CS6383 : Mini-Assignment #1

Technical Report

Raj Ambekar (CS22MTECH12008)
Soumya Banerjee (CS22MTECH12011)

January 15, 2023

Description and Usage of shell script to generate output :

The file Mini-Asgn-1_CS22MTECH12011.tar.gz needs to be unzipped, which contains the shell script **execute.sh** which needs to be run from the same directory of the submitted file.

The submitted file contains the technical report pdf, input test cases / 5 non-trivial programs (all .c files) along with specific programs related to dead code elimination and constant propagation and the shell script.

Upon running the script, the .c files would be moved into individual folders with the same name as that of the .c files , and the folders would be filled with the required output files such as the LLVM-IR, assembly codes, dom tree, cfg etc. each of which has been described in the following sections.
We have **assumed** that the llvm-project directory which we have cloned is in the home directory of the person running the script so as to successfully execute the clang,opt etc. commands written in the script.

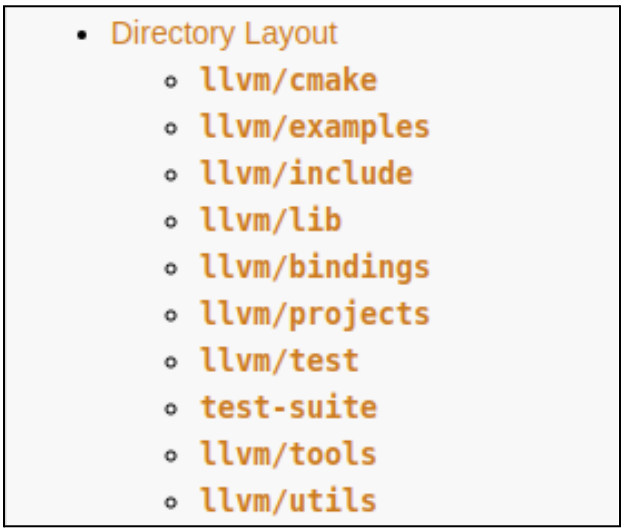
Finally, this technical report gives a detailed description of our understanding and observations that we have come across while exploring the LLVM compiler infrastructure.

Few Key Observations :

We have used the legacy pass manager in our script as the new pass manager wasn't working properly for a few flags like -view-cfg etc. in LLVM 14.
The -dot-cfg wasn't working in our case, the dot file comes out as some garbage value, unlike the -dot-dom which works, due to which we have used -view-cfg to show the cfg as pop-up images on screen while running the script.
We have taken a total of 7 programs for observational purposes and have tried all of the mentioned optimizations, tasks on all of them for exploratory reasons, so the code in our script works on all of the .c files present in the submitted directory.

- **LLVM directory hierarchy description :**

  LLVM is a set of toolchains that contains all the libraries, tools and header files needed to process Intermediate Representations (IR) and convert them into object files.Tools include an assembler, disassembler, bitcode analyzer, and bitcode optimizer. It also contains basic regression tests.
  LLVM directory is a collection of libraries which are used for building, transforming (optimizing) and executing codes.The following is a snippet of the LLVM directory layout :

