Profile-guided optimization in Go 1.21 - The Go Programming Language

Michael Pratt 5 September 2023

The Go Blog

Earlier in 2023, Go 1.20 shipped a preview of profile-guided optimization (PGO) for users to test. After addressing known limitations in the preview, and with additional refinements thanks to community feedback and contributions, PGO support in Go 1.21 is ready for general production use! See the profile-guided optimization user guide for complete documentation.

Below we will run through an example of using PGO to improve the performance of an application. Before we get to that, what exactly is “profile-guided optimization”?

When you build a Go binary, the Go compiler performs optimizations to try to generate the best performing binary it can. For example, constant propagation can evaluate constant expressions at compile time, avoiding runtime evaluation cost. Escape analysis avoids heap allocations for locally-scoped objects, avoiding GC overheads. Inlining copies the body of simple functions into callers, often enabling further optimization in the caller (such as additional constant propagation or better escape analysis). Devirtualization converts indirect calls on interface values whose type can be determined statically into direct calls to the concrete method (which often enables inlining of the call).

Go improves optimizations from release to release, but doing so is no easy task. Some optimizations are tunable, but the compiler can’t just “turn it up to 11” on every optimization because overly aggressive optimizations can actually hurt performance or cause excessive build times. Other optimizations require the compiler to make a judgment call about what the “common” and “uncommon” paths in a function are. The compiler must make a best guess based on static heuristics because it can’t know which cases will be common at run time.

Or can it?

With no definitive information about how the code is used in a production environment, the compiler can operate only on the source code of packages. But we do have a tool to evaluate production behavior: profiling. If we provide a profile to the compiler, it can make more informed decisions: more aggressively optimizing the most frequently used functions, or more accurately selecting common cases.

Using profiles of application behavior for compiler optimization is known as *Profile-Guided Optimization (PGO)* (also known as Feedback-Directed Optimization (FDO)).

Example

Let's build a service that converts Markdown to HTML: users upload Markdown source to `/render`, which returns the HTML conversion. We can use `gitlab.com/golang-commonmark/markdown` to implement this easily.

Set up

```
$ go mod init example.com/markdown
$ go get gitlab.com/golang-commonmark/markdown@bf3e522c626a
```

In `main.go`:

```
package main

import (
    "bytes"
    "io"
    "log"
    "net/http"
    _ "net/http/pprof"

    "gitlab.com/golang-commonmark/markdown"
)

func render(w http.ResponseWriter, r *http.Request) {
    if r.Method != "POST" {
        http.Error(w, "Only POST allowed", http.StatusMethodNotAllowed)
        return
    }

    src, err := io.ReadAll(r.Body)
    if err != nil {
        log.Printf("error reading body: %v", err)
        http.Error(w, "Internal Server Error",
http.StatusInternalServerError)
        return
    }
```

```
    md := markdown.New(
        markdown.XHTMLOutput(true),
        markdown.Typographer(true),
        markdown.Linkify(true),
        markdown.Tables(true),
    )

    var buf bytes.Buffer
    if err := md.Render(&buf, src); err != nil {
        log.Printf("error converting markdown: %v", err)
        http.Error(w, "Malformed markdown", http.StatusBadRequest)
        return
    }

    if _, err := io.Copy(w, &buf); err != nil {
        log.Printf("error writing response: %v", err)
        http.Error(w, "Internal Server Error",
http.StatusInternalServerError)
        return
    }
}

func main() {
    http.HandleFunc("/render", render)
    log.Printf("Serving on port 8080...")
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Build and run the server:

```
$ go build -o markdown.nopgo.exe
$ ./markdown.nopgo.exe
2023/08/23 03:55:51 Serving on port 8080...
```

Let's try sending some Markdown from another terminal. We can use the `README.md` from the Go project as a sample document:

```
$ curl -o README.md -L "https://raw.githubusercontent.com/golang
/go/c16c2c49e2fa98ae551fc6335215fadd62d33542/README.md"
```

