# From Slow to Go

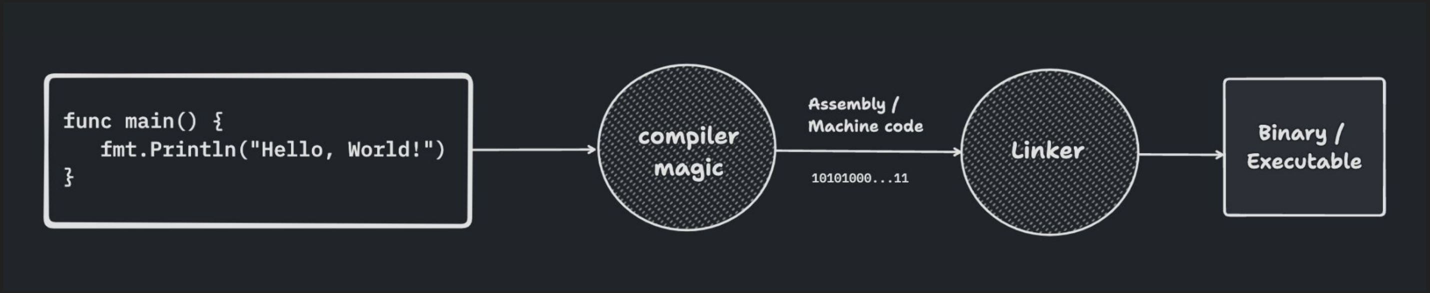
## Boosting your code with Profile-Guided Optimization

# $whoami

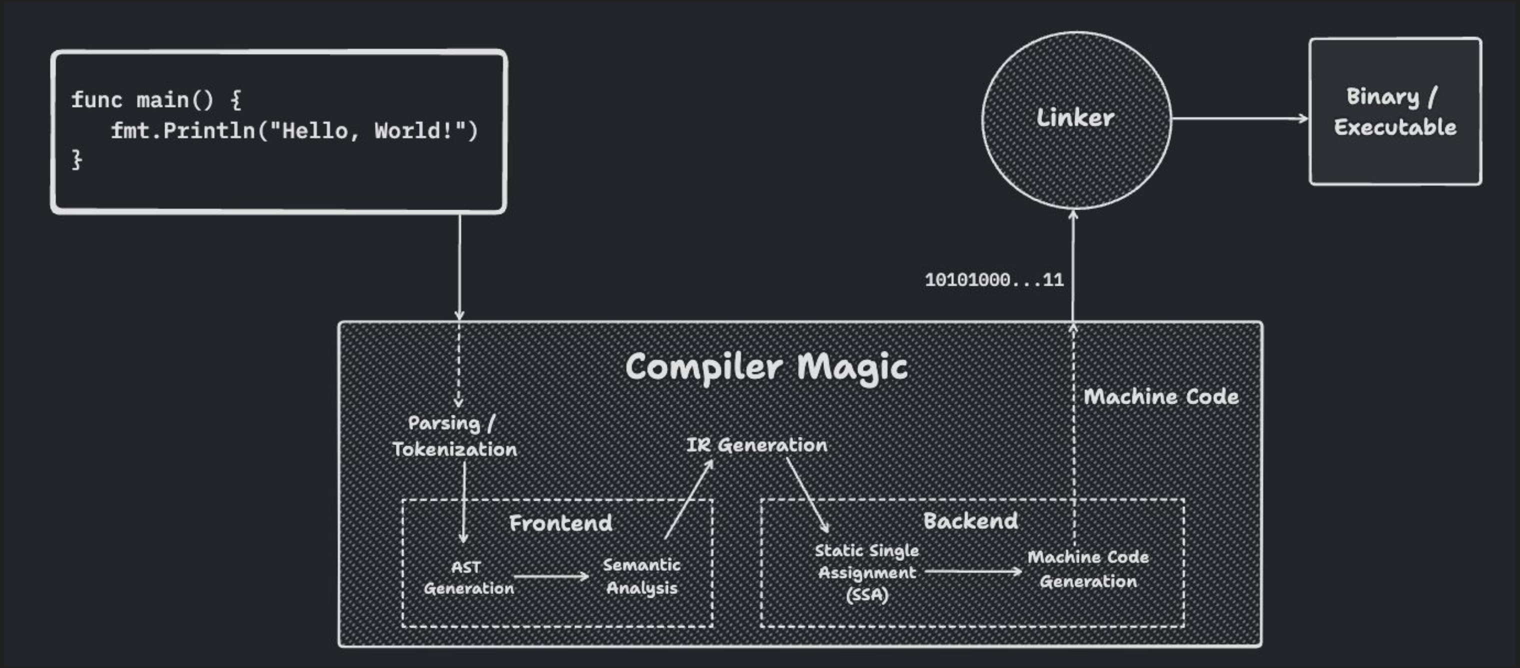
- Yashvardhan Kukreja (Yash)
- Software Engineer @ Red Hat
- Masters @ University of Waterloo
- Working on Openshift, Kubernetes and cloud-native stuff
- Free time? - Open source Dev, Running, Building

