Faculty of Engineering of the University of Porto

Master in Informatics and Computing Engineering

Software Systems Architecture

Understand Someone Else's Architecture — LLVM

Homework 04

Team 32

José Francisco Veiga up202108753@up.pt

Marco Vilas Boas up202108774@up.pt

Pedro Lima up202108806@up.pt

Pedro Januário up202108768@up.pt

Pedro Marcelino up202108754@up.pt

March 2025

Introduction

LLVM is a well-known open-source compiler. The purpose of this document is to describe its architecture, as we could perceive from Chapter 11 of Volume I of the book The Architecture of Open Source Applications [3].

Architecture summary

As we know, compilers are used to translate a high level language, such as C, into machine code that can directly execute on the hardware, or bytecode that can be run in an interpreter or virtual machine.

A compiler is usually divided in 3 parts: the front-end, the intermediate-representation layer and the back-end. This makes it, in theory, easier to add support to a new language (just make a new front-end) and to add support to a new target architecture (just make a new back-end).

Historically, compiler implementations, like GCC, suffered from significant architectural limitations, because they were implemented as big monoliths, with tightly coupled components. This makes it very hard to reuse certain parts of the compiler, limits how flexible it can be in supporting multiple languages and targets and makes the overall process of development much harder for humans.

The LLVM project, that started in the year 2000, took a different, innovative approach – instead of a big, monolithic application that relies heavily on global variables and has code from all parts reaching relying on each other, LLVM fundamentally reimagines compiler architecture through a modular, library-based design that addresses these traditional challenges.

The key architectural patterns that distinguish this project are:

- LLVM Intermediate Representation, a first-class language: this language serves as a universal, self-contained code representation that is completely independent from any programming language and from any target processor. It also supports both serialization and deserialization, which allows for some cool tricks and use cases such as link-time code generation. It also provides a consistent and well-defined interface for all compiler phases, making it easier to use modules in different projects.
- Modular Library based design: the compiler components are implemented as loosely-coupled libraries, which allows for ‘pick and choose’ optimizations and code generation according to one’s needs. It also provides the advantage that only the components used need to be linked, which reduces overhead.
- Flexible Compilation Phases: even though LLVM features the classic three-phase compiler design, each phase can be independently configured, which, besides the traditional benefits of supporting multiple source languages and targets, also enables just-in-time and link-time optimizations.

Compared with traditional compilers, LLVM is a perfect example on how a good architecture can solve many of the problems and allow for better software use.

LLVM also introduces two key innovations: Target Description Files, which allow defining hardware targets declaratively, allowing consistency across different tools, and Advanced Testing, which allows to test each optimization alone, automatically reduces and generates test cases for failures, and makes it easy to implement regression testing.

LLVM transforms compiler infrastructure by treating compiler components as flexible, interchangeable libraries. This approach gives developers unprecedented freedom in building and customizing compilation toolchains, and has already had a huge impact in the world.

Diagrams

In this section, we present some diagrams that illustrate and hopefully help better understand LLVM's architecture.

Traditional three-phase compiler architecture

LLVM follows the traditional three-phase compiler architecture. The frontend component is responsible for parsing and validating the input code, then translating it into LLVM's Internal Representation. The IR is optionally fed through a series of analysis and optimization passes which improve the code, which is finally sent into a code generator to produce native machine code.

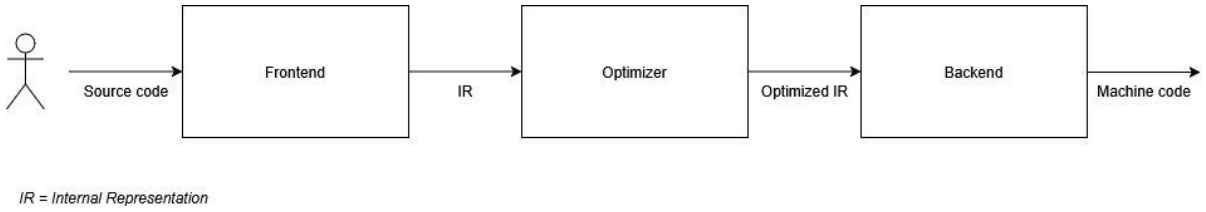

Fig. 1: Three-phase compiler architecture.

