Profile-guided optimization - The Go Programming Language

Starting in Go 1.20, the Go compiler supports profile-guided optimization (PGO) to further optimize builds.

Table of Contents:

Overview

Profile-guided optimization (PGO), also known as feedback-directed optimization (FDO), is a compiler optimization technique that feeds information (a profile) from representative runs of the application back into to the compiler for the next build of the application, which uses that information to make more informed optimization decisions. For example, the compiler may decide to more aggressively inline functions which the profile indicates are called frequently.

In Go, the compiler uses CPU pprof profiles as the input profile, such as from runtime/pprof or net/http/pprof.

As of Go 1.21, benchmarks for a representative set of Go programs show that building with PGO improves performance by around 2-7%. We expect performance gains to generally increase over time as additional optimizations take advantage of PGO in future versions of Go.

Collecting profiles

The Go compiler expects a CPU pprof profile as the input to PGO. Profiles generated by the Go runtime (such as from runtime/pprof and net/http/pprof) can be used directly as the compiler input. It may also be possible to use/convert profiles from other profiling systems. See the appendix for additional information.

For best results, it is important that profiles are *representative* of actual behavior in the application's production environment. Using an unrepresentative profile is likely to result in a binary with little to no improvement in production. Thus, collecting profiles directly from the production environment is recommended, and is the primary method that Go's PGO is designed for.

The typical workflow is as follows:

Build and release an initial binary (without PGO).

Collect profiles from production.

When it's time to release an updated binary, build from the latest source and provide the production profile.

GOTO 2

Go PGO is generally robust to skew between the profiled version of an application and the version building with the profile, as well as to building with profiles collected from already-optimized binaries. This is what makes this iterative lifecycle possible. See the AutoFDO section for additional details about this workflow.

If it is difficult or impossible to collect from the production environment (e.g., a command-line tool distributed to end users), it is also possible to collect from a representative benchmark. Note that constructing representative benchmarks is often quite difficult (as is keeping them representative as the application evolves). In particular, *microbenchmarks are usually bad candidates for PGO profiling*, as they only exercise a small part of the application, which yields small gains when applied to the whole program.

Building with PGO

The standard approach to building is to store a pprof CPU profile with filename `default.pgo` in the main package directory of the profiled binary. By default, `go build` will detect `default.pgo` files automatically and enable PGO.

Committing profiles directly in the source repository is recommended as profiles are an input to the build important for reproducible (and performant!) builds. Storing alongside the source simplifies the build experience as there are no additional steps to get the profile beyond fetching the source.

For more complex scenarios, the `go build -pgo` flag controls PGO profile selection. This flag defaults to `-pgo=auto` for the `default.pgo` behavior described above. Setting the flag to `-pgo=off` disables PGO optimizations entirely.

If you cannot use `default.pgo` (e.g., different profiles for different scenarios of one binary, unable to store profile with source, etc), you may directly pass a path to the profile to use (e.g., `go build -pgo=/tmp/foo.pprof`).

Note: A path passed to `-pgo` applies to all main packages. e.g., `go build -pgo=/tmp/foo.pprof ./cmd/foo ./cmd/bar` applies `foo.pprof` to both binaries `foo` and `bar`, which is often not what you want. Usually different binaries should have different profiles, passed via separate `go build` invocations.

Note: Before Go 1.21, the default is `-pgo=off`. PGO must be explicitly enabled.

Notes

Collecting representative profiles from production

Your production environment is the best source of representative profiles for your application, as described in Collecting profiles.

The simplest way to start with this is to add net/http/pprof to your application and then fetch `/debug/pprof/profile?seconds=30` from an arbitrary instance of your service. This is a great way to get started, but there are ways that this may be unrepresentative:

This instance may not be doing anything at the moment it gets profiled, even though it is usually busy.

Traffic patterns may change throughout the day, making behavior change throughout the day.

Instances may perform long-running operations (e.g., 5 minutes doing operation A, then 5 minutes doing operation B, etc). A 30s profile will likely only cover a single operation type.

Instances may not receive fair distributions of requests (some instances receive more of one type of request than others).

