ComPile: A Large IR Dataset From Production Sources

Aiden Grossman

In Collaboration with Ludger Paehler, Konstantinos Parasyris, Tal Ben-Nun, Jacob Hegna, William Moses, Jose M Monsalve Diaz, Mircea Trofin, and Johannes Doerfert
Motivation

● Prior existing large datasets aimed at ML for code.
  ○ Not compilable
  ○ Distribution is typically whatever can be found on Github
    ■ Some exceptions like competition sourced datasets, still suffer from distribution problems
● Existing prior datasets for working with compilers
  ○ Mostly small, can change the semantics of the code.
ComPile

- A large 2+TB dataset of (textual) IR from production sources.
- Extensible tooling to allow for creation of large datasets including internal code.
- Trying to sidestep the dataset distribution problem by just including code that people actually use.
- Currently useful for large scale analyses, pretraining large models/code-size tasks.
- Looking to extend to performance related tasks in the future.
Sources

All this talk about production sources - what actually are they?
Rust Crates

- Building approximately 100,000 Rust Crates from the Rust Crates Index and extracting all generated bitcode.
- Not filtering out any crates, attempting to build all publicly available crates.
- Commonly used throughout the Rust ecosystem.
- All packages use a consistent build system, cargo.
Swift Packages

- Attempting to build 6000+ publicly available Swift Packages.
- Used commonly throughout the Apple/Swift ecosystem, especially in mobile development.
- Lots of build failures due to our infrastructure running on Linux where several key dependencies aren’t present.
- Looking to improve representation of Swift.
Spack (HPC-Centric Package Manager)

- Attempt to build 5000+ applications packaged for deployment on HPC systems.
- Constitutes the majority of C/C++ within ComPile.
- Mostly aimed at HPC, not necessarily representative of datacenter applications, looking into ways to improve this situation.
Julia

- Attempt to build 8000+ Julia Packages.
- Mostly focused on scientific computing/HPC applications.
- Packages produce a lot of bitcode, but a lot of it is subtly different than other languages.
  - Lots of subtle breakages with things like IPO due to how Julia represents calling conventions.
Individual Projects

- We also include several large single projects to further demonstrate the abilities of our tooling and bolster ComPile:
  - Linux kernel
  - Chromium
  - Firefox
- Compose ~25GB of Bitcode on their own, mostly C/C++.
- The current license constraints that we have mean that we don’t distribute Firefox/Linux in the public version of the dataset.
IR Collection Methodology

- Take IR directly after the frontend so it can be used for *anything* downstream.
- Different pipeline per package ecosystem.
- Collecting data using tooling recently upstreaming into llvm. `llvm/utils/mlgo-utils` in the monorepo.
- Stored as BC to keep it forwards-compatible.
Dataset Postprocessing

Not all data is good or useful
Deduplication

- Take advantage of the highly structure form of LLVM-IR.
- Use LLVM’s `StructuralHash` which hashes based on the semantics of the IR.
- Slightly lossy, enables near matching. Working on making this more configurable.

<table>
<thead>
<tr>
<th>Language</th>
<th>C</th>
<th>C++</th>
<th>Julia</th>
<th>Rust</th>
<th>Swift</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (Bitcoin)</td>
<td>13.18 GB</td>
<td>81.08 GB</td>
<td>197.06 GB</td>
<td>482.13 GB</td>
<td>4.52 GB</td>
<td>777.97 GB</td>
</tr>
<tr>
<td>Size (Text)</td>
<td>60.88 GB</td>
<td>334.28 GB</td>
<td>1292.12 GB</td>
<td>1868.23 GB</td>
<td>21.98 GB</td>
<td>3577.49 GB</td>
</tr>
<tr>
<td>Dedup. Size (Bitcoin)</td>
<td>7.87 GB</td>
<td>66.55 GB</td>
<td>129.97 GB</td>
<td>310.05 GB</td>
<td>3.94 GB</td>
<td>518.38 GB</td>
</tr>
<tr>
<td>Dedup. Size (Text)</td>
<td>33.82 GB</td>
<td>265.65 GB</td>
<td>855.89 GB</td>
<td>1221.11 GB</td>
<td>18.87 GB</td>
<td>2395.34 GB</td>
</tr>
</tbody>
</table>
License Tagging

- Can only distribute IR where we know the code is permissively licensed
- Use a combination of package ecosystem dependent techniques to derive license information.
- All license and provenance information is included for each module.
- Currently include MIT, Apache-2.0, BSD-3-Clause

<table>
<thead>
<tr>
<th>Language</th>
<th>Permissively Licensed</th>
<th>Permissively Licensed with license file</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rust</td>
<td>79.9%</td>
<td>67.2%</td>
</tr>
<tr>
<td>Julia</td>
<td>88.6%</td>
<td>88.6%</td>
</tr>
<tr>
<td>Spack</td>
<td>57.0%</td>
<td>38.6%</td>
</tr>
<tr>
<td>Swift</td>
<td>94.3%</td>
<td>94.3%</td>
</tr>
</tbody>
</table>
Use Cases

All this data - but why?
IR centric LLMs

- Very large corpus of IR suitable for training billion parameter models.
- Promising initial results on using ComPile to train smaller LLMs on tasks that demonstrate an “understanding” of IR like post-optimization code-size prediction.