IR "transactions"

As stated in the reference book, LLVM's "magic" resides around the optimizer, in such a way that "[t]he most important aspect of its design is the LLVM Intermediate Representation". Thus, it is worth illustrating how IR code is handled and transmitted between the Frontend and the Optimizer. The modularity (namely, the separation between the two latter components) is complemented by a conversion tool that allows capturing the generated IR and saving it in a convenient on-disk format ("bitcode").

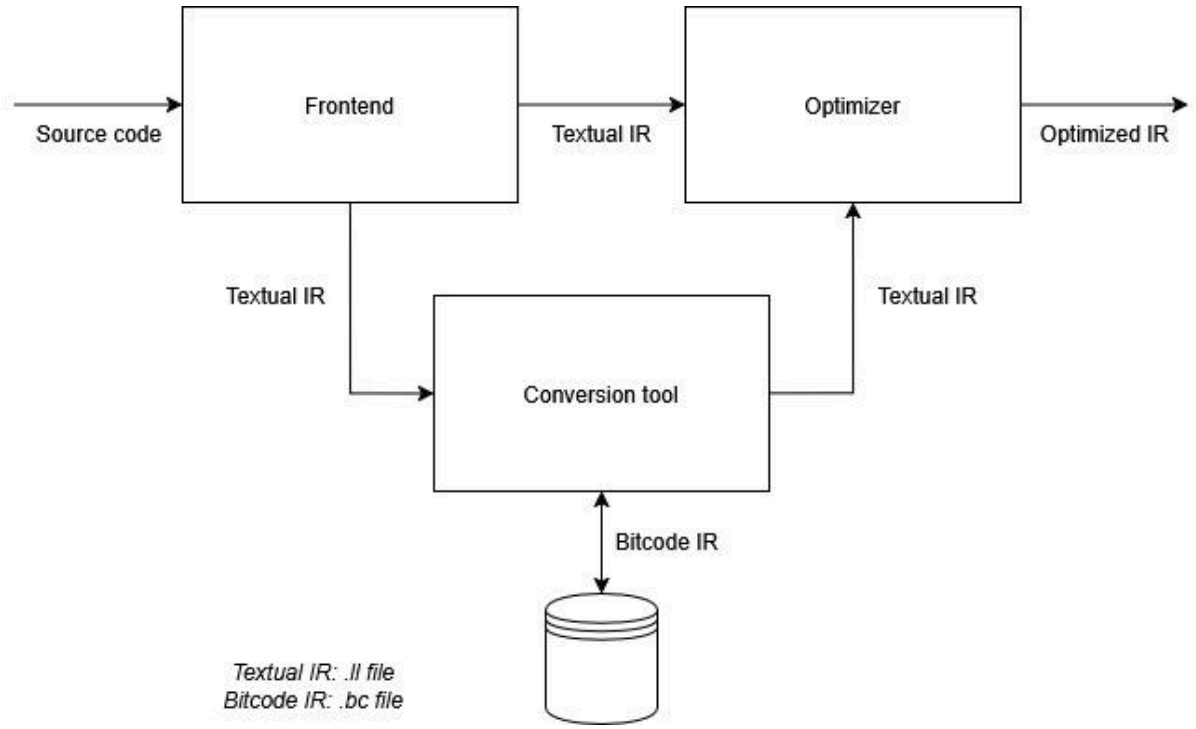

Fig. 2: Detail of the IR transactions between the Frontend, the Optimizer and the auxiliary IR Conversion Tool.

Architectural patterns

LLVM has a highly modular architecture and to achieve this, LLVM employs various architectural and design patterns. Here we will see some of them, demonstrating how they contribute to LLVM's efficiency and adaptability.

Interpreter

The most apparent architectural pattern in LLVM is the Interpreter pattern, as described in *Pattern-Oriented Software Architecture, Volume 4* [2].LLVM is fundamentally a family of compilers, making it naturally aligned with the principles of the Interpreter pattern. The whole objective of this system is to parse a high-level language into machine code. To achieve this, it repeats the Interpreter pattern multiple times, transforming the high-level code into LLVM IR and then LLVM IR into machine code.