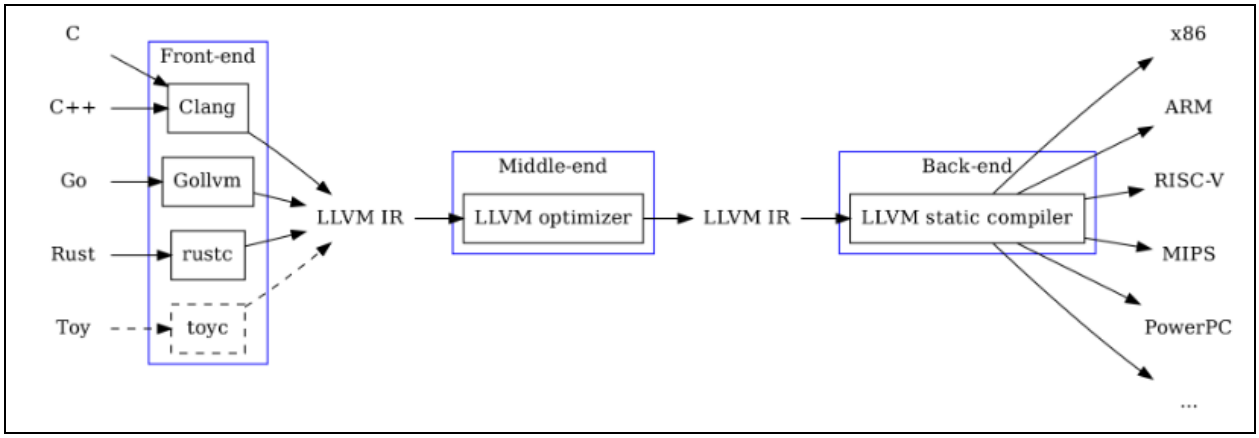


- **LLVM IR:**

  LLVM IR is a low-level intermediate representation used by the LLVM compiler framework. You can think of LLVM IR as a platform-independent assembly language with an infinite number of function local registers.

  Compilers are therefore often split into three components, the front-end, middle-end and back-end; each with a specific task that takes IR as input and/or produces IR as output.

  - Front-end: compiles source language to IR.
  - Middle-end: optimizes IR.
  - Back-end: compiles IR to machine code.



- **Assembly language:**

  The equivalent assembly code of the .c file is generated using the -S option with clang through which we get the equivalent .s file. In compilers name mangling is a technique used to resolve various problems caused by the need to resolve unique names of the programming entities. The clang uses _Z1 prefix and a suffix depending on the data type of parameters passed to the functions to resolve the conflicts.

- **Compiler toolchain and options :**

  a) Optimizations

The optimizations that we have applied to our input cases are -O1, -O2, -O3, -Os and -Oz. -O0 is the default case which does not perform any optimization. -O1 performs some optimization, -O2 performs moderate optimization and -O3 performs more optimizations which make the code

run faster but compile time slower. -Os reduces the executable size, and -Oz aggressively reduces the executable size.
We haven't used -Og since it is equivalent to -O1 and -O4 (and higher) which is equivalent to -O3.

b) Constant propagation and dead code elimination

Constant propagation is an optimization technique in which a known variable containing a constant value is replaced by the value of the constant throughout the function to make the operation faster. It is a local optimization technique.
Consider the following code snippet:
a=100;
b=300+**a**/2;
c=500+(b+**a**)+**a**/23;

The equivalent of the above code after constant propagation can be written as:
a=100;
b=300+**100**/2;
c=500+(b+**100**)+**100**/23;
Constant propagation can be implemented in LLVM using -sccp option along with pass manager.

Dead code elimination : Dead code elimination is a compiler optimization technique which removes the part of code that does not affect the program output. For example a loop condition which always turns out to be true.

```
if(2<3){
    printf("Hello");}
 else{
```

```
        printf("GoodBye");
    }
```

In above code snippet the else part can be removed as it does not have impact on the output and thus the program size can be shrinked.

In LLVM dead code elimination can be implemented using -dce option along with pass manager.

c) Relevant tools in LLVM :

- llvm-as

The assembler transforms the human readable LLVM assembly to LLVM bitcode.

- llvm-dis

The disassembler transforms the LLVM bitcode to human readable LLVM assembly.

- lli

lli is the LLVM interpreter, which can directly execute LLVM bitcode (although very slowly). For architectures that support it (currently x86, Sparc, and PowerPC), by default, lli will function as a Just-In-Time compiler (if the functionality was compiled in), and will execute the code much faster than the interpreter.

- llc

llc is the LLVM backend compiler, which translates LLVM bitcode to a native code assembly file.