```
$ curl --data-binary @README.md http://localhost:8080/render
<h1>The Go Programming Language</h1>
<p>Go is an open source programming language that makes it easy to build
simple,
reliable, and efficient software.</p>
...
```

Profiling

Now that we have a working service, let's collect a profile and rebuild with PGO to see if we get better performance.

In `main.go`, we imported net/http/pprof which automatically adds a `/debug/pprof/profile` endpoint to the server for fetching a CPU profile.

Normally you want to collect a profile from your production environment so that the compiler gets a representative view of behavior in production. Since this example doesn't have a "production" environment, I have created a simple program to generate load while we collect a profile. Fetch and start the load generator (make sure the server is still running!):

```
$ go run github.com/prattmic/markdown-pgo/load@latest
```

While that is running, download a profile from the server:

```
$ curl -o cpu.pprof "http://localhost:8080/debug/pprof/profile?seconds=30"
```

Once this completes, kill the load generator and the server.

Using the profile

The Go toolchain will automatically enable PGO when it finds a profile named `default.pgo` in the main package directory. Alternatively, the `-pgo` flag to `go build` takes a path to a profile to use for PGO.

We recommend committing `default.pgo` files to your repository. Storing profiles alongside your source code ensures that users automatically have access to the profile simply by fetching the repository (either via the version control system, or via `go get`) and that builds remain reproducible.

Let's build:

```
$ mv cpu.pprof default.pgo
$ go build -o markdown.withpgo.exe
```

We can check that PGO was enabled in the build with `go version`:

```
$ go version -m markdown.withpgo.exe
```

```
./markdown.withpgo.exe: go1.21.0
...
        build   -pgo=/tmp/pgo121/default.pgo
```

Evaluation

We will use a Go benchmark version of the load generator to evaluate the effect of PGO on performance.

First, we will benchmark the server without PGO. Start that server:

```
$ ./markdown.nopgo.exe
```

While that is running, run several benchmark iterations:

```
$ go get github.com/prattmic/markdown-pgo@latest
$ go test github.com/prattmic/markdown-pgo/load -bench=. -count=40 -source
$(pwd)/README.md > nopgo.txt
```

Once that completes, kill the original server and start the version with PGO:

```
$ ./markdown.withpgo.exe
```

While that is running, run several benchmark iterations:

```
$ go test github.com/prattmic/markdown-pgo/load -bench=. -count=40 -source
$(pwd)/README.md > withpgo.txt
```

Once that completes, let's compare the results:

```
$ go install golang.org/x/perf/cmd/benchstat@latest
$ benchstat nopgo.txt withpgo.txt
goos: linux
goarch: amd64
pkg: github.com/prattmic/markdown-pgo/load
cpu: Intel(R) Xeon(R) W-2135 CPU @ 3.70GHz
        │  nopgo.txt  │             withpgo.txt              │
        │   sec/op    │   sec/op     vs base                 │
Load-12   374.5µ ± 1%   360.2µ ± 0%  -3.83% (p=0.000 n=40)
```

The new version is around 3.8% faster! In Go 1.21, workloads typically get between 2% and 7% CPU usage improvements from enabling PGO. Profiles contain a wealth of information about application behavior and Go 1.21 just begins to crack the surface by using this information for a limited set of optimizations. Future releases will continue improving performance as more parts of the compiler take advantage of PGO.

Next steps

In this example, after collecting a profile, we rebuilt our server using the exact same source code used in the original build. In a real-world scenario, there is always ongoing development. So we may collect a profile from production, which is running last week's code, and use it to build with today's source code. That is perfectly fine! PGO in Go can handle minor changes to source code without issue. Of course, over time source code will drift more and more, so it is still important to update the profile occasionally.

For much more information on using PGO, best practices and caveats to be aware of, please see the profile-guided optimization user guide. If you are curious about what is going on under the hood, keep reading!

Under the hood

To get a better understanding of what made this application faster, let's take a look under the hood to see how performance has changed. We are going to take a look at two different PGO-driven optimizations.

Inlining

To observe inlining improvements, let's analyze this markdown application both with and without PGO.

I will compare this using a technique called differential profiling, where we collect two profiles (one with PGO and one without) and compare them. For differential profiling, it's important that both profiles represent the same amount of **work**, not the same amount of time, so I've adjusted the server to automatically collect profiles, and the load generator to send a fixed number of requests and then exit the server.

The changes I have made to the server as well as the profiles collected can be found at https://github.com/prattmic/markdown-pgo. The load generator was run with `-count=300000 -quit`.

As a quick consistency check, let's take a look at the total CPU time required to handle all 300k requests:

