Static Program Analysis - DCC888
Federal University of Minas Gerais
Jun 2024

Reproducibility of Published Results

Kaio Henrique Andrade Ananias

The extra assignment

This is the extra assignment of the Static Program Analysis course (DCC888) from the Federal University of Minas Gerais. The task has the following requirements:

1. Choose a paper presented in a recent compiler-related conference, such as CGO, PLDI, or CC.
2. Download the material that the authors have made publicly available for the paper. If there is no such material, but the student still wants to reproduce the paper's results, he/she can write to the paper's authors, asking for their implementation.
3. Run the implementation, trying to reproduce at least one of the experiments in the paper. Even if it is impossible to reproduce the experiments, the student can still write a report about his/her tries, to claim the extra points.
4. Write a short report about the entire procedure. This report must contain the URL of the material that has been downloaded, plus a brief description of the student's experience with it. If the student has not been able to reproduce those experiments, than the report must explain the reasons for this failure.

1 The Paper

I have chosen the paper: Engelke, A., & Schwarz, T. (2024, March). **Compile-Time Analysis of Compiler Frameworks for Query Compilation**. In 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (pp. 233-244). IEEE. DOI: 10.1109/CGO57630.2024.10444856

The paper explores the efficiency of different compiler frameworks such as GCC, LLVM, Cranelift, and DirectEmit in the context of Just-In-Time Compilers. To conduct the experiments the authors have used the Umbra compiling database system to measure the compile-time and run-time of database queries on the TPC-DS and TPC-H benchmark.

The analysis revealed that while LLVM provides the best execution performance, it can be optimized for faster compile times. Cranelift, on the other hand, offers similar run-time performance to unoptimized LLVM but compiles 20-35% faster. The single-pass compiler stood out by compiling significantly faster (16 times faster than Cranelift) while maintaining comparable execution performance.

2 Artifact Evaluation

The paper has an appendix with detailed information on how to conduct and evaluate the results from the artifacts. They explain the dependencies needed, artifact metadata, and experiment customization. The authors have made publicly through the *Zenodo* platform all the necessary files to reproduce the experiments. The artifact can be downloaded through the link `https://doi.org/10.5281/zenodo.10357363`.

Inside the artifact folder, there is a `README.md` file explaining how to reproduce each of the experiments conducted in the paper. The authors have created a `Makefile` to automatically generate and run all the experiments, including some of the figures that the paper shows. You can also verify the original raw result generated within the hardware configuration presented in the paper by exploring the `results-paper` directory.

3 Reproducubility

I have conducted the experiments using an Apple M1 with 4 cores and 16GiB of RAM running macOS 13.2. Unfortunately, the authors didn't share a `Dockerfile` with the necessary environment to reproduce the results. I highly recommend running the experiment inside a docker if you're using an Apple operating system. The makefile didn't work very well for some steps on the native machine.

The steps I have taken to proceed with the experiment:

1. Install Docker (24.0.5) [1]
2. Pull the Ubuntu 24.04 LTS

   ```
   $ docker run -ti ubuntu /bin/bash
   ```

3. Install the necessary dependencies in the container

   ```
   $ apt-get install build-essential \\
       cmake gcc-12 perl sed git latexmk texlive-full --assume-yes
   ```

3.1 Running the experiment

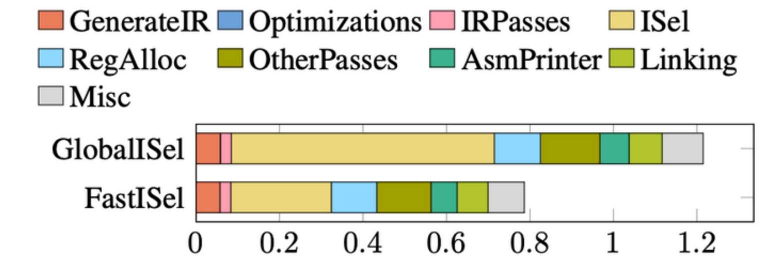
You need to download the benchmark files used in the paper. This can be achieved by running the `make prepare` command. This will download, decompress, and initialize the databases with both *TPCDS* and *TCPH* files even though the latter can only be run on an x86-64 machine. This step can take a while.

You have different options according to the artifact documentation to run the experiments. You can run the full experiment with `make all`, which will repeat compilation/execution 20 times after an extra warm-up run (not measured); reporting the sum of the compilation/execution time of all queries. You can also run each configuration at a time. For example: `make res-llvmcheap-aarch64`.

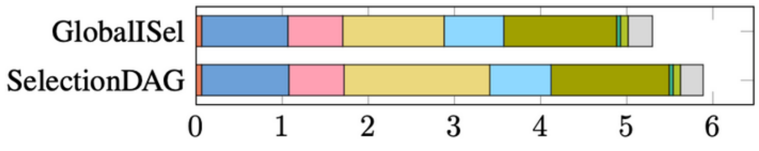
[1]https://www.docker.com/products/docker-desktop/

Figure 1 and 2 were automatically generated by the command `make plots` using the result of the previous step. As you can see, the results are very similar (but not the same) to Figure 6 and Table III of the paper. The similarity can be explained by the fact that I have used the same environment to conduct the Aarch64 version of the results. Most importantly, the results convey the ones shown in the paper.

