LLVM-IR-Dataset-Utils - Scalable Tooling for IR Datasets

Aiden Grossman
University of California, Davis
Work performed while at LLNL

April 28, 2024

Outline

Collaborators

Ludger Paehler

Konstantinos Parasyris

Tal Ben-Nun

Jacob Hegna

William Moses

Jose Monsalve-Diaz

Mircea Trofin

Johannes Doerfert

INTRODUCTION

1. What problem are we actually trying to solve?
   1.1 As ML models become bigger, more data is needed to train them.
   1.2 Many compiler analyses are bespoke and not evaluated against multiple languages.
   1.3 Other efforts, like translation validation, can also benefit from more data.
2. We believe that having *scalable tooling* for creating and analyzing large IR datasets helps solve these problems.

What is ComPile?

1. ComPile is a ready made IR dataset made using this tooling.
2. Contains only appropriately licensed projects.
3. Approximately 600GB of Bitcode (2.8TB of textual IR).
4. Available for download on HuggingFace (`https://huggingface.co/datasets/llvm-ml/ComPile`).

Dataset Distribution

Programing Language	Bitcode (GB)	Textual IR (GB)
C	2	10
C++	29	103
Julia	164	1088
Rust	400	1524
Swift	7	36

General Process

1. Run the normal build process with additional flags to emit/embed bitcode.
    1.1 Pass flags to extract bitcode as early as possible after the frontend.
2. Extract the bitcode into a corpus.
3. Collect multiple corpora into a large dataset.

IR Extraction Tooling - mlgo-utils

1. Upstreamed in the monorepo `llvm/utils/mlgo-utils`
2. Extracts IR in a variety of circumstances:
   2.1 Structured compilation database (eg `compile_commands.json`) with embedded bitcode.
   2.2 "Loose" Bitcode files in the build directory.
   2.3 Several other cases not used here (eg ThinLTO).

Extracting C/C++ IR

1. Build with clang.
2. Pass the flag `-Xclang -fembed-bitcode=all` to enable bitcode embedding.
3. Extract the IR from the object files in the build directory.
   3.1 When possible, we use `compile_commands.json`.
   3.2 Otherwise, `mlgo-utils` finds all object files in the build directoy.
4. We also extract compilation command lines and include them in the corpus.

Extracting Julia IR

1. Julia's compilation model is very different from a standard AOT compiler.
   1.1 Code is JITed as needed and the exact types aren't known until runtime.
2. Precompiling packages with common types has become common with `PrecompileTools.jl`
3. We use a custom patch to make Julia emit unoptimized bitcode, using the precompilation tooling and running unit tests to force the lowering of functions.
4. The IR from Julia has some caveats:
   4.1 The IR has not been legalized at this point.
   4.2 Running IPO passes creates correctness issues.

Extracting Rust IR

1. For building Rust code, we use `cargo` an build individual targets with `--emit=llvm-bc -C no-prepopulate-passes` in debug mode.
2. After building multiple targets, we copy the bitcode files over into a corpus using `mlgo-utils`.
3. We pass `no-prepopulate-passes` to help ensure that we are getting bitcode before any passes are run over the LLVM-IR.

EXTRACTING SWIFT IR

1. We use the standard swift compiler (on Linux) and pass the following command line flags:
   1.1 `-c release` to build in release mode.
   1.2 `-Xswiftc -embed-bitcode` to enable bitcode embedding.
   1.3 `--emit-swift-module-separately` as it is required to embed bitcode.
   1.4 `-Xswiftc -Onone` to prevent any premature optimizations.

What is a builder?

With `llvm-ir-dataset-utils`, we have the concept of a builder that is responsible for a couple things:

1. Installing dependencies required to build a package when necessary.
2. Building as many targets in the package as possible with the appropriate flags.
3. Extracting the resulting bitcode into a corpus in a specified directory.
4. Logging build information for possible later analysis.

Each builder is specific to a *package ecosystem* or build system rather than a language.

General Scaling Principles

1. We use `ray` to distribute build jobs across a single node and across multiple nodes when available.
2. The tooling is designed to split work across multiple nodes as the cost of building this many packages is quite high.
3. We handle failures gracefully, but log all of them. There are a lot of different failure cases.

SCALING C/C++

1. We use the open-source HPC package manager Spack as our source. This gives us 5000 C/C++ packages.
2. Many packages fail to build mostly due to clang/gcc compatibility issues, but almost all core dependencies build.
3. We're interested in expanding to other sources of vetted C/C++ in the future.

Scaling Rust

1. We have a builder for arbitrary cargo packages.
2. For building ComPile, we run this over the entirety of `crates.io`.
3. We observe many individual targets failing at this level for various reasons, but still collect a large number of targets.

Scaling Julia

1. We have a builder for arbitrary Julia packages.
2. For building ComPile, we run this over the entirety of `https://juliahub.com/ui/Packages`.
3. Building Julia packages takes a large amount of time, mostly due to the need to run unit tests to force lowering of functions.

