Assignment 1: Introduction to Basics of LLVM

Introduction to Program Analysis and Optimization

Deadline: 26th January, 2025

January 10, 2025

Abstract

LLVM is an open-source compiler infrastructure that provides a set of tools and libraries for building compilers, debuggers, and other software development tools. Initially conceived at the University of Illinois at Urbana, LLVM comprises a collection of adaptable and reusable technologies for constructing diverse compilers and related software utilities.

We will heavily rely on the LLVM Compiler infrastructure for our programming assignments. Ideally, you should have access to an x86-based machine, preferably operating on Linux. The primary goal of this initial assignment is to acquaint you with the LLVM infrastructure. The assignment involves installing LLVM from the ground up and exploring its analysis and optimization techniques. This exercise serves as a practical introduction to LLVM's capabilities and functionalities.

1 LLVM Framework Installation (0 pts)

1.1 Required Softwares

We suggest you to check the software requirements for LLVM installation. You can use a conda environment to install the prerequisite packages.

```
conda create -n pa python=3.10
conda activate pa
pip install cmake
```

1.2 Cloning Into Repositories

Clone the LLVM repository from Git Hub and change the version to `release/14.x`. We need to change llvm-version to `release/14.x` in order to ensure consistency among all students. To clone the LLVM repository and change the version, run the two commands below.

```
git clone https://github.com/llvm/llvm-project.git -b release/14.x
```

The repository is around 1.27 GB in size. Before cloning, make sure you have good network connectivity.

1.3 Building LLVM

After successfully cloning LLVM-14, the next step is to build both LLVM and clang from the ground up. LLVM employs CMake to create a build system, allowing you to specify the desired build system for compiling the LLVM source files. We recommend using "Unix Makefiles" as the build system for Ubuntu. To build LLVM and clang, execute the following command:

```
cmake -S llvm -B build -G "Unix Makefiles" -DLLVM_ENABLE_PROJECTS="clang"
    -DLLVM_TARGETS_TO_BUILD="X86" -DLLVM_CCACHE_BUILD=ON
cd build
make -j4
```

`cmake` provides multiple options to build the files. One such option is `-DLLVM_ENABLE_PROJECTS="..."`. You can specify additional LLVM subprojects for building by providing a semicolon-separated list, including any of `clang`, `clang-tools-extra`, `lldb`, `lld`, `polly`, or `cross-project-tests`. We suggest building only `clang` for now.

After you have successfully executed `cmake`, you need to enter into the `build` folder and use `make -jn` command to build LLVM and clang. The -j flag used in the make command is used to specify the number of parallel jobs (threads) that make can initiate simultaneously when building a project. For instance, if you run make -j4, it instructs make to start up to 4 concurrent jobs (threads) to build the target(s) specified in the makefile. The number after the -j flag indicates the maximum number of jobs that can run simultaneously. If you omit a number after -j, `make` will execute as many jobs as the number of processor cores available in your system by default. To get more information about LLVM installation, you can visit the following link.

2 Generating & Analyzing LLVM IR (100 pts)

2.1 Generating and Inspecting LLVM IR (15 pts)

You have been given five files: `file1.c`, `file2.c`, `file3.cpp`, `file4.cpp`, and `file5.c`. Each file contains unique code that will generate different LLVM IR.
Objective: You are required to compile these files into LLVM IR using `clang` and `clang++`.
Important Note:

- Use the following command format to compile each file to LLVM IR:

  ```
  ../build/bin/clang -S -emit-llvm file_name.c -o file_name.ll
  ../build/bin/clang++ -S -emit-llvm file_name.cpp -o file_name.ll
      -I/usr/include/c++/(ver_number) -I/usr/include/x86_64-linux-gnu/c++/(ver_number)
  ```

  Replace `ver_number` with the C++ version of your system. We strongly recommend to use `c++-11` for all the assignments.

- You are only required to generate LLVM IR files (`.ll` format). Do not generate files in `.bc` format.
- Do not use pre-built `clang` binaries; ensure that you are compiling from a fresh build of `clang`.

Automate the Process:

1. You have been provided with a partially written `run.sh` script that automates the compilation process. Fill in the script with the correct commands for all five files. Ensure that the script works on your system and that it produces the correct `.ll` files for each input file.
2. If you face issues with the script not generating correct files or errors during compilation, you must debug and fix them yourself. Document the process and reasoning behind any changes you make to the script.

2.2 Analysis of LLVM IR (35 pts)

After generating the `.ll` files for the five provided C/C++ files, complete the following tasks:

2.2.1 LLVM IR Instruction Analysis (25 pts)

Task: Open the generated `.ll` files and manually identify at least five unique LLVM IR instructions (such as `alloca`, `load`, `store`, `icmp`, `ret`, etc.). For each instruction, trace its role back to a specific line or construct in the source code. Explain why the instruction appears in the IR and how it reflects the logic of the source code. While explaining, specify clearly the IR instructions and corresponding source code line number, followed by its role in the source code.

