Profile Guided Optimization

`rustc` supports doing profile-guided optimization (PGO). This chapter describes what PGO is, what it is good for, and how it can be used.

What Is Profiled-Guided Optimization?

The basic concept of PGO is to collect data about the typical execution of a program (e.g. which branches it is likely to take) and then use this data to inform optimizations such as inlining, machine-code layout, register allocation, etc.

There are different ways of collecting data about a program's execution. One is to run the program inside a profiler (such as `perf`) and another is to create an instrumented binary, that is, a binary that has data collection built into it, and run that. The latter usually provides more accurate data and it is also what is supported by `rustc`.

Usage

Generating a PGO-optimized program involves following a workflow with four steps:

1. Compile the program with instrumentation enabled (e.g. `rustc -Cprofile-generate=/tmp/pgo-data main.rs`)
2. Run the instrumented program (e.g. `./main`) which generates a `default_<id>.profraw` file
3. Convert the `.profraw` file into a `.profdata` file using LLVM's `llvm-profdata` tool
4. Compile the program again, this time making use of the profiling data (for example `rustc -Cprofile-use=merged.profdata main.rs`)

An instrumented program will create one or more `.profraw` files, one for each instrumented binary. E.g. an instrumented executable that loads two instrumented dynamic libraries at runtime will generate three `.profraw` files. Running an instrumented binary multiple times, on the other hand, will re-use the respective `.profraw` files, updating them in place.

These `.profraw` files have to be post-processed before they can be fed back into the compiler. This is done by the `llvm-profdata` tool. This tool is most easily installed via

```
rustup component add llvm-tools-preview
```

Note that installing the `llvm-tools-preview` component won't add `llvm-profdata` to the `PATH`. Rather, the tool can be found in:

```
~/.rustup/toolchains/<toolchain>/lib/rustlib/<target-triple>/bin/
```

Alternatively, an `llvm-profdata` coming with a recent LLVM or Clang version usually works too.

The `llvm-profdata` tool merges multiple `.profraw` files into a single `.profdata` file that can then be fed back into the compiler via `-Cprofile-use`:

```
# STEP 1: Compile the binary with instrumentation
rustc -Cprofile-generate=/tmp/pgo-data -O ./main.rs

# STEP 2: Run the binary a few times, maybe with common sets of args.
#         Each run will create or update `.profraw` files in /tmp/pgo-data
./main mydata1.csv
./main mydata2.csv
./main mydata3.csv

# STEP 3: Merge and post-process all the `.profraw` files in /tmp/pgo-data
llvm-profdata merge -o ./merged.profdata /tmp/pgo-data

# STEP 4: Use the merged `.profdata` file during optimization. All `rustc`
#         flags have to be the same.
rustc -Cprofile-use=./merged.profdata -O ./main.rs
```

A Complete Cargo Workflow

Using this feature with Cargo works very similar to using it with `rustc` directly. Again, we generate an instrumented binary, run it to produce data, merge the data, and feed it back into the compiler. Some things of note:

- We use the `RUSTFLAGS` environment variable in order to pass the PGO compiler flags to the compilation of all crates in the program.
- We pass the `--target` flag to Cargo, which prevents the `RUSTFLAGS` arguments to be passed to Cargo build scripts. We don't want the build scripts to generate a bunch of `.profraw` files.
- We pass `--release` to Cargo because that's where PGO makes the most sense. In theory, PGO can also be done on debug builds but there is little reason to do so.
- It is recommended to use *absolute paths* for the argument of `-Cprofile-generate` and `-Cprofile-use`. Cargo can invoke `rustc` with varying working directories, meaning that `rustc` will not be able to find the supplied `.profdata` file. With absolute paths this is not an issue.
- It is good practice to make sure that there is no left-over profiling data from previous compilation sessions. Just deleting the directory is a simple way of doing so (see `STEP 0` below).

This is what the entire workflow looks like:

```
# STEP 0: Make sure there is no left-over profiling data from previous runs
rm -rf /tmp/pgo-data

# STEP 1: Build the instrumented binaries
RUSTFLAGS="-Cprofile-generate=/tmp/pgo-data" \
    cargo build --release --target=x86_64-unknown-linux-gnu

# STEP 2: Run the instrumented binaries with some typical data
./target/x86_64-unknown-linux-gnu/release/myprogram mydata1.csv
./target/x86_64-unknown-linux-gnu/release/myprogram mydata2.csv
./target/x86_64-unknown-linux-gnu/release/myprogram mydata3.csv

# STEP 3: Merge the `.profraw` files into a `.profdata` file
llvm-profdata merge -o /tmp/pgo-data/merged.profdata /tmp/pgo-data

# STEP 4: Use the `.profdata` file for guiding optimizations
```

```
RUSTFLAGS="-Cprofile-use=/tmp/pgo-data/merged.profdata" \
    cargo build --release --target=x86_64-unknown-linux-gnu
```

Troubleshooting

- It is recommended to pass `-Cllvm-args=-pgo-warn-missing-function` during the `-Cprofile-use` phase. LLVM by default does not warn if it cannot find profiling data for a given function. Enabling this warning will make it easier to spot errors in your setup.

- There is a known issue in Cargo prior to version 1.39 that will prevent PGO from working correctly. Be sure to use Cargo 1.39 or newer when doing PGO.

Further Reading

`rustc`'s PGO support relies entirely on LLVM's implementation of the feature and is equivalent to what Clang offers via the `-fprofile-generate` / `-fprofile-use` flags. The Profile Guided Optimization section in Clang's documentation is therefore an interesting read for anyone who wants to use PGO with Rust.