# Let’s talk about compilation

- Computer doesn’t understand Go.
- It just knows 0s and 1s.
- Compiler translates your Go code to 0s and 1s.



# The “magic” in Compiler Magic



# More Magic? Compiler Optimizations!

- Transform your code in a more optimized variant before translating it further for your computer
- Compile-time slowness -> Runtime performance (worth it!)
- Optimized?
  - Lower size of the executable
  - Lesser number of instructions and code jumps
  - Exploit the underlying hardware - SIMD, Branch prediction, etc.
  - Help writing cleaner code with zero-cost abstractions

# Some examples

- Pre-calculation of constants
- Loop unrolling
- Dead-store elimination
- … and countless other optimizations

Pre-calculation of Constants

```
a = 2 * 3 + 5  ───────→  a = 11
```

Loop Unrolling

```
for x:=0; x<y*2; x++ {            for x:=0; x<y*2; {
    bar(x);           ───────→        bar(x++)
}                                     bar(x++)
                                  }
```

Dead-store elimination

```
a = x+y+z
a = a+b      ───────→  a = 5*c
a = 5*c
```

# “Inlining” - Another interesting optimization

- Calling a function is slow
  - Pushing the parameters to the stack
  - Jumping to the function’s code
  - Returning to the original location
- Inlining to the rescue!
  - Take the code of the function and place it directly where it’s invoked
  - Eliminates the function call

**A very ideal case of inlining**

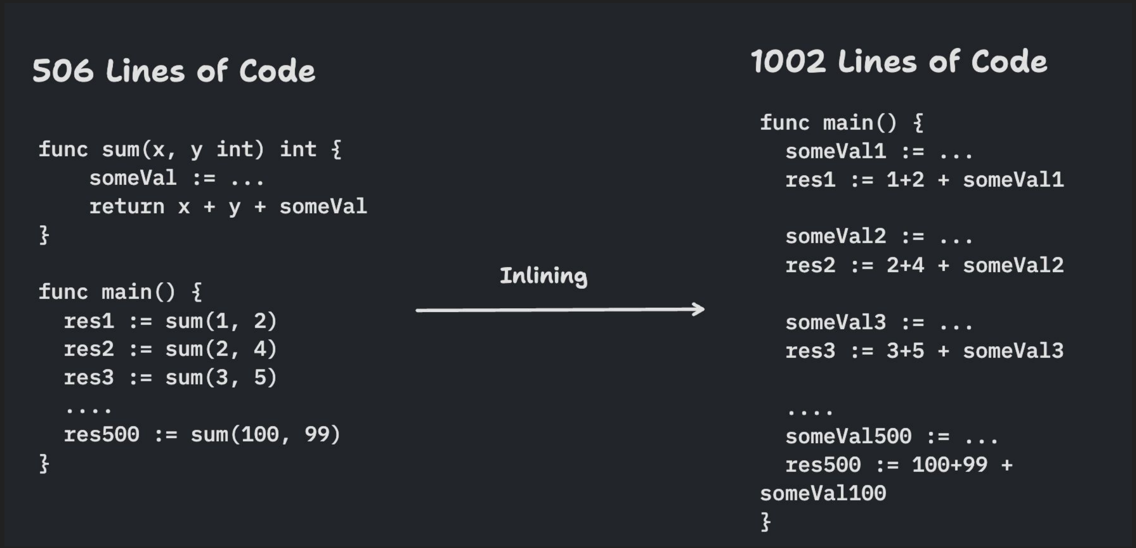
```go
func sum(x, y int) int {
  return x + y
}

func main() {
  res1 := sum(1, 2)
  res2 := sum(2, 4)
  fmt.Println(res1, res2)
}
```

