**The Extra Project Assignment - Reproducibility of Published Results**

João Vitor Fröhlich
ID: 2024675039

**1. Choose a paper presented in a recent compiler related conference, such as CGO, PLDI or CC.**

I chose the paper “Stale Profile Matching” by Amir Ayupov, Maksin Pachenco and Sergey Pupyrev, presented at CC 2024 and available at: https://doi.org/10.1145/3640537.3641573.

---

**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 authors of the paper, asking for their implementation.**

The material is publicly available at the llvm-project repository, under the BOLT project: https://github.com/llvm/llvm-project/.

---

**3. Run the implementation, trying to reproduce at least one of the experiments in the paper. Even if it is not possible to reproduce the experiments, the student can still write a report about his/her tries, to claim the extra points.**

Two experiments were run with some changes, and the changes, procedures and results are discussed in the next section.

---

**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, then the report must explain the reasons for this failure. Two examples can be found below:**

There is no artifact to reproduce the experiment. There is a tutorial available at: https://github.com/llvm/llvm-project/blob/main/bolt/docs/OptimizingClang.md, that shows how to optimize clang using BOLT, and this was the base used for running the experiment.

The paper presented 5 reproducible experiments, which use open source applications (Clang, GCC, MySQL, Rocksdb and Chromium), as well as one

experiment with applications from Meta, which couldn't be replicated for obvious reasons. For this report, I will describe my experience reproducing the Clang and GCC experiments.

The GCC experiment was run in the same setup in the point of view of software: version 9.3.0 and 8.3.0, but Clang was run with different versions: clang 17 and clang 15. The reason behind this change was because I wasn't able to compile a version older than clang 15 on my computer, so I wasn't able to reproduce with the 3000 commits difference that the authors used. One important thing to note is that, in order to compile GCC 8.3.0, I needed to apply a patch to fix some error that occurred, as described in https://stackoverflow.com/a/63939657.

The experiment consists in 5 steps:

1. Compile an older version of the application (in this case, GCC 8.3.0 and clang 15);
2. Get a profile for this old version using perf;
3. Compile the new version of the application (GCC 9.3.0 and clang 17);
4. Get a profile for this new version;
5. Optimize the new version using the collected profiles.

Just as a remainder, GCC is available at: https://gcc.gnu.org/releases.html and clang at: https://github.com/llvm/llvm-project/. Now I will describe each step of the experiment:

1. Compile an older version of the application

The steps to compile each compiler differs.

To compile GCC, there is the configuration script, which was run following the configs presented in the paper. Also, the makefile has a target called profiledbootstrap, which produces a PGO version of GCC. Also, the compiled version was done with the --emit-relocs option from ld linker, optimization -O3 and -flto, to enable LTO. This same setup was run to GCC 8.3.0 and GCC 9.3.0, but for the GCC 8.3.0 there was the additional step of applying the patch. The commands and flags to configure and build GCC are presented below.

```
$SRCDIR/configure --enable-linker-build-id --enable-bootstrap --enable-languages=c,c++ --with-gnu-as --with-gnu-ld --disable-multilib

make LDFLAGS="-Wl,-q" BOOT_CFLAGS="-O3 -flto -g -Wl,-q -fno-reorder-blocks-and-partition" profiledbootstrap
make DESTDIR=$DESTDIR install
```

To compile Clang the process is a bit more tedious, since you need to run manually each step of bootstrapping a compiler, which consists of three steps: first we compile a bootstrap compiler using an existent compiler, to be enable to use it to compile itself, second, we produce a full compiler build using the bootstrap compiler, and

third, we produce another bootstrap compiler to keep optimizing the second stage compiler. So first, we compile a bootstrap compiler with some of Clang features, that will be important in the second stage. Differently from GCC, Clang uses cmake, and also suggests to use Ninja as the build tool.

```
mkdir ${TOPLEV}/stage1
cd ${TOPLEV}/stage1
cmake -G Ninja ${TOPLEV}/llvm-project/llvm -DLLVM_TARGETS_TO_BUILD=X86 \
      -DCMAKE_BUILD_TYPE=Release \
      -DCMAKE_C_COMPILER=gcc -DCMAKE_CXX_COMPILER=g++ -DCMAKE_ASM_COMPILER=gcc \
      -DLLVM_ENABLE_PROJECTS="clang;lld" \
      -DLLVM_ENABLE_RUNTIMES="compiler-rt" \
      -DCOMPILER_RT_BUILD_SANITIZERS=OFF -DCOMPILER_RT_BUILD_XRAY=OFF \
      -DCOMPILER_RT_BUILD_LIBFUZZER=OFF \
      -DCMAKE_INSTALL_PREFIX=${TOPLEV}/stage1/install
ninja install -j4
```

With this stage 1 compiler, we produce an instrumented version of Clang on stage 2:

```
CPATH=${TOPLEV}/stage1/install/bin/
cmake -G Ninja ${TOPLEV}/llvm-project/llvm -DLLVM_TARGETS_TO_BUILD=X86 \
    -DCMAKE_BUILD_TYPE=Release \
    -DCMAKE_C_COMPILER=$CPATH/clang -DCMAKE_CXX_COMPILER=$CPATH/clang++ \
    -DLLVM_ENABLE_PROJECTS="clang;lld" \
    -DLLVM_USE_LINKER=lld -DLLVM_BUILD_INSTRUMENTED=ON \
    -DCMAKE_INSTALL_PREFIX=${TOPLEV}/stage2-prof-gen/install
ninja install -j4
```