```
$ go tool pprof -top cpu.nopgo.pprof | grep "Total samples"
Duration: 116.92s, Total samples = 118.73s (101.55%)
$ go tool pprof -top cpu.withpgo.pprof | grep "Total samples"
Duration: 113.91s, Total samples = 115.03s (100.99%)
```

CPU time dropped from ~118s to ~115s, or about 3%. This is in line with our benchmark results, which is a good sign that these profiles are representative.

Now we can open a differential profile to look for savings:

```
$ go tool pprof -diff_base cpu.nopgo.pprof cpu.withpgo.pprof
File: markdown.profile.withpgo.exe
Type: cpu
```

```
Time: Aug 28, 2023 at 10:26pm (EDT)
Duration: 230.82s, Total samples = 118.73s (51.44%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top -cum
Showing nodes accounting for -0.10s, 0.084% of 118.73s total
Dropped 268 nodes (cum <= 0.59s)
Showing top 10 nodes out of 668
      flat  flat%   sum%        cum   cum%
    -0.03s 0.025% 0.025%     -2.56s  2.16%  gitlab.com/golang-commonmark
/markdown.ruleLinkify
     0.04s 0.034% 0.0084%     -2.19s  1.84%  net/http.(*conn).serve
     0.02s 0.017% 0.025%     -1.82s  1.53%  gitlab.com/golang-commonmark
/markdown.(*Markdown).Render
     0.02s 0.017% 0.042%     -1.80s  1.52%  gitlab.com/golang-commonmark
/markdown.(*Markdown).Parse
    -0.03s 0.025% 0.017%     -1.71s  1.44%  runtime.mallocgc
    -0.07s 0.059% 0.042%     -1.62s  1.36%  net/http.(*ServeMux).ServeHTTP
     0.04s 0.034% 0.0084%     -1.58s  1.33%
net/http.serverHandler.ServeHTTP
    -0.01s 0.0084% 0.017%     -1.57s  1.32%  main.render
     0.01s 0.0084% 0.0084%     -1.56s  1.31%
net/http.HandlerFunc.ServeHTTP
    -0.09s 0.076% 0.084%     -1.25s  1.05%  runtime.newobject
(pprof) top
Showing nodes accounting for -1.41s, 1.19% of 118.73s total
Dropped 268 nodes (cum <= 0.59s)
Showing top 10 nodes out of 668
      flat  flat%   sum%        cum   cum%
    -0.46s  0.39%  0.39%     -0.91s  0.77%  runtime.scanobject
    -0.40s  0.34%  0.72%     -0.40s  0.34%  runtime.nextFreeFast (inline)
     0.36s   0.3%  0.42%      0.36s   0.3%  gitlab.com/golang-commonmark
/markdown.performReplacements
    -0.35s  0.29%  0.72%     -0.37s  0.31%  runtime.writeHeapBits.flush
     0.32s  0.27%  0.45%      0.67s  0.56%  gitlab.com/golang-commonmark
/markdown.ruleReplacements
    -0.31s  0.26%  0.71%     -0.29s  0.24%  runtime.writeHeapBits.write
    -0.30s  0.25%  0.96%     -0.37s  0.31%  runtime.deductAssistCredit
```

