ComPile: A Large IR Dataset from Production Sources

Aiden Grossman[1]* **Ludger Paehler**[2] **Konstantinos Parasyris**[3] **Tal Ben-Nun**[3]
Jacob Hegna[4] **William Moses**[5] **Jose M Monsalve Diaz**[6] **Mircea Trofin**[7]
Johannes Doerfert[3]
[1]UC Davis [2]Technical University of Munich [3]Lawrence Livermore National Laboratory
[4]University of Minnesota [5]University of Illinois Urbana Champaign
[6] Argonne National Laboratory [7] Google, Inc.
`amgrossman@ucdavis.edu` `ludger.paehler@tum.de`
`{parasyris1,talbn,jdoerfert}@llnl.gov` `jacobhegna@gmail.com`
`wsmoses@illinois.edu` `jmonsalvediaz@anl.gov` `mtrofin@google.com`

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 ChatGPT, Google's Bard, or Anthropic's Claude, 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 70 years of programming language and compiler development by not utilizing the structure inherent to programs in the data collection. This detracts from the performance of models working over a tokenized representation of input code and precludes the use of these models in the compiler itself. To work towards better intermediate representation (IR) based models, we fully utilize the LLVM compiler infrastructure, shared by a number of languages, to generate a 182B token dataset of LLVM IR with a 144B token public version [2]. 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. Our dataset shows great promise for large language model training, and machine-learned compiler components.

1 Introduction

In several pieces of previous work (8; 14), the transformative potential of machine learning was harnessed, machine-learned heuristic replacements developed, and in some cases (21) the *heuristics were upstreamed to the main LLVM codebase*, improving all code run through LLVM when the ML heuristics are enabled. Orthogonal to the replacement of heuristics with machine learning, a large number of people have explored the ordering of compiler passes (4; 10). While the learning of pass orderings was initially held back by the lack of easy-to-access, high-performance reinforcement learning environments to validate new reinforcement learning strategies, this has by now been addressed with the introduction of CompilerGym (4). In contrast, the learning of entirely new heuristics, optimization passes, and other compiler components with large language models (22; 3) to realize the transformative potential of this model class is held back partially by the lack of large

---

[*] Corresponding author

[2] `https://huggingface.co/datasets/llvm-ml/ComPile`

Machine Learning for Systems Workshop at 37th NeurIPS Conference, 2023, New Orleans, LA, USA.

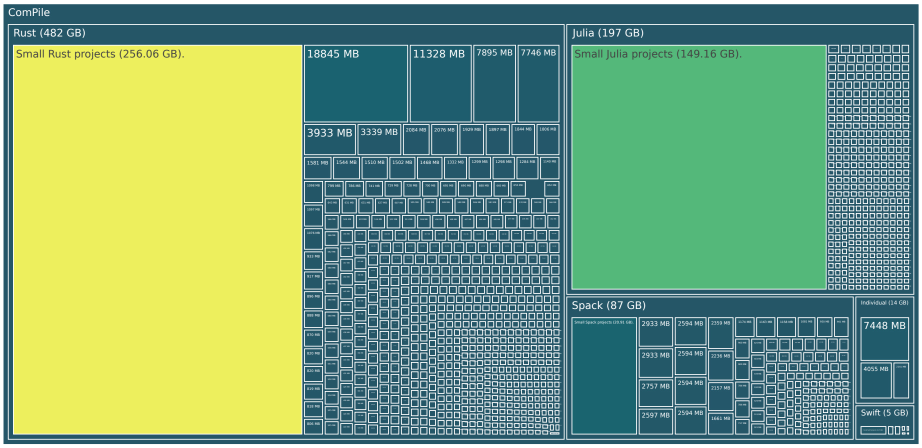

Figure 1: Size distribution of LLVM intermediate representation (IR) bitcode within ComPile before de-duplication within and among languages. Projects that we considered small and pooled had less than 100MB of bitcode.

datasets of high-quality code to train such models properly. Models are only trained on smaller datasets, such as Anghabench (5), Exebench (2), and HPCORPUS (11), or sometimes rely on synthetic benchmarks. Small datasets ultimately lead to smaller, worse-performing models (9).