A more robust strategy is collecting multiple profiles at different times from different instances to limit the impact of differences between individual instance profiles. Multiple profiles may then be merged into a single profile for use with PGO.

Many organizations run "continuous profiling" services that perform this kind of fleet-wide sampling profiling automatically, which could then be used as a source of profiles for PGO.

Merging profiles

The pprof tool can merge multiple profiles like this:

```
$ go tool pprof -proto a.pprof b.pprof > merged.pprof
```

This merge is effectively a straightforward sum of samples in the input, regardless of wall duration of the profile. As a result, when profiling a small time slice of an application (e.g., a server that runs indefinitely), you likely want to ensure that all profiles have the same wall duration (i.e., all profiles are collected for 30s). Otherwise, profiles with longer wall duration will be overrepresented in the merged profile.

AutoFDO

Go PGO is designed to support an "AutoFDO" style workflow.

Let's take a closer look at the workflow described in Collecting profiles:

Build and release an initial binary (without PGO).

Collect profiles from production.

When it's time to release an updated binary, build from the latest source and provide the production profile.

GOTO 2

This sounds deceptively simple, but there are a few important properties to note here:

Development is always ongoing, so the source code of the profiled version of the binary (step 2) is likely slightly different from the latest source code getting built (step 3). Go PGO is designed to be robust to this, which we refer to as *source stability*.

This is a closed loop. That is, after the first iteration the profiled version of the binary is already PGO-optimized with a profile from a previous iteration. Go PGO is also designed to be robust to this, which we refer to as *iterative stability*.

Source stability is achieved using heuristics to match samples from the profile to the compiling source. As a result, many changes to source code, such as adding new functions, have no impact on matching existing code. When the compiler is not able to match changed code, some optimizations are lost, but note that this is a *graceful degradation*. A single function failing to match may lose out on optimization opportunities, but overall PGO benefit is usually spread across many functions. See the source stability section for more details about matching and degradation.

Iterative stability is the prevention of cycles of variable performance in successive PGO builds (e.g., build #1 is fast, build #2 is slow, build #3 is fast, etc). We use CPU profiles to identify hot functions to target with optimizations. In theory, a hot function could be sped up so much by PGO that it no longer appears hot in the next profile and does not get optimized, making it slow again. The Go compiler takes a conservative approach to PGO optimizations, which we believe prevents significant variance. If you do observe this kind of instability, please file an issue at https://go.dev/issue/new.

Together, source and iterative stability eliminate the requirement for two-stage builds where a first, unoptimized build is profiled as a canary, and then rebuilt with PGO for production (unless absolutely peak performance is required).

Source stability and refactoring

As described in above, Go's PGO makes a best-effort attempt to continue matching samples from older profiles to the current source code. Specifically, Go uses line offsets within functions (e.g., call on 5th line of function foo).

Many common changes will not break matching, including:

Changes in a file outside of a hot function (adding/changing code above or below the function).

Moving a function to another file in the same package (the compiler ignores source filenames altogether).

Some changes that may break matching:

Changes within a hot function (may affect line offsets).

Renaming the function (and/or type for methods) (changes symbol name).

Moving the function to another package (changes symbol name).

If the profile is relatively recent, then differences likely only affect a small number of hot functions, limiting the impact of missed optimizations in functions that fail to match. Still, degradation will slowly accumulate over time since code is rarely refactored *back* to its old form, so it is important to collect new profiles regularly to limit source skew from production.

One situation where profile matching may significantly degrade is a large-scale refactor that renames many functions or moves them between packages. In this case, you may take a short-term performance hit until a new profile shows the new structure.

For rote renames, an existing profile could theoretically be rewritten to change the old symbol names to the new names. github.com/google/pprof/profile contains the primitives required to rewrite a pprof profile in this way, but as of writing no off-the-shelf tool exists for this.

Performance of new code

When adding new code or enabling new code paths with a flag flip, that code will not be present in the profile on the first build, and thus won't receive PGO optimizations until a new profile reflecting the new code is collected. Keep in mind when evaluating the rollout of new code that the initial release will not represent its steady state performance.

Frequently Asked Questions

Is it possible to optimize Go standard library packages with PGO?