**Model Performance**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Tokenization</th>
<th>Task</th>
<th>Percent Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>450M</td>
<td>GPT</td>
<td>Size</td>
<td>5.0%</td>
</tr>
<tr>
<td>250M</td>
<td>LLVM</td>
<td>Size</td>
<td>6.3%</td>
</tr>
<tr>
<td>450M</td>
<td>GPT</td>
<td>Size O3</td>
<td>6.3%</td>
</tr>
<tr>
<td>250M</td>
<td>LLVM</td>
<td>Size O3</td>
<td>9.7%</td>
</tr>
</tbody>
</table>

Table 2: Performance of best performing models (that have been trained so far) in each category.
Compiler Testing/Analysis

- Enables the validation of the myriad of assumptions that compiler engineers make.
- Enable large scale testing of correctness using techniques like translation validation with Alive2 and mutation-based fuzzing.
- Enables the exploration of the effectiveness of new optimization techniques.
Further Applications

- Large dataset to train learned cost models on.
- Bayesian optimization of existing tunable heuristics within the compiler.
- Head room analysis
- Benchmarking new compiler techniques
Future Directions

There is always a lot of additional work to do…
Short term directions

● Source code to IR module mappings.
  ○ Currently planning this for C/C++

● Additional Builders
  ○ We want to pull data from additional sources, in particular traditional Linux packages, and Fortran code.

● Dataset balance
  ○ The dataset is currently skewed towards Rust/Julia. Working on quantifying the effect this skew has in addition to correcting it by pulling in data from more sources.
Future Directions for Downstream Applications

● Further investigation on our end on domain-specific tokenization for LLVM IR.
  ○ Mostly focused on laying the groundwork for future work on performance modeling.
● ML based basic block level cost modeling
● Very excited to see how the community is able to utilize this resource!
Tooling/Dataset Availability

Hard to build off our work if it isn’t public…
ComPile

- Open split of the dataset available at [https://huggingface.co/datasets/llvm-ml/ComPile](https://huggingface.co/datasets/llvm-ml/ComPile)
- Smaller than the closed version due to licensing constraints
Using ComPile

- Quick experiments!
- The dataset is reasonably big, downloading takes a long time.
- Each row contains a raw bitcode module and associated metadata

```python
>>> from datasets import load_dataset
>>> ds = load_dataset('llvm-ml/ComPile', split='train', streaming=True)
Resolving data files: 100% 774/774 [00:00<00:00, 186.55it/s]
```
llvm-ir-dataset-utils

- Working on open-sourcing the tooling under the LLVM umbrella.
- Useful for constructing the dataset as well as performing various analyses.
- If you’re interested in using this (or the dataset that depends on this tooling), please leave a comment on the RFC!

As part of our recent effort to build a large dataset of LLVM-IR for applications in machine learning, compiler testing, and more ([2309.15432] ComPile: A Large IR Dataset from Production Sources), we have created tooling to build large IR datasets from package indices and to process the resulting information. We’d be interested in moving this under the LLVM umbrella as an incubator project. Note that this effort is complementary to [RFC: Upstreaming elements of the MLGO tooling] as we utilize this tooling extensively. We believe this provides a couple key advantages to the community:

- It makes it easier for other parties to contribute to the effort, especially those that restrict/require approval for general open source contributions, but might have an exception for contributing to LLVM.
- Puts the tooling in an easily discoverable, centralized place where it complements the rest of LLVM well to aid in the previously mentioned efforts.
ComPile: A Large IR Dataset from Production Sources

Aiden Greenwood 🌟, Lodger Peschler 🌟, Konstantinos Papanikolaou 🌟, Tali Ben-Nun 🌟, Jacob Hugon 🌟, William Musco 🌟, Jose M Monashef Dinz 🌟, Mirco Trifin ⬤, Johannes Doerfler 🌟

Abstract

Code is increasingly becoming a core data modality of modern machine learning research impacting not only the way we write code with conversational agents like OpenAI’s Chat-GPT, Google’s Bard, or Anthropic’s Claude, but the way we translate code from one language into another, but also the compiler infrastructure underlying the language. While modeling approaches may vary and representations differ, the targeted tasks often remain the same within the individual classes of models. Relying solely on the ability of modern models to extract information from unstructured code does not take advantage of 50 years of programming language and compiler development by not utilizing the structure inherent to programs in the data collection. This detriment to the performance of models working on a tokenized representation of input code and precludes the use of those models in the compiler itself. To work towards the first intermediate representation (IR) based model, we fully utilize the LLVM compiler infrastructure, shared by a number of languages, to generate a 1629 token dataset of LLVM IR. We generated this dataset from programming languages built on the shared LLVM infrastructure, including Rust, Swift, Julia, and C/C++, by hooking into LLVM code generation either through the language’s package manager or the compiler directly to extract the dataset of intermediate representations from production-grade programs. Statistical analysis proves the utility of our dataset not only for large language model training, but also for the introspection into the code generation process itself with the dataset showing great promise for machine-learned compiler components.

1Work performed while at LLVM
2Previously tokenized subset of the dataset available under: loggingface/03/tesseract/UCDavis/ComPile
Questions? Answers (Hopefully)