```
     0.29s  0.24%  0.72%      0.10s 0.084%  gitlab.com/golang-commonmark
/markdown.ruleText
    -0.29s  0.24%  0.96%     -0.29s  0.24%  runtime.(*mspan).base (inline)
    -0.27s  0.23%  1.19%     -0.42s  0.35%  bytes.(*Buffer).WriteRune
```

When specifying `pprof -diff_base`, the values in displayed in pprof are the *difference* between the two profiles. So, for instance, `runtime.scanobject` used 0.46s less CPU time with PGO than without. On the other hand, `gitlab.com/golang-commonmark/markdown.performReplacements` used 0.36s more CPU time. In a differential profile, we typically want to look at the absolute values (`flat` and `cum` columns), as the percentages aren't meaningful.

`top -cum` shows the top differences by cumulative change. That is, the difference in CPU of a function and all transitive callees from that function. This will generally show the outermost frames in our program's call graph, such as `main` or another goroutine entry point. Here we can see most savings are coming from the `ruleLinkify` portion of handling HTTP requests.

`top` shows the top differences limited only to changes in the function itself. This will generally show inner frames in our program's call graph, where most of the actual work is happening. Here we can see that individual savings are coming mostly from `runtime` functions.

What are those? Let's peek up the call stack to see where they come from:

```
(pprof) peek scanobject$
Showing nodes accounting for -3.72s, 3.13% of 118.73s total
----------------------------------------------------------+-------------
      flat  flat%   sum%        cum   cum%   calls calls% + context
----------------------------------------------------------+-------------
                                             -0.86s 94.51% |
runtime.gcDrain
                                             -0.09s  9.89% |
runtime.gcDrainN
                                              0.04s  4.40% |
runtime.markrootSpans
    -0.46s  0.39%  0.39%     -0.91s  0.77%                |
runtime.scanobject
                                             -0.19s 20.88% |
runtime.greyobject
                                             -0.13s 14.29% |
runtime.heapBits.nextFast (inline)
                                             -0.08s  8.79% |
```

```
runtime.heapBits.next
                                                -0.08s  8.79% |
runtime.spanOfUnchecked (inline)
                                                 0.04s  4.40% |
runtime.heapBitsForAddr
                                                -0.01s  1.10% |
runtime.findObject
-----------------------------------------------------------+-------------
(pprof) peek gcDrain$
Showing nodes accounting for -3.72s, 3.13% of 118.73s total
-----------------------------------------------------------+-------------
      flat  flat%   sum%        cum   cum%   calls calls% + context
-----------------------------------------------------------+-------------
                                                   -1s   100% |
runtime.gcBgMarkWorker.func2
     0.15s  0.13%  0.13%        -1s  0.84%                |
runtime.gcDrain
                                                -0.86s 86.00% |
runtime.scanobject
                                                -0.18s 18.00% |   runtime.
(*gcWork).balance
                                                -0.11s 11.00% |   runtime.
(*gcWork).tryGet
                                                 0.09s  9.00% |
runtime.pollWork
                                                -0.03s  3.00% |   runtime.
(*gcWork).tryGetFast (inline)
                                                -0.03s  3.00% |
runtime.markroot
                                                -0.02s  2.00% |
runtime.wbBufFlush
                                                 0.01s  1.00% |
runtime/internal/atomic.(*Bool).Load (inline)
                                                -0.01s  1.00% |
runtime.gcFlushBgCredit
                                                -0.01s  1.00% |
runtime/internal/atomic.(*Int64).Add (inline)
```

```
-----------------------------------------------------------+-------------
```

So `runtime.scanobject` is ultimately coming from `runtime.gcBgMarkWorker`. The Go GC Guide tells us that `runtime.gcBgMarkWorker` is part of the garbage collector, so `runtime.scanobject` savings must be GC savings. What about `nextFreeFast` and other `runtime` functions?

