Experiment No. 11

AIM: Case study on LLVM tool

THEORY: What is LLVM?

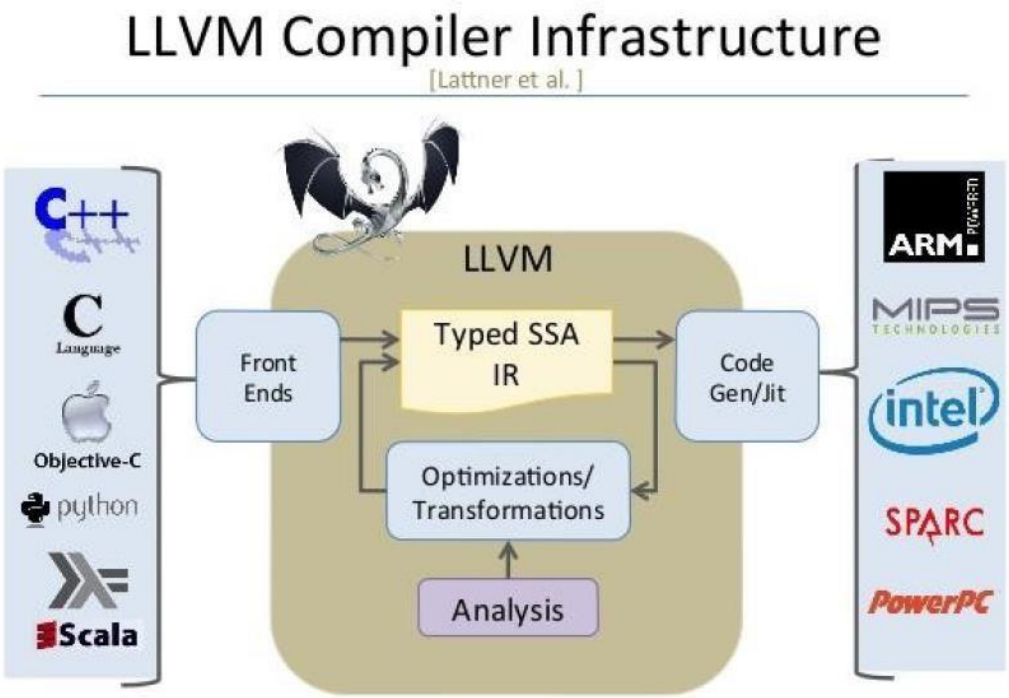
- LLVM is an acronym that stands for low level virtual machine. It also refers to a compiling technology called the LLVM project, which is a collection of modular and reusable compiler and toolchain technologies.
- The LLVM project has grown beyond its initial scope as the project is no longer focused on traditional virtual machines.
- LLVM is a compiler and a toolkit for building compilers, which are programs that convert instructions into a form that can be read and executed by a computer.
- The LLVM project is a collection of modular and reusable compiler and tool chain technologies. LLVM helps build new computer languages and improve existing languages.
- It automates many of the difficult and unpleasant tasks involved in language creation, such as porting the output code to multiple platforms and architectures.

HOW DOES A LLVM COMPILER WORK ?

- On the front end, the LLVM compiler infrastructure uses clang —a compiler for programming languages C, C++ and CUDA - to turn source code into an interim format. Then the LLVM clang code generator on the back end turns the interim format into final machine code.
- The compiler has five basic phases:
- Lexical Analysis - Converts program text into words and tokens (everything apart from words, such as spaces and semicolons).
- Parsing - Groups the words and tokens from the lexical analysis into a form that makes sense.
- Semantic Analyser — Identifies the types and logics of the programs.

- Optimization - Cleans the code for better run-time performance and addresses memory-related issues.
- Code Generation - Turns code into a binary file that is executable LLVM COMPILER

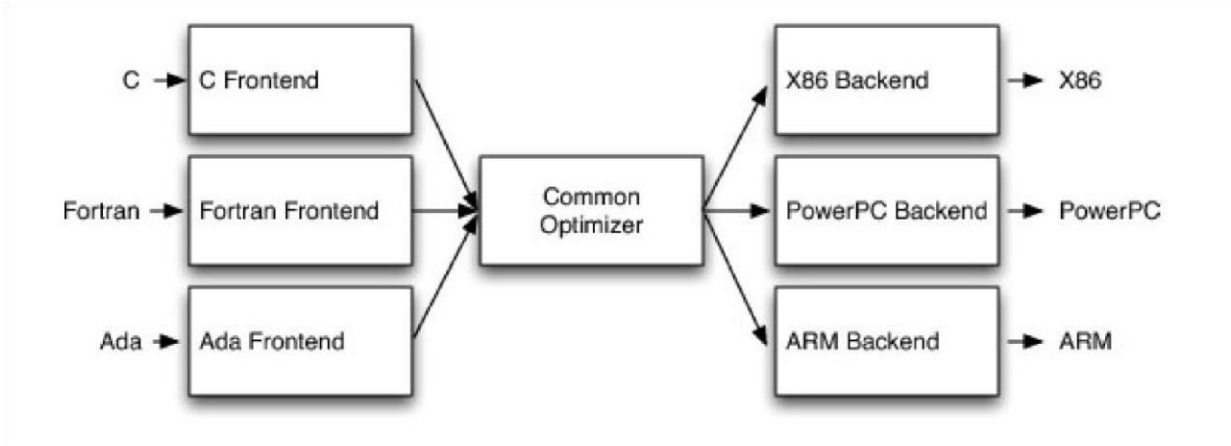
ARCHITECTURE:



At the frontend you have Perl, and many other high level languages. At the backend, you have the natives code that runs directly on the machine.