2.2.2 Function Attributes in LLVM IR (10 pts)

In the IR files, you will notice function attributes like `noinline`, `nounwind`, and `uwtable`. Identify any two function attributes that appear before the function definitions in the IR. For each attribute, explain:

- Its purpose and what effect it has on the function or the program.
- Why such attributes may be applied in the context of your provided files.

2.3 `-mem2reg` Pass Exploration and Troubleshooting (20 pts)

Use the opt tool to run the `-mem2reg` pass on the generated IR files and store the result in a new `.ll` file (for each original file).
Task:

1. You must compare the original IR file and the transformed IR file. Identify specific differences between them.
2. Write a detailed explanation of the transformations. Why were certain variables promoted to registers?
3. In cases where no transformation is visible, you must identify the reason your IR is not being transformed. If the `-mem2reg` pass does not transform your IR, you need to troubleshoot and figure out why the pass is not working.
4. After solving any issues, update your `run.sh` script to include the successful execution of the `-mem2reg` pass. Remember that this script should:
   (a) Compile the C/C++ files to LLVM IR.
   (b) Run the `-mem2reg` pass on each IR file.
   (c) Output the transformed `.ll` files.

2.4 Comparative Analysis and Name Mangling (30 pts)

Task:

1. C vs. C++ IR Differences: (5 pts)
   Compare the IR files for a C program and a C++ program from the provided files.
   (a) List three notable differences you observe in the IR for C vs. C++.
   (b) For each difference, explain why it occurs based on the structure or features of C and C++.
2. Name Mangling in C++ (15 pts): Notice that in the IR of `file3.cpp` and its corresponding IR, some function names will look different from those in the source code. This phenomenon is called Name Mangling.
   (a) Explain why this name mangling happens.
   (b) Without providing a precise algorithm, describe the general process of how name mangling works in C++. Think about how functions in C++ might be renamed to handle overloading, scoping, and namespace issues.

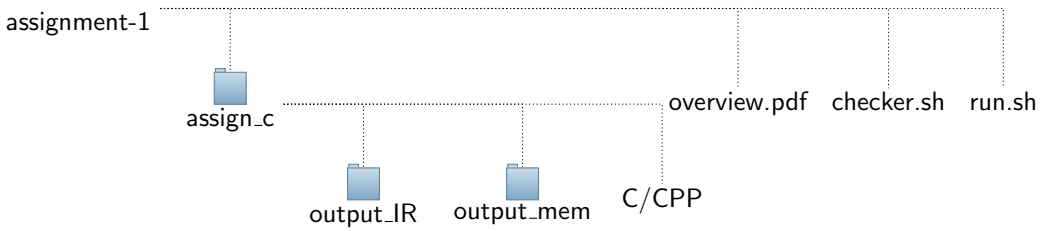

Figure 1: Template directory structure of the deliverable.

3 Submission Guidelines

The structure of the given template directory is mentioned in Figure 1.

There are five C/CPP codes inside the `assign_c` folder, which you must use for the tasks mentioned in Section 2. Generated IR files should be kept inside the `output_IR` folder, and corresponding transformed IR files (after running `-mem2reg`) should be kept inside `output_mem` folder. Your generated IR files should have the same name as the corresponding C/CPP files (e.g., the IR file corresponding to file1.c should be named file1.ll). Similarly, transformed IR files should have the suffix "_mem" after the original C/CPP file name (e.g., the transformed IR file corresponding to file1.c should be named file1_mem.ll).

Here are some DOs and DONTs for the assignment.

DOs

- Use git commit to upload the assignment.
- Run the script in `checker.sh` file and submit the assignment only after receiving an "Accept" output from the script. It checks the naming conventions and folder structure. Note: `checker.sh` cannot validate the correctness of your assignment results.
- Clone the assignment repository inside the llvm-project folder.
- TAs should only run `run.sh` file to generate IR files and transformed IR files. If the files are not automatically generated after running `run.sh`, your assignment may get zero marks.
- Write your **name** and **roll number** clearly in `overview.pdf` file. Write all the answers in the `overview.pdf` file.
- Try to submit within the deadline. There is a late penalty (**20**%) for each day after the deadline.

DONTs

- Do not submit assignment if the `checker.sh` gives "Rejected". Your assignment will not be evaluated if your directory structure or naming conventions do not match.
- Do not change the name of any files or folders.
- Do not use the GPT tool to write the theory answers. There will be a heavy penalty for such behaviour.