Viceroy Summer 2024 Research - LLVM Conversion

Samantha Brewer

This documentation is written based on Ubuntu 22.04.

The [language]-convert folders (found in the same directory as this document) contain projects tested for LLVM, and will be referenced throughout this document.

Each individual project includes a commands.txt, showing the specific commands to convert that project to LLVM. If the LLVM bitcode output is functional, it will show how to run it, otherwise it will show how to compile and run the project normally without bitcode conversion.

When testing a specific language, please follow the installation steps for that language below first.

Note about exporting variables: due to the way Linux works, exporting variables only saves that variable for that specific terminal run, and will reset when the terminal is closed or another is open.

LLVM Background/Context

LLVM was used in this research to generate LLVM Intermediate Representation, and so will be used solely in that context even though it has a lot of other functionality. Specifically, to convert code to IR, the following were used (links to corresponding doc page on LLVM):

- llvm-link: links individual bitcode files together (sparsely documented, not sure if just concatenation or some sort of compiler)
- llvm-as: assembler, to make functional .bc from readable .ll
  ```
  llvm-as INPUT.ll -o OUTPUT.bc
  ```
- llvm-dis: disassembler, to make viewable .ll from functional .bc
  ```
  llvm-dis INPUT.bc -o OUTPUT.ll
  ```
- (llvm-) lli: to run functional .bc LLVM files
  ```
  lli NAME.bc
  ```

Specific command workflows for each project/conversion test is included as a text file in the project's folder.

LLVM by itself only converts files to LLVM IR one-to-one. The linker *should* be able to compile together all of these bitcode files, but for large projects it's not really viable to do manually, especially since these merged bitcode files could not be run properly, such as the output error below for the c-cpp-convert/wllvm-convert/makefile-project:

```
JIT session error: Symbols not found: [ _Z3addii ]
lli: Failed to materialize symbols: { (main, { $..main.o.bc.__inits.0,
__orc_init_func..main.o.bc, main }) }
```

However they seemed to work for the chatGPT model generation, so could be looked into more.

C/C++ Conversion

- Individual Conversion Possible: Yes
- Multi-file Conversion Possible: Yes (WLLVM)
- Automated: Yes (See C2LLVM.sh)

- Installation

C/C++ LLVM conversion uses LLVM version 14 as that is the version that comes with the basic apt install command, compared to version 18 for Fortran below.

All required prerequisites can be automatically installed by running the c2llvm-setup.sh using bash.

```
sudo apt-get install bash
sudo bash c2llvm-setup.sh
```

The specific prerequisites installed with the shell script are:

- `sudo apt install g++-12`
  - Ran into a compiling issue on the makefile project for Ubuntu 22.04 (C++ program was missing iostream), may not be necessary
- `sudo apt install llvm`
  - LLVM, necessary, installs version 14 including clang (required for WLLVM)
- `sudo apt install cmake`
  - Needed to compile CMake projects
- `sudo apt install python3-pip`
  - Needed to install WLLVM
- `sudo pip install wllvm`
  - WLLVM for multi-file projects

- Single File LLVM Conversion

  Manual Conversion Commands:

  - [to make functional .bc file from .c] `clang -O3 -emit-llvm NAME.c -c -o NAME.bc`
  - [to make viewable .ll file from .c] `clang -O3 -emit-llvm NAME.c -S -o NAME.ll`
  - [to make functional .bc from viewable .ll] `llvm-as INPUT.ll -o OUTPUT.bc`
  - [to make viewable .ll from functional .bc] `llvm-dis INPUT.bc -o OUTPUT.ll`

  To run functional .bc LLVM files:

  ```
  lli NAME.bc
  ```

  just as you would do

  ```
  clang NAME.c -o NAME
  ./NAME
  ```

  to run C code.

  In `/c-cpp-convert/singlefile-convert/helloworld/` is a test hello world file for LLVM conversion. Check corresponding commands.txt for how to convert or run code. Bitcode **IS** functional.

  In `/c-cpp-convert/singlefile-convert/malloc/` is a test malloc file for LLVM conversion. Check corresponding commands.txt for how to convert or run code. Bitcode **IS** functional.

- WLLVM

  Whole Program LLVM (https://github.com/travitch/whole-program-llvm) is a wrapper script that handles combining all the C/C++ files in a given project into one single bitcode file.

  For these test files, the user is expected to use the C2LLVM shell script. If the user is curious what commands are used for conversion, they can look in c2llvm.sh. Note that the shell command does `export LLVM_COMPILER=clang`. If manually running the WLLVM commands instead of using the shell, they need to run that export command every time they open a new terminal (like how sudo requires login).