At the centre is your intermediate code representation. If every high level language can be represented in this LLVM IR format, then analysis tools based on this IR can be easily reused - that is the basic rationale.

LLVM IR:



The LLVM project/infrastructure: This is an umbrella for several projects that, together, form a complete compiler: frontends, backends, optimizers, assemblers, linkers, libc++, compiler-rt, and a JIT engine. The word "LLVM" has this meaning, for example, in the following sentence: "LLVM consists of several projects".

An LLVM-based compiler: This is a compiler built partially or completely with the LLVM infrastructure. For example, a compiler might use LLVM for the frontend and backend but use GCC and GNU system libraries to perform the final link. LLVM has this meaning in the following sentence, for example: "I used LLVM to compile C programs to a MIPS platform".

LLVM libraries: This is the reusable code portion of the LLVM infrastructure. For example, LLVM has this meaning in the sentence: "My project uses LLVM to generate code through its Just-in-Time compilation framework".

LLVM core: The optimizations that happen at the intermediate language level and the backend algorithms form the LLVM core where the project started. LLVM has this meaning in the following sentence: "LLVM and Clang are two different projects".

The LLVM IR: This is the LLVM compiler intermediate representation. LLVM has this meaning when used in sentences such as "I built a frontend that translates my own language to LLVM".

Three Primary LLVM Components:

The LLVM Virtual Instruction Set –

1. The common language- and target-independent IR
2. Internal (IR) and external (persistent) representation

A collection of well-integrated libraries - Analyses, optimizations, code generators, JIT compiler, garbage collection support, profiling, ...

A collection of tools built from the libraries - Assemblers, automatic debugger, linker, code generator, compiler driver, modular optimizer,..

<u>Why LLVM ?</u>

- A need for a compiler that allows better diagnostics
- Better integration with IDEs, a license that is compatible with commercial products.

- Fast compiler that is easy to develop and maintain.
- Useful for performing optimizations and transformations on code

Code Generation Using LLVM:

The LLVM target-independent code generator is a framework that provides a suite of reusable components for translating the LLVM internal representation to the machine code for a specified target—either in assembly form (suitable for a static compiler) or in binary machine code format (usable for a JIT compiler).

The LLVM target-independent code generator consists of six main components:

1. Abstract target description interfaces which capture important properties about various aspects of the machine, independently of how they will be used. These interfaces are defined in include/llvm/Target/.
2. Classes used to represent the code being generated for a target. These classes are intended to be abstract enough to represent the machine code for any target machine. These classes are defined in include/llvm/CodeGen/. At this level, concepts like “constant pool entries” and “jump tables” are explicitly exposed.
3. Classes and algorithms used to represent code at the object file level, the MC Layer. These classes represent assembly level constructs like labels, sections, and instructions. At this level, concepts like “constant pool entries” and “jump tables” don’t exist.

4. Target-independent algorithms used to implement various phases of native code generation (register allocation, scheduling, stack frame representation, etc). This code lives in lib/CodeGen/.

5. §. Implementations of the abstract target description interfaces for particular targets. These machine descriptions make use of the components provided by LLVM, and can optionally provide custom target-specific passes, to build complete code generators for a specific target. Target descriptions live in lib/Target/.
6. 6. The target-independent JIT components. The LLVM JIT is completely target independent (it uses the TargetJlTlnfo structure to interface for target-specific issues. The code for the target- independent JIT lives in lib/ExecutionEngine/JIT.

Depending on which part of the code generator you are interested in working on, different pieces of this will be useful to you. In any case, you should be familiar with the target description and machine code representation classes. If you want to add a backend for a new target, you will need to implement the target description classes for your new target and understand the LLVM code representation. If you are interested in implementing a new code generation algorithm, it should only depend on the target-description and machine code representation classes, ensuring that it is Portable.

Uses of LLVM:

LLVM is a modular compiler infrastructure: — Primary focus is on providing good interfaces & robust components — LLVM can be used for many things other than simple static compilers.

LLVM provides language- and target-independent components: — Does not force use of JIT, GC, or a particular object model — Code from different languages can be linked together and optimized.

LLVM is well designed and provides aggressive functionality: — Inter procedural optimization, link-time/install-time optimization today.

CONCLUSION: Thus we have studied about LLVM and code generation using LLVM.