```
(pprof) peek nextFreeFast$
Showing nodes accounting for -3.72s, 3.13% of 118.73s total
-----------------------------------------------------------+-------------
      flat  flat%   sum%        cum   cum%   calls calls% + context
-----------------------------------------------------------+-------------
                                             -0.40s   100% |
runtime.mallocgc (inline)
    -0.40s  0.34%  0.34%     -0.40s  0.34%                |
runtime.nextFreeFast
-----------------------------------------------------------+-------------
(pprof) peek writeHeapBits
Showing nodes accounting for -3.72s, 3.13% of 118.73s total
-----------------------------------------------------------+-------------
      flat  flat%   sum%        cum   cum%   calls calls% + context
-----------------------------------------------------------+-------------
                                             -0.37s   100% |
runtime.heapBitsSetType
                                                  0     0% |   runtime.
(*mspan).initHeapBits
    -0.35s  0.29%  0.29%     -0.37s  0.31%                |
runtime.writeHeapBits.flush
                                             -0.02s  5.41% |
runtime.arenaIndex (inline)
-----------------------------------------------------------+-------------
                                             -0.29s   100% |
runtime.heapBitsSetType
    -0.31s  0.26%  0.56%     -0.29s  0.24%                |
runtime.writeHeapBits.write
                                              0.02s  6.90% |
runtime.arenaIndex (inline)
-----------------------------------------------------------+-------------
(pprof) peek heapBitsSetType$
```

```
Showing nodes accounting for -3.72s, 3.13% of 118.73s total
----------------------------------------------------------+-------------
      flat  flat%   sum%        cum   cum%   calls calls% + context
----------------------------------------------------------+-------------
                                             -0.82s   100% |
runtime.mallocgc
    -0.12s   0.1%   0.1%     -0.82s  0.69%                |
runtime.heapBitsSetType
                                             -0.37s 45.12% |
runtime.writeHeapBits.flush
                                             -0.29s 35.37% |
runtime.writeHeapBits.write
                                             -0.03s  3.66% |
runtime.readUintptr (inline)
                                             -0.01s  1.22% |
runtime.writeHeapBitsForAddr (inline)
----------------------------------------------------------+-------------
(pprof) peek deductAssistCredit$
Showing nodes accounting for -3.72s, 3.13% of 118.73s total
----------------------------------------------------------+-------------
      flat  flat%   sum%        cum   cum%   calls calls% + context
----------------------------------------------------------+-------------
                                             -0.37s   100% |
runtime.mallocgc
    -0.30s  0.25%  0.25%     -0.37s  0.31%                |
runtime.deductAssistCredit
                                             -0.07s 18.92% |
runtime.gcAssistAlloc
----------------------------------------------------------+-------------
```

Looks like `nextFreeFast` and some of the others in the top 10 are ultimately coming from `runtime.mallocgc`, which the GC Guide tells us is the memory allocator.

Reduced costs in the GC and allocator imply that we are allocating less overall. Let's take a look at the heap profiles for insight:

```
$ go tool pprof -sample_index=alloc_objects -diff_base heap.nopgo.pprof
heap.withpgo.pprof
File: markdown.profile.withpgo.exe
```