Yes. PGO in Go applies to the entire program. All packages are rebuilt to consider potential profile-guided optimizations, including standard library packages.

Is it possible to optimize packages in dependent modules with PGO?

Yes. PGO in Go applies to the entire program. All packages are rebuilt to consider potential profile-guided optimizations, including packages in dependencies. This means that the unique way your application uses a dependency impacts the optimizations applied to that dependency.

Will PGO with an unrepresentative profile make my program slower than no PGO?

It should not. While a profile that is not representative of production behavior will result in optimizations in cold parts of the application, it should not make hot parts of the application slower. If you encounter a program

where PGO results in worse performance than disabling PGO, please file an issue at https://go.dev/issue/new.

Can I use the same profile for different GOOS/GOARCH builds?

Yes. The format of the profiles is equivalent across OS and architecture configurations, so they may be used across different configurations. For example, a profile collected from a linux/arm64 binary may be used in a windows/amd64 build.

That said, the source stability caveats discussed above apply here as well. Any source code that differs across these configurations will not be optimized. For most applications, the vast majority of code is platform-independent, so degradation of this form is limited.

As a specific example, the internals of file handling in package `os` differ between Linux and Windows. If these functions are hot in the Linux profile, the Windows equivalents will not get PGO optimizations because they do not match the profiles.

You may merge profiles of different GOOS/GOARCH builds. See the next question for the tradeoffs of doing so.

How should I handle a single binary used for different workload types?

There is no obvious choice here. A single binary used for different types of workloads (e.g., a database used in a read-heavy way in one service, and write-heavy in another service) may have different hot components, which benefit from different optimizations.

There are three options:

Build different versions of the binary for each workload: use profiles from each workload to build multiple workload-specific builds of the binary. This will provide the best performance for each workload, but may add operational complexity with regard to handling multiple binaries and profile sources.

Build a single binary using only profiles from the “most important” workload: select the “most important” workload (largest footprint, most performance sensitive), and build using profiles only from that workload. This provides the best performance for the selected workload, and likely still modest performance improvements for other workloads from optimizations to common code shared across workloads.

Merge profiles across workloads: take profiles from each workload (weighted by total footprint) and merge them into a single “fleet-wide” profile used to build a single common profile used to build. This likely provides modest performance improvements for all workloads.

How does PGO affect build time?

Enabling PGO builds will likely cause measurable increases in package build times. The most noticeable

component of this is that PGO profiles apply to all packages in a binary, meaning that the first use of a profile requires a rebuild of every package in the dependency graph. These builds are cached like any other, so subsequent incremental builds using the same profile do not require complete rebuilds.

If you experience extreme increases in build time, please file an issue at https://go.dev/issue/new.

Note: Parsing of the profile by the compiler can also add significant overhead, particularly for large profiles. Using large profiles with a large dependency graph can significantly increase build times. This is tracked by https://go.dev/issue/58102 and will be addressed in a future release.

How does PGO affect binary size?

PGO can result in slightly larger binaries due to additional function inlining.

Appendix: alternative profile sources

CPU profiles generated by the Go runtime (via runtime/pprof, etc) are already in the correct format for direct use as PGO inputs. However, organizations may have alternative preferred tooling (e.g., Linux perf), or existing fleet-wide continuous profiling systems which they wish to use with Go PGO.

Profiles from alternative source may be used with Go PGO if converted to the pprof format, provided they follow these general requirements:

One of the sample indices should have type/unit "samples"/"count" or "cpu"/"nanoseconds".

Samples should represent samples of CPU time at the sample location.

The profile must be symbolized (Function.name must be set).

Samples must contain stack frames for inlined functions. If inlined functions are omitted, Go will not be able to maintain iterative stability.

Function.start_line must be set. This is the line number of the start of the function. i.e., the line containing the `func` keyword. The Go compiler uses this field to compute line offsets of samples (`Location.Line.line - Function.start_line`). **Note that many existing pprof converters omit this field.**

Note: Before Go 1.21, DWARF metadata omits function start lines (`DW_AT_decl_line`), which may make it difficult for tools to determine the start line.

See the PGO Tools page on the Go Wiki for additional information about PGO compatibility of specific third-party tools.