Division of Tasks

LLVM is structured into three distinct components: the frontend, the optimizer, and the backend. This separation of concerns aligns with multiple architectural patterns described in *Pattern-Oriented Software Architecture, Volume 1* [1], namely Pipes and Filters, Layers, and Broker.

Pipes and Filters

The LLVM compilation process can be viewed through the Pipes and Filters pattern. The various programming languages, including high-level languages, LLVM IR, the TableGen

(tblgen) language, and machine code, function as pipes. Meanwhile, the frontend, optimizer, and backend act as filters that transform the data at each stage. This pattern allows for modular processing, where each stage refines the input before passing it to the next stage.

Layers

Another perspective is to consider the LLVM architecture as an implementation of the Layers pattern. The compiler can be seen as comprising three distinct layers that interact only with adjacent layers. The frontend, optimizer, and backend serve as separate layers, with the LLVM Intermediate Representation acting as the interface between them. The first and third layers, frontend and backend, can be replaced independently as long as the new components conform to the established LLVM IR interface. This layering facilitates maintainability and scalability by ensuring that changes in one layer do not necessitate modifications in others.

Broker

Although LLVM is not a distributed system, its architecture can also be analyzed using the Broker pattern. At first glance, this classification may seem unconventional, but the optimizer works as a mediator between various frontends and backends, effectively coordinating their interactions through the LLVM IR. This structure mirrors the Broker pattern described in *Pattern-Oriented Software Architecture, Volume 1* [1], where a broker component facilitates communication between decoupled components. While LLVM's components typically operate within the same computer, they are modular enough that, hypothetically, they could be deployed on separate systems, with minimal architectural changes required for distributed execution. This modularity enables easy exchange between different languages and target architectures.

Encapsulated Implementation

LLVM is structured primarily as a library, with most of its functionalities provided as modular components that can be reused or extended. This modular design aligns closely with the Encapsulated Implementation pattern, as detailed in *Pattern-Oriented Software Architecture, Volume 4* [2], which emphasizes hiding implementation details behind well-defined interfaces. Each compiler optimization is encapsulated within a “Pass” class, following a template-based approach that ensures a clear separation between interface and implementation. This allows LLVM users to invoke optimizations without concerning themselves with their internal workings, ensuring that the system is adaptable and resilient to any future changes.

Quality attributes

When analyzing the architecture of such a complex system like LLVM, it’s not enough to understand *what* it does: we must also focus on *how well* it does it, *how* it performs. Also commonly known as non-functional requirements, these quality attributes describe key aspects of software performance, influencing design decisions, usability and adaptability.

They are particularly important in this specific case of LLVM, as this is a system designed as a foundational infrastructure for compilers, tools and applications.
Below, we identify the most important quality attributes we found that characterize LLVM's design and implementation.

Modularity

LLVM is intentionally "designed as a set of reusable libraries with well-defined interfaces", rather than a monolithic compiler. Each component, such as the optimizer, code generator or front end, can be developed, extended and used independently. This ensures modularity, facilitating flexible combinations of components and enabling developers to build custom ones.

Reusability

The architecture of LLVM emphasizes reusability across "a broad variety of statically and runtime compiled languages", platforms and use cases. The LLVM IR acts as a neutral, language-agnostic format, allowing any front end to plug into the system. On the back end, multiple target architectures can be supported by adding specific code generators. This design covers projects like Clang, OpenCL and others to reuse large portions of LLVM infrastructure.

Performance

LLVM generates code with a high performance through several mechanisms, including a powerful optimization pipeline and support for link-time and install-time optimization (LTO & ITO). The system allows for tuning the performance of specific target hardware, such as instruction scheduling or register pressure reduction. Its IR is designed for optimizations, enabling transformations and analysis that are both powerful and efficient.

References

[1] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley.
[2] Buschmann, F., Henney, K., & Schmidt, D. C. (2007). *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing*. Wiley.
[3] Brown, A., & Wilson, G. (Eds.). (2011). *The Architecture of Open Source Applications, Volume 1*. Lulu.com.