With this instrumented build, we produce a stage 3 compiler to get profile data, in order to produce a PGO version of the stage 2 later:

```
CPATH=${TOPLEV}/stage2-prof-gen/install/bin
cmake -G Ninja ${TOPLEV}/llvm-project/llvm -DLLVM_TARGETS_TO_BUILD=X86 \
    -DCMAKE_BUILD_TYPE=Release \
    -DCMAKE_C_COMPILER=$CPATH/clang -DCMAKE_CXX_COMPILER=$CPATH/clang++ \
    -DLLVM_ENABLE_PROJECTS="clang" \
    -DLLVM_USE_LINKER=lld -DCMAKE_INSTALL_PREFIX=${TOPLEV}/stage3-train/install
ninja install -j4
```

Now, we use the profile collected here to produce a PGO optimized build of Clang, as well as use --emit-relocs to build it with LTO. In this new stage 2 build, we shall reuse the stage 1 compiler:

```
CPATH=${TOPLEV}/stage1/install/bin/
export LDFLAGS="-Wl,-q"
cmake -G Ninja ${TOPLEV}/llvm-project/llvm -DLLVM_TARGETS_TO_BUILD=X86 \
    -DCMAKE_BUILD_TYPE=Release \
    -DCMAKE_C_COMPILER=$CPATH/clang -DCMAKE_CXX_COMPILER=$CPATH/clang++ \
    -DLLVM_ENABLE_PROJECTS="clang;lld" \
    -DLLVM_ENABLE_LTO=Full \
    -DLLVM_PROFDATA_FILE=${TOPLEV}/stage2-prof-gen/profiles/clang.profdata \
    -DLLVM_USE_LINKER=lld \
    -DCMAKE_INSTALL_PREFIX=${TOPLEV}/stage2-prof-use-lto/install
ninja install -j4
```

Finally, we have an optimized version of Clang that we might use with BOLT.

---

2. Get a profile for this old version using perf.

To do this, we can run the same steps for GCC and Clang, since the way to collect a profile for an application is using it, and both of them are used to compile C/C++ programs.

First, we produce a new stage 3 compiler for Clang, then we compile it (using GCC or Clang, depending on the cmake options) using perf. The commands to do this are the following:

```
CPATH=${TOPLEV}/stage2-prof-use-lto/install/bin/
cmake -G Ninja ${TOPLEV}/llvm-project/llvm -DLLVM_TARGETS_TO_BUILD=X86 -DCMAKE_BUILD_TYPE=Release \
    -DCMAKE_C_COMPILER=$CPATH/clang -DCMAKE_CXX_COMPILER=$CPATH/clang++ \
    -DLLVM_USE_LINKER=lld -DCMAKE_INSTALL_PREFIX=${TOPLEV}/stage3/install
sudo perf record -e cycles:u -j any,u -- ninja install
```

One important thing to notice here is that, in order to run perf with the -e cycles:u option, I needed an Intel CPU, to use the LBR (Last Branch Record) feature, which can generate more precise profile information.

The perf command generates a perf.data file that contains a lot of data (around 30GB) for each compiler. This file will be processed in step 5.

---

The steps 3 and 4 are very similar to steps 1 and 2, so I won't describe them here.

---

5. Optimize the new version using the collected profiles.

First, we need to process the perf.data file produced at step 2 and 4, to feed BOLT with a format it can recognize. To do this, I've used the perf2bolt application,

provided along with BOLT. This application receives a perf.data file and the binary and produces a .fdata and a .yaml files as output, that contains information from the profile and the execution of this profile. To run the application, I've used the following command (this example uses Clang 15):

perf2bolt $CPATH/clang-15 -p perf.data -o clang-15.fdata -w clang-15.yaml

Now, we can use this clang-15.yaml file with BOLT to optimize binaries. For example, to optimize Clang 15 using the profile of Clang 15, we will run:

llvm-bolt $CPATH/clang-15 -o $CPATH/clang-15.bolt -b clang-15.yaml -reorder-blocks=ext-tsp -reorder-functions=hfsort+ -split-functions -split-all-cold -dyno-stats -icf=1 -use-gnu-stack

Which will produce an optimized binary called clang-15.bolt.

The same thing can be done to optimize Clang 17 using the profile of Clang 15, but this will have a lot of stale profile information. So, to use the paper's work, I pass the flag --infer-stale-profile to llvm-bolt, and this will make it try to reuse the profile information. The same thing works with GCC.

---

Lastly, we need to check if the results match with the results described in the paper. It's worth noting that the hardware setup is very different.

For Clang:

Optimizing Clang 17 with the profile of Clang 17 got a speedup of 8.79%, optimizing it with the profile of Clang 15 (without the --infer-stale-profile flag) got a speedup of 0.98%, and using the --infer-stale-profile flag a speedup of 2.32%, against the 11.9%, 3.4% and 9.3% presented in the paper, respectively. The results shown here look consistent with the paper's results.

For GCC:

Optimizing GCC 9.3.0 with the profile of GCC 9.3.0 got a speedup of 6.17%, optimizing it with the profile of GCC 8.3.0 (without --infer-stale-profile) 0.63% and with the --infer-stale-profile flag a **slowdown** of 0.28%, which is very opposed to the 7.9%, 2.5% and 5.9% of speedup, respectively. These results are very opposed to consistent. Some guesses about why the results are different are the optimization flags (-O3 and -flto), the patch that I needed to apply and also the hardware setup. Also, there could be some mistake I've made during the GCC setup, since there wasn't any tutorial on how to optimize GCC, which made the experiment harder to set up and more error prone.

---

I found it very hard to run these experiments, since I needed a Intel CPU computer, and also there wasn’t much material available on how to run this, so I had to come up with many ideas on how to run them, and this might have affected the results.