```
Type: alloc_objects
Time: Aug 28, 2023 at 10:28pm (EDT)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for -12044903, 8.29% of 145309950 total
Dropped 60 nodes (cum <= 726549)
Showing top 10 nodes out of 58
      flat  flat%   sum%        cum   cum%
  -4974135  3.42%  3.42%   -4974135  3.42%  gitlab.com/golang-commonmark
/mdurl.Parse
  -4249044  2.92%  6.35%   -4249044  2.92%  gitlab.com/golang-commonmark
/mdurl.(*URL).String
   -901135  0.62%  6.97%    -977596  0.67%  gitlab.com/golang-commonmark
/puny.mapLabels
   -653998  0.45%  7.42%    -482491  0.33%  gitlab.com/golang-commonmark
/markdown.(*StateInline).PushPending
   -557073  0.38%  7.80%    -557073  0.38%  gitlab.com/golang-commonmark
/linkify.Links
   -557073  0.38%  8.18%    -557073  0.38%  strings.genSplit
   -436919   0.3%  8.48%    -232152  0.16%  gitlab.com/golang-commonmark
/markdown.(*StateBlock).Lines
   -408617  0.28%  8.77%    -408617  0.28%  net/textproto.readMIMEHeader
    401432  0.28%  8.49%     499610  0.34%  bytes.(*Buffer).grow
    291659   0.2%  8.29%     291659   0.2%  bytes.(*Buffer).String
(inline)
```

The `-sample_index=alloc_objects` option is showing us the count of allocations, regardless of size. This is useful since we are investigating a decrease in CPU usage, which tends to correlate more with allocation count rather than size. There are quite a few reductions here, but let's focus on the biggest reduction, `mdurl.Parse`.

For reference, let's look at the total allocation counts for this function without PGO:

```
$ go tool pprof -sample_index=alloc_objects -top heap.nopgo.pprof | grep
mdurl.Parse
   4974135  3.42% 68.60%    4974135  3.42%  gitlab.com/golang-commonmark
/mdurl.Parse
```

The total count before was 4974135, meaning that `mdurl.Parse` has eliminated 100% of allocations!

Back in the differential profile, let's gather a bit more context:

```
(pprof) peek mdurl.Parse
Showing nodes accounting for -12257184, 8.44% of 145309950 total
----------------------------------------------------------+-------------
      flat  flat%   sum%        cum   cum%   calls calls% + context
----------------------------------------------------------+-------------
                                          -2956806 59.44% |
gitlab.com/golang-commonmark/markdown.normalizeLink
                                          -2017329 40.56% |
gitlab.com/golang-commonmark/markdown.normalizeLinkText
  -4974135  3.42%  3.42%   -4974135  3.42%                |
gitlab.com/golang-commonmark/mdurl.Parse
----------------------------------------------------------+-------------
```

The calls to `mdurl.Parse` are coming from `markdown.normalizeLink` and `markdown.normalizeLinkText`.

```
(pprof) list mdurl.Parse
Total: 145309950
ROUTINE ======================== gitlab.com/golang-commonmark/mdurl.Parse
in /usr/local/google/home/mpratt/go/pkg/mod/gitlab.com/golang-commonmark
/mdurl@v0.0.0-20191124015652-932350d1cb84/parse
.go
  -4974135   -4974135 (flat, cum)  3.42% of Total
         .          .     60:func Parse(rawurl string) (*URL, error) {
         .          .     61:   n, err := findScheme(rawurl)
         .          .     62:   if err != nil {
         .          .     63:           return nil, err
         .          .     64:   }
         .          .     65:
  -4974135   -4974135     66:   var url URL
         .          .     67:   rest := rawurl
         .          .     68:   hostless := false
         .          .     69:   if n > 0 {
         .          .     70:           url.RawScheme = rest[:n]
         .          .     71:           url.Scheme, rest =
strings.ToLower(rest[:n]), rest[n+1:]
```

Full source for these functions and callers can be found at:

[mdurl.Parse]

[markdown.normalizeLink]

[markdown.normalizeLinkText]

So what happened here? In a non-PGO build, `mdurl.Parse` is considered too large to be eligible for inlining. However, because our PGO profile indicated that the calls to this function were hot, the compiler did inline them. We can see this from the "(inline)" annotation in the profiles:

```
$ go tool pprof -top cpu.nopgo.pprof | grep mdurl.Parse
     0.36s   0.3% 63.76%      2.75s  2.32%  gitlab.com/golang-commonmark
/mdurl.Parse
$ go tool pprof -top cpu.withpgo.pprof | grep mdurl.Parse
     0.55s  0.48% 58.12%      2.03s  1.76%  gitlab.com/golang-commonmark
/mdurl.Parse (inline)
```

`mdurl.Parse` creates a `URL` as a local variable on line 66 (`var url URL`), and then returns a pointer to that variable on line 145 (`return &url, nil`). Normally this requires the variable to be allocated on the heap, as a reference to it lives beyond function return. However, once `mdurl.Parse` is inlined into `markdown.normalizeLink`, the compiler can observe that the variable does not escape `normalizeLink`, which allows the compiler to allocate it on the stack. `markdown.normalizeLinkText` is similar to `markdown.normalizeLink`.

The second largest reduction shown in the profile, from `mdurl.(*URL).String` is a similar case of eliminating an escape after inlining.

In these cases, we got improved performance through fewer heap allocations. Part of the power of PGO and compiler optimizations in general is that effects on allocations are not part of the compiler's PGO implementation at all. The only change that PGO made was to allow inlining of these hot function calls. All of the effects to escape analysis and heap allocation were standard optimizations that apply to any build. Improved escape behavior is a great downstream effect of inlining, but it is not the only effect. Many optimizations can take advantage of inlining. For example, constant propagation may be able to simplify the code in a function after inlining when some of the inputs are constants.

Devirtualization

In addition to inling, which we saw in the example above, PGO can also drive conditional devirtualization of interface calls.

Before getting to PGO-driven devirtualization, let's step back and define "devirtualization" in general. Suppose

you have code that looks like something like this:

```
f, _ := os.Open("foo.txt")
var r io.Reader = f
r.Read(b)
```

Here we have a call to the `io.Reader` interface method `Read`. Since interfaces can have multiple implementations, the compiler generates an *indirect* function call, meaning it looks up the correct method to call at run time from the type in the interface value. Indirect calls have a small additional runtime cost compared to direct calls, but more importantly they preclude some compiler optimizations. For example, the compiler can't perform escape analysis on an indirect call since it doesn't know the concrete method implementation.

But in the example above, we *do* know the concrete method implementation. It must be `os.(*File).Read`, since `*os.File` is the only type that could possibly be assigned to `r`. In this case, the compiler will perform *devirtualization*, where it replaces the indirect call to `io.Reader.Read` with a direct call to `os.(*File).Read`, thus allowing other optimizations.

(You are probably thinking "that code is useless, why would anyone write it that way?" This is a good point, but note that code like above could be the result of inlining. Suppose `f` is passed into a function that takes an `io.Reader` argument. Once the function is inlined, now the `io.Reader` becomes concrete.)

PGO-driven devirtualization extends this concept to situations where the concrete type is not statically known, but profiling can show that, for example, an `io.Reader.Read` call targets `os.(*File).Read` most of the time. In this case, PGO can replace `r.Read(b)` with something like:

```
if f, ok := r.(*os.File); ok {
    f.Read(b)
} else {
    r.Read(b)
}
```

That is, we add a runtime check for the concrete type that is most likely to appear, and if so use a concrete call, or otherwise fall back to the standard indirect call. The advantage here is that the common path (using `*os.File`) can be inlined and have additional optimizations applied, but we still maintain a fallback path because a profile is not a guarantee that this will always be the case.

In our analysis of the markdown server we didn't see PGO-driven devirtualization, but we also only looked at the top impacted areas. PGO (and most compiler optimizations) generally yield their benefit in the aggregate of very small improvements in lots of different places, so there is likely more happening than just what we looked at.

Inlining and devirtualization are the two PGO-driven optimizations available in Go 1.21, but as we've seen,

these often unlock additional optimizations. In addition, future versions of Go will continue to improve PGO with additional optimizations.