Inlining →

```go
func main() {
  res1 := 1 + 2
  res2 := 2 + 4
  fmt.Println(res1, res2)
}
```

# But what if?

- Too many invocations
- Too much inlining
- Too many new lines of code
- A bloated binary
- Instruction Cache Misses
- Page Faults
- Thrashing (on light devices)
- Trade offs :(

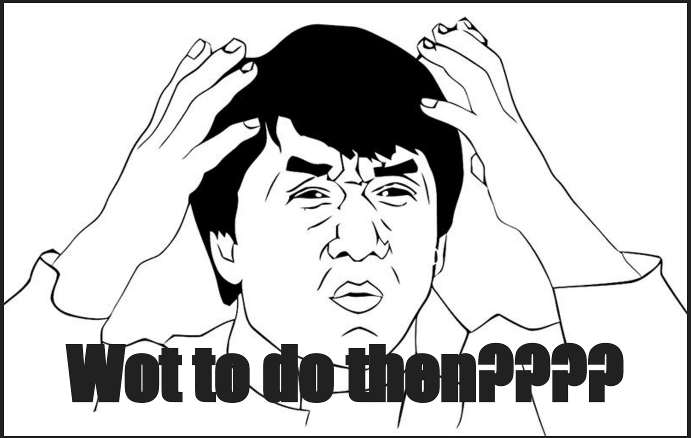


# Less Inlining

Bad runtime performance due to function call overhead

# More Inlining

Bad runtime performance due to bigger executable and page faults



# Just have the right amount of inlining

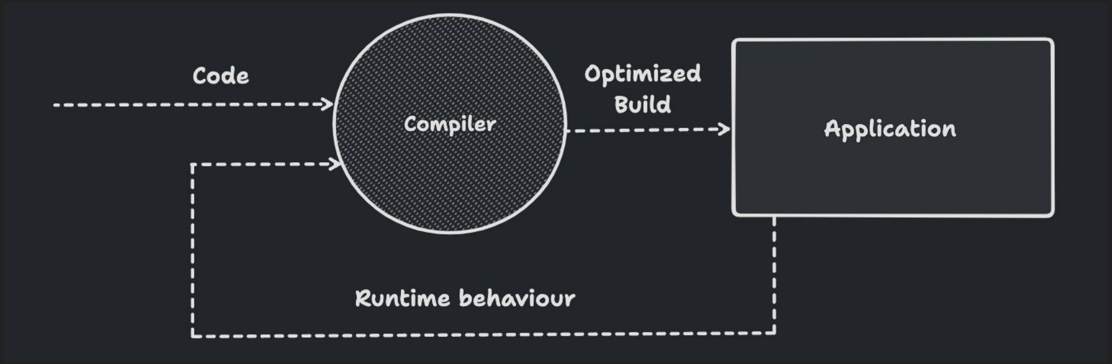
- Inline the “hot” functions to get
  - The functions which run a lot more frequently in runtime
  - Gives you the high performance of avoiding a bunch of functions calls in runtime.
- Not inline the “cold” functions to save on the binary size
  - The functions which run much less often to save on the binary size
  - Save you on the binary leading to lower page faults and better instruction cache hits.

# But Compilers don’t know a lot

- Compilers only see the code you wrote
- Not enough to tell how frequently a function would execute in runtime.

# Clearly, Compilers need more info!