I. LLVM



(a) Compile-time [s] of LLVM-cheap on AArch64.



(b) Compile-time [s] of LLVM-optimized on AArch64.

Figure 1: Compile-time breakdown of LLVM on AArch64, comparing FastISel, SelectionDAG, and GlobalISel.

	x86-64 comp	exec	AArch64 comp	exec
Interpreter	—	—	0.025326 s	61.622502 s
DirectEmit	—	—	—	—
Cranelift	—	—	0.641063 s	20.13521 s
LLVM-cheap	—	—	0.799514 s	24.205422 s
LLVM-opt	—	—	6.173232 s	16.951506 s
GCC	—	—	47.891793 s	19.186986 s

Figure 2: Compile-time and Execution performance of the different back-ends on TPC-DS sf=10.

I have generated Figures 3 and 4 by plotting the results of `res-cranelift-aarch64` and `res-gcc-aarch64` on a bar chart. Both graphs represent the compilation time taken by each back-end, in this case, Cranelift, and GCC. Those graphs are not presented in the paper since the authors highly most of the x86 results to the detriment of Aarch64.

The full reproduction of the experiments takes approximately 3 and a half hours to complete in my setup configuration. It used the 16 GiB of RAM and 100% of the CPU almost the entire time.

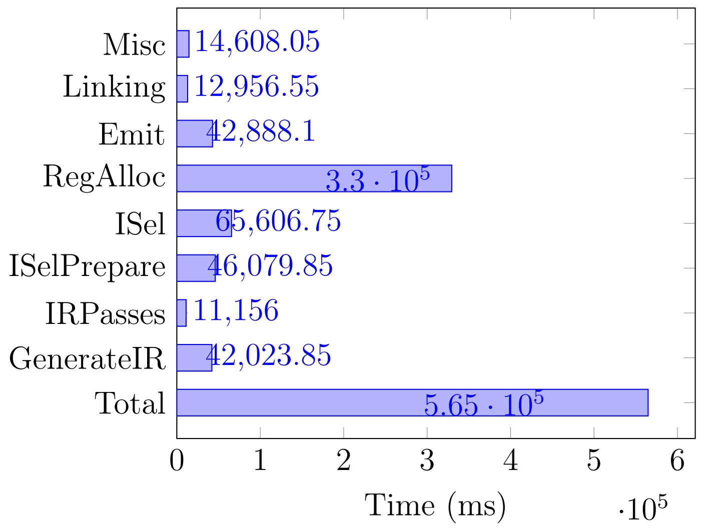


Figure 3: Compile time of Cranelift on Aarch64

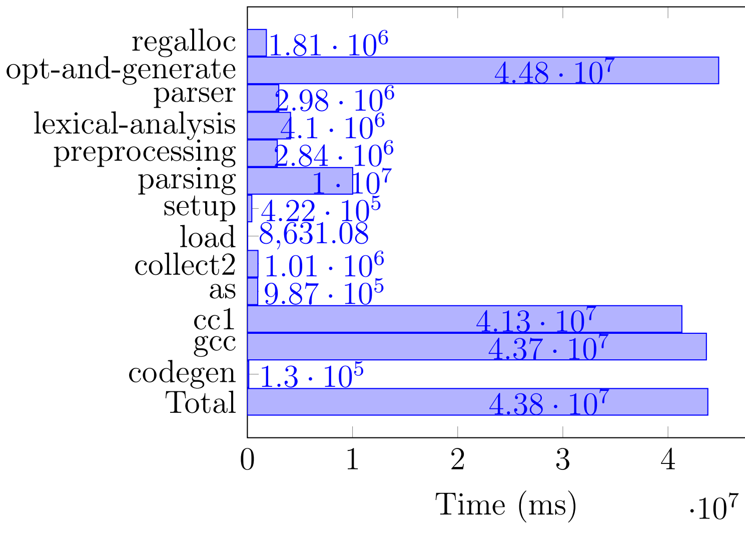


Figure 4: Compile time of GCC on Aarch64

4 Conclusion

The authors took great care in creating well-writing documentation and a configuration setup to make it easy for readers to reproduce the results presented in the paper. I was able to overcome all issues found in the installation and generation of the results. Some of the figures presented in this report were automatically created by the scripts available in the artifact making it effortless to reproduce.

Unfortunately, I wasn't able to reproduce the experiments for the `x86-64` version of the benchmark due to the lack of a computer with a CPU of this architecture. A more precise evaluation could consider both architectures.

Overall, I had a good experience reproducing results from a Compilers paper. Some of the errors faced were easily overcome by carrying out the experiments within a virtualized environment simulating a Linux machine.