  The 'simple' projects to test with WLLVM compilation were taken from: https://github.com/secure-software-engineering/phasar/wiki/Whole-Program-Analysis-(using-WLLVM)

Specifically, the files were taken from the PhASAR link on the page (It links to https://phasar.org/download/, the example projects are at the bottom of the page) and the C2LLVM shell script was written based on the instructions.

Simple Makefile Project

In **`/c-cpp-convert/wllvm-convert/makefile-project/`** is a test makefile project for WLLVM conversion. Check corresponding commands.txt for how to convert or run code. Bitcode **IS NOT** functional.

Simple CMake Project

In **`/c-cpp-convert/wllvm-convert/cmake-project/`** is a test CMake project for WLLVM conversion. Check corresponding commands.txt for how to convert or run code. Bitcode **IS** functional.

The HW1 Use Case is the use case the team used for the full pipeline demo and is an old homework assignment of mine (to demonstrate real life use rather than specifically tailored projects).

HW1 Use Case

In **`/c-cpp-convert/wllvm-convert/hw1-usecase/HW1/`** is the CMake project real-life use case for WLLVM conversion. Check corresponding commands.txt for how to convert or run code. Bitcode **IS** functional.

Fortran Conversion

- Individual Conversion Possible: Yes
- Multi-file Conversion Possible: Yes (Flang and LLVM-link)
- Automated: No (Time Constraints)

- Installation

Note that Flang, the LLVM Fortran compiler, is one of several “Flang” compilers. Double check you’re using the LLVM one (https://flang.llvm.org/docs/). Version 18 of LLVM ships with it by default. Flang is also “not ready for production usage” per the earlier link, so not confident how much can be done with it currently.

For Fortran, LLVM Version 18 had to be installed for the Flang compiler. I used https://apt.llvm.org/ to download version 18 for Debian/Ubuntu. The below command, given on the site, automatically installs the *latest* stable version of LLVM, which may be a newer version than what was used for this Fortran conversion documentation.

```
sudo bash -c "$(wget -O - https://apt.llvm.org/llvm.sh)"
```

The latest version uses different commands than normal LLVM. For example, to check installation of LLVM installed via sudo apt install for C/C++ (version 14) and then via the above bash command (version 18), the commands and output are as below:

`llvm-config --version`	`14.0.0`
`llvm-config-18 --version`	`18.1.8`

So just make sure you are using the updated LLVM commands (if you're not sure the specific name, just do the normal command then hit tab until you see some commands, example shown below). If the latest version is newer than version 18, then the '18' in the command is likely replaced with the newer version number.

- ## Single File LLVM Conversion

  In **`/fortran-convert/helloworld/`** is a test hello world file for LLVM conversion. Check corresponding commands.txt for how to convert or run code. Bitcode **IS NOT** functional.

- ## Multi File LLVM Conversion

  In **`/fortran-convert/multifile/`** is a two-file Fortran project for LLVM conversion. Check corresponding commands.txt for how to convert or run code. The original Fortran code was found here: https://fortran-lang.org/learn/building_programs/linking_pieces/. Bitcode **IS NOT** functional.

- ## Go WLLVM

  I tried to look into a Go version of WLLVM (https://github.com/SRI-CSL/gllvm) for its support for Flang, but it was even less documented than WLLVM itself. Since Fortran support was only lightly tested to demonstrate extensibility and WLLVM worked already for C/C++, GLLVM was dropped. However, below are all of the commands I did to install it.

1. Follow instructions for installing Go here: https://go.dev/doc/install
2. `export PATH=$PATH:/usr/local/go/bin` (Exported variable! Every time you interact with GLLVM in a terminal, you need to run this command first, like sudo requiring login)
3. `go install github.com/SRI-CSL/gllvm/cmd/...@latest` (Assuming installed LLVM version that includes clang and flang)

Rust Conversion

Due to time constraints and unfamiliarity with the language, I wasn't able to really get to the testing stage for Rust. However, below are good links to specific things I was researching (not including the many stack overflow pages I visited).

- Getting Rust to output LLVM IR: https://rustc-dev-guide.rust-lang.org/backend/debugging.html#get-your-hands-on-raw-llvm-input
- Manually combining Rust files into LLVM IR, including missing symbols: https://medium.com/@squanderingtime/manually-linking-rust-binaries-to-support-out-of-tree-llvm-passes-8776b1d037a4