1.1 Contributions

Focussing on the paradigm of taking a pre-trained basic building block, a "foundation model", we pose the question *"What does a modern, large code training dataset for compilers actually have to look like?"* and construct a high-quality dataset of a similar scale to existing LLM datasets solely at the level of LLVM-IR. Within this context, we associate quality with the *usage* of code, with code being used more often being of higher quality for our purposes. Correctly being able to reason about very widespread code in production systems is incredibly important for compiler work. In the short term, we believe our dataset will enable the training of larger language models for compilers useful for an ever broader array of downstream tasks after fine-tuning, and in the long-term, enable use-cases such as direct performance prediction to obtain a reliable runtime estimate without ever running a single line of code. To these goals, our work makes the following contributions:

- The introduction of a 2.4TB dataset (closed), respectively 1.9TB dataset (public) of textual LLVM-IR from Rust, Julia, Swift, and C/C++ with 182B, respectively 144B [3] tokens at a vocabulary size of 10k. See table 1.1 for the number of tokens at varying vocabulary sizes.
- Open-sourcing of our workflow and compiler tooling to construct massive-scale IR datasets.

Vocabulary Size	10,000	50,000	100,000	200,000
ComPile (closed)	182 B	119 B	102 B	87 B
ComPile (public)	144 B	94 B	81 B	69 B

2 Background

Building upon package ecosystems as sources of intermediate representation is ideal due to the large amount of packaged code and the abstraction over the build systems of individual projects. In addition, package ecosystems act as a filter. Only code that gets used in production systems will get packaged. The build system abstraction is due to a common compilation wrapper over compilation recipes with their exact specification of dependencies. Modifying these build processes allows us to take advantage of this existing infrastructure. In this work, we choose to specifically focus on

---

[3] Values linearly interpolated from raw size counts.

Language	C	C++	Julia	Rust	Swift	Total
Size (Bitcode)	13 GB	81 GB	197 GB	482 GB	5 GB	778 GB
Size (Text)	61 GB	334 GB	1292 GB	1868 GB	22 GB	3577 GB
Dedup. Size (Bitcode)	8 GB	67 GB	130 GB	310 GB	4 GB	518 GB
Dedup. Size (Text)	34 GB	266 GB	856 GB	1221 GB	19 GB	2395 GB

Table 1: Amount of IR contained within ComPile in textual and bitcode form before and after deduplication.

utilizing package managers that explicitly allow setting compiler flags, such as the from-source package manager Spack (6) that is focused on high-performance computing (HPC).

In addition to utilizing package managers, we also take advantage of several aspects of the LLVM compilation infrastructure (16), particularly the Clang C/C++ frontend and LLVM-IR, the intermediate representation LLVM uses. The full process of compilation, such as the one performed by Clang with LLVM during the compilation of C/C++, is composed of three main stages: the frontend, the middle-end, and the backend. A compiler frontend has the job of taking a piece of source code, typically a single source file, sometimes called a translation unit, and generating a *module* of intermediate representation that can then be processed by a compiler middle-end, such as LLVM. A module typically contains multiple functions, referenced globals, and relevant metadata. Compiler intermediate representations, or IRs, are designed to sit between the source programming language and the compiler's output, assembly. They are typically designed to be source-language and target-agnostic. Within LLVM, the compiler middle-end operates over the IR produced by the frontend through a series of grouped operations called passes. A *pass* is designed to perform a specific task, such as removing dead code, simplifying the control flow graph, or combining instructions. After optimization, the compiler backend takes over, performing the necessary tasks to transform the (mostly) target-agnostic IR into target-specific machine code that can be executed on the target machine. The backend typically performs tasks such as instruction selection, instruction scheduling, and register allocation. We compose our dataset, *ComPile*, of LLVM-IR, as it gives a common framework across programming languages and target platforms. These properties and more make LLVM-IR a great modality for a compiler-centric dataset useful for compiler tasks such as program analysis, optimization, and code generation.