Scaling Swift

1. We use `SwiftPM` as a package manager/build system and can build arbitrary swift packages.
2. For building ComPile, we run this over the entirety of `https://swiftpackageindex.com/`.
3. We run into a large number of build failures on Linux due to dependency availability issues, like SwiftUI.

Bespoke Builders

1. We have bespoke builders for specific projects, supporting the following build systems:
   1.1 CMake
   1.2 Autotools
   1.3 Arbitrary shell commands
2. Useful for including large single projects in large datasets or including internal code.
3. Individual projects are specified using JSON.

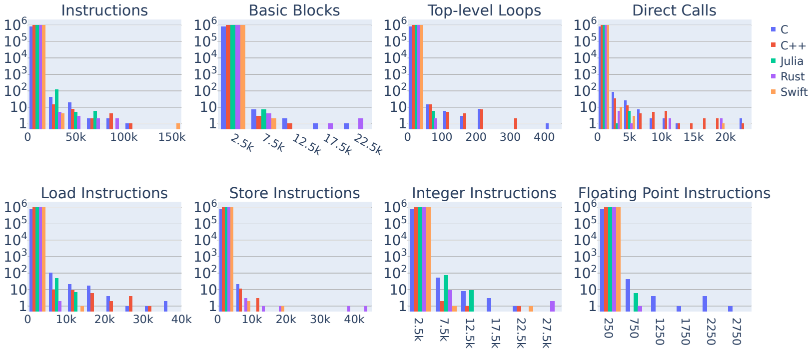
Analysis Tooling

1. Designed to scale to large clusters.
2. Works on dataset built from source, HF version is future work.
3. Can do a variety of different function and module-specific analyses.

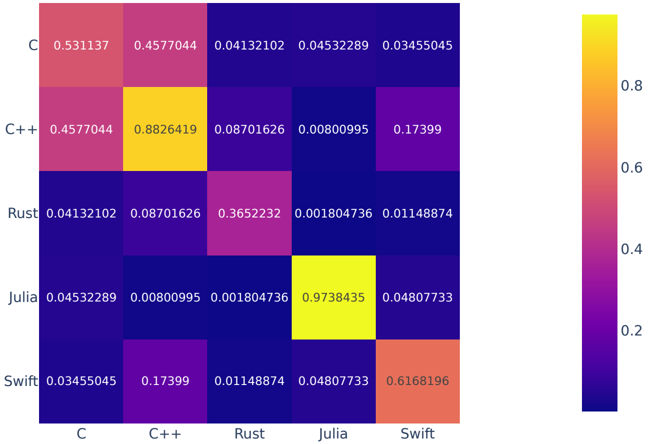
Example Analyses

1. Pass mutation frequency by language.
2. Opcode distribution by language.
3. Function property distribution by language.
4. Function duplication within and between languages.

Example Analyses - Function Properties



Example Analyses - Function Duplication



LICENSE CONSIDERATIONS

1. License constraints are important for a variety of use cases:
    1.1 Code in the distributed dataset needs to be licensed for that purpose.
    1.2 Different companies have different licensing constraints (eg prohibition of AGPL).
2. To deal with license constraints, we have tooling for ltering by license.

DETERMINING LICENSES

1. Each corpus manifest that describes a speci c package is expected to have a license eld.
2. All license les from the top level of the source directory are copied over to the corpus.
3. A project is considered to be license compliant if it is licensed under the set of allowed licenses and has a license le matching that license ID.

HOW CAN I RUN THIS?

1. Code is available in a PR: https://github.com/llvm/llvm-project/pull/72320
2. More information on the dataset: https://llvm-ml.github.io/ComPile/
3. Documentation: https://llvm-ir-dataset-utils.vercel.app/
4. Incubator project RFC: https://discourse.llvm.org/t/rfc-incubator-project-for-llvm-ir-dataset-utils/74940 If you're interested in using this or the dataset itself, please express your interest on the RFC!

FUTURE DIRECTIONS

1. Improve code quality!
   1.1 Add better testing infrastructure.
   1.2 Some refactoring/other misc code quality improvements to help productionize it.
2. Add source code mappings. Prototype work for this has already completed.
3. Add more sources, particularly C/C++ from linux distros.
4. Capturing of link-time dependencies.
5. Additional languages (eg Fortran).

Work Building On/Using Large IR Datasets

1. Compile time analyses
2. Cost modelling
3. ML for IR-related tasks (code size prediction)
4. Input generation to enable executing the code for performance and runtime analyses.

Acknowledgements

We would like to thank Valentin Churavy for his assistance in understanding the Julia compiler. Additionally, we would like to thank Todd Gamblin, Alec Scott, Harmen Stoppels, and Massimiliano Culpo for their assisstance with Spack. Finally, we would like to thank Nikita Popov, Arthur Eubanks, and other LLVM contributors who helped review patches that made this work possible.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-PRES-862525).

Questions?

Answers! (Hopefully)