high degree of runtime variability, such as code that makes different decisions based on the input data. It can also be used to optimize code that exhibits a high degree of locality, such as code that accesses data in a predictable pattern.

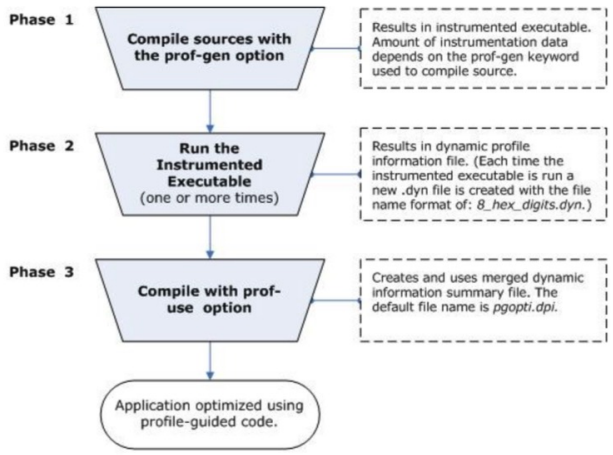

Figure 2. Profile-guided optimization.

LLVM is able to carry out transparent offline optimization of applications during idle-time on an end-user's system, so it apply an offline, idle-time reoptimizer. This optimizer can use profile information gathered from end-user runs of the application. It can even reoptimize an application multiple times in response to changing usage patterns over time. What' more, it can tailor the code to detailed features of a single target machine, so that code can be run on many different machine of different architectures and operating systems.

Unlike the runtime optimizer, PGO at can perform much more aggressive optimizations because it is run offline. Nevertheless, runtime optimization can further improve performance because of the ability to perform optimizations based on runtime values. So LLVM will organize the runtime and offline reoptimizers to ensure the highest achievable performance.

5 Conclusion

After a deep dive into LLVM, we can see deep knowledge and understanding of the compiler system of Chris Lattner, who is the leader designer of LLVM, both in terms of the overall design philosophy and the subtleties of the implementation. Unlike GCC's highly integrated design, LLVM is clearly more flexible, developer-friendly, and extensible. Although llvm has changed a lot, Chris Lattner is still one of the most important contributors today, and we can still learn a lot from his work 20 years ago, and appreciate the charm of an excellent compiler system design.

References

[1] https://en.wikipedia.org/wiki/Interprocedural_optimization
[2] https://en.wikipedia.org/wiki/Just-in-time_compilation
[3] https://en.wikipedia.org/wiki/Profile-guided_optimization
[4] LLVM: A Compilation Framework for Lifelong Program Analysis Transformation, a published paper by Chris Lattner, Vikram Adve
[5] https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming/profile-guided-optimization-pgo.html