3 Dataset Construction

To construct the IR dataset, we use a set of curated sources from five different languages. focusing on code used in production systems. We include the majority of Spack (6), the Rust Crates Index, the Julia Package Index, the Swift Package Index, and several large single projects. Individual project sources are defined in `.json` files. While most projects are hosted in repositories on GitHub, we also added sources consisting of archived compressed source codes such as tarball files. The builders then ingest the information from the project on its build system, either through the manifest information, which contains the information on the building mechanism and commands, or through an ecosystem specific manifest processed by a script that is then processed into a complete package manifest. Next in the workflow is the LLVM-IR extraction. Extracting IR depends on the way the IR is presented in the source. A manifest that contains a list of LLVM bitcode modules extracted from the project is then created. Leaning into the shared LLVM compiler infrastructure, we are able to take advantage of existing LLVM tools and LLVM passes to obtain information about the LLVM-IR modules. After building, IR extraction, and deduplication, the dataset is then ready for downstream usage in analysis or training capacities. [4]

The aim of our IR extraction approach is to extract IR immediately after the frontend, before any LLVM optimization passes have run. To extract the bitcode into a structured corpus, we take advantage of the ml-compiler-opt tooling from MLGO (21) as it allows for the extraction of IR in a variety of cases. During IR extraction, we also collect some additional data, such as debug information, as it is represented in the IR. We specifically collect bitcode rather than textual IR as LLVM supports reading bitcode produced by older versions of LLVM but has no such support for textual IR, which is also easily produced by running `llvm-dis` over the collected corpus.

---

[4] Scripts and builders to reproduce the entire dataset are available under the `llvm-ir-dataset-utils` subdirectory under https://zenodo.org/doi/10.5281/zenodo.10155760

Name of Dataset	Tokens	Size	Languages
The Stack (13)	-	2.9 TB	358 Languages
ComPile (closed)	182 B	2.4 TB	Rust, Swift, Julia, C/C++
ComPile (public)	144 B	1.9 TB	Rust, Swift, Julia, C/C++
Code Llama (20)	197 B	859 GB	-
TransCoder (15)	163 B	744 GB	C++, Java, Python
AlphaCode (17)	-	715.1 GB	12 Languages
LLM for Compiler Opt. (3)	373 M	1 GB	C/C++

Table 2: Breakdown of Related Datasets.

Training dataset deduplication can be important for the performance of several key model characteristics. (1; 12). To this end, We deduplicate the entire dataset presented in this paper at the module level by computing a combined hash of all global variables and functions, deduplicating based on a hashing implementation that only captures semantic details of the IR. We chose to deduplicate at the module level as this ensures the majority of the duplicate code is removed from the dataset while leaving all significant context within each module for performing module-level tasks.

Please see appendix A4 for the exact details of our approach to the filtering of the closer version of the dataset, to arrive at the public version.

4 Related Work

Most pretraining datasets for large language models (17; 13; 18) contain large swaths of code, scraping source code from hosting services like GitHub, and GitLab without taking the quality of the included code into account. Datasets of this type also do not guarantee that any of the code is compilable, and often contain auxiliary files such as documentation in Markdown. Complementary to these large pretraining-scale datasets, there exist a number of smaller, more focused datasets aimed at the fine-tuning of already pretrained large language models (23; 17; 19). These datasets are primarily collected through data extraction from coding competitions (17; 19), or the scraping of curated websites (23). This guarantees a higher level of quality in regards to buildability and structure for the included code, hence making them more optimal for fine-tuning. However, the data collection methodology implicitly introduces a lack of variety in the datasets, reducing model performance (7). For example, coding competititon datasets might include a couple thousand coding exercises which contain a great many solutions to the same exercises, but yet they are only solving the very same set of coding problems.

Additionally, there exist a number of domain-specific datasets (11; 2; 5). Often beginning with the web-scraping of large amounts of code, these approaches modify the resulting code in a number of ways. Examples include the modification of arbitrary source files to make them compilable (5) or executable (2). ComPile, while being able to fulfill similar dataset demands, offers a number of key advantages. The code in our dataset, by means of our dataset construction methodology, consists only of compilable code, using the same compilation toolchain as used for production deployments without changing semantics. Collecting IR before optimization allows for IR at any stage of the compilation pipeline to be easily generated. This allows ComPile to go significantly beyond the capabilities of previous compiler-targeted datasets.

5 Conclusion

In this work, we presented ComPile, a novel dataset of LLVM-IR collected from a number of package ecosystems consisting of large production-grade codebases. It is significantly larger than previous finetuning-focussed, and compiler-focussed code datasets, albeit smaller than large language model-focussed code pretraining datasets. ComPile's increased size in combination with its quality-focused construction methodology not only enables the systematic evaluation of previous work, but opens up entirely new avenues of research for IR-centric machine learning, and most specifically machine-learned compiler componentry for which the scale of this dataset paves the way to an entirely new generation of machine learning models for compilers.

6 Acknowledgements

This work was in parts prepared by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-855448).

We would like to thank Valentin Churavy, Todd Gamblin, Alec Scott, Harmen Stoppels, Massimiliano Culpo, Nikita Popov, and Arthur Eubanks for their assistance with understanding the relevant language-specific optimization pipelines and assistance with getting upstreamed patches through code review.

References

[1] ALLAMANIS, M. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (2019), pp. 143–153.

[2] ARMENGOL-ESTAPÉ, J., WOODRUFF, J., BRAUCKMANN, A., MAGALHÃES, J. W. D. S., AND O'BOYLE, M. F. P. Exebench: an ml-scale dataset of executable c functions. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (New York, NY, USA, Jun 2022), MAPS 2022, Association for Computing Machinery, p. 50–59.

[3] CUMMINS, C., SEEKER, V., GRUBISIC, D., ELHOUSHI, M., LIANG, Y., ROZIERE, B., GEHRING, J., GLOECKLE, F., HAZELWOOD, K., SYNNAEVE, G., AND LEATHER, H. Large language models for compiler optimization. arXiv:2309.07062 [cs].

[4] CUMMINS, C., WASTI, B., GUO, J., CUI, B., ANSEL, J., GOMEZ, S., JAIN, S., LIU, J., TEYTAUD, O., STEINER, B., ET AL. Compilergym: Robust, performant compiler optimization environments for ai research. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2022), IEEE, pp. 92–105.

[5] DA SILVA, A. F., KIND, B. C., DE SOUZA MAGALHÃES, J. W., ROCHA, J. N., FERREIRA GUIMARÃES, B. C., AND QUINÃO PEREIRA, F. M. Anghabench: A suite with one million compilable c benchmarks for code-size reduction. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (Feb 2021), p. 378–390.

[6] GAMBLIN, T., LEGENDRE, M., COLLETTE, M. R., LEE, G. L., MOODY, A., DE SUPINSKI, B. R., AND FUTRAL, S. The spack package manager: bringing order to hpc software chaos. p. 1–12.

[7] GUO, Z. C., AND MOSES, W. S. Enabling transformers to understand low-level programs.

[8] HAJ-ALI, A., AHMED, N. K., WILLKE, T., SHAO, Y. S., ASANOVIC, K., AND STOICA, I. Neurovectorizer: end-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (New York, NY, USA, Feb 2020), CGO 2020, Association for Computing Machinery, p. 242–255.

[9] HOFFMANN, J., BORGEAUD, S., MENSCH, A., BUCHATSKAYA, E., CAI, T., RUTHERFORD, E., CASAS, D. D. L., HENDRICKS, L. A., WELBL, J., CLARK, A., ET AL. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556* (2022).

[10] HUANG, Q., HAJ-ALI, A., MOSES, W., XIANG, J., STOICA, I., ASANOVIC, K., AND WAWRZYNEK, J. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning. arXiv:2003.00671 [cs].

[11] KADOSH, T., HASABNIS, N., MATTSON, T., PINTER, Y., AND OREN, G. Quantifying openmp: Statistical insights into usage and adoption. *arXiv preprint arXiv:2308.08002* (2023).

[12] KANDPAL, N., WALLACE, E., AND RAFFEL, C. Deduplicating training data mitigates privacy risks in language models. In *International Conference on Machine Learning* (2022), PMLR, pp. 10697–10707.

[13] KOCETKOV, D., LI, R., ALLAL, L. B., LI, J., MOU, C., FERRANDIS, C. M., JERNITE, Y., MITCHELL, M., HUGHES, S., WOLF, T., ET AL. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533* (2022).

[14] KULKARNI, S., CAVAZOS, J., WIMMER, C., AND SIMON, D. Automatic construction of inlining heuristics using machine learning. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (Feb 2013), p. 1–12.

[15] LACHAUX, M.-A., ROZIERE, B., CHANUSSOT, L., AND LAMPLE, G. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511* (2020).

[16] LATTNER, C., AND ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* (2004), IEEE, pp. 75–86.

[17] LI, Y., CHOI, D., CHUNG, J., KUSHMAN, N., SCHRITTWIESER, J., LEBLOND, R., ECCLES, T., KEELING, J., GIMENO, F., DAL LAGO, A., ET AL. Competition-level code generation with alphacode. *Science 378*, 6624 (2022), 1092–1097.

[18] MARKOVTSEV, V., AND LONG, W. Public git archive: a big code dataset for all. In *Proceedings of the 15th International Conference on Mining Software Repositories* (2018), pp. 34–37.

[19] PURI, R., KUNG, D. S., JANSSEN, G., ZHANG, W., DOMENICONI, G., ZOLOTOV, V., DOLBY, J., CHEN, J., CHOUDHURY, M., DECKER, L., ET AL. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655* (2021).

[20] ROZIÈRE, B., GEHRING, J., GLOECKLE, F., SOOTLA, S., GAT, I., TAN, X. E., ADI, Y., LIU, J., REMEZ, T., RAPIN, J., ET AL. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[21] TROFIN, M., QIAN, Y., BREVDO, E., LIN, Z., CHOROMANSKI, K., AND LI, D. Mlgo: a machine learning guided compiler optimizations framework. *arXiv preprint arXiv:2101.04808* (2021).

[22] YANG, C., WANG, X., LU, Y., LIU, H., LE, Q. V., ZHOU, D., AND CHEN, X. Large language models as optimizers. *arXiv preprint arXiv:2309.03409* (2023).

[23] ZHU, M., JAIN, A., SURESH, K., RAVINDRAN, R., TIPIRNENI, S., AND REDDY, C. K. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474* (2022).

A Permissively-Licensed Dataset Size

Source	Total	Under Permissive Licenses	with License Files
Rust	586 GB	468 GB	394 GB
Julia	210 GB	186 GB	186 GB
Spack	118 GB	67.3 GB	45.5 GB
Swift	7.35 GB	6.93 GB	6.93 GB
Total	921 GB	728 GB	632 GB

Table 3: Permissively licensed subset of ComPile in Bitcode size.

To filter our closed-source dataset for permissively licensed projects, we filter the entire database of projects compiler into ComPile for the `MIT`, `Apache-2.0`, the `BSD-3-Clause`, and the `BSD-2-Clause` licenses. For this we obtain the license information from package repositories, GitHub, and in part manually using the `go-license-detector` [5], and distribute provenance information, and license text along with the dataset to comply with terms.

---

[5] https://github.com/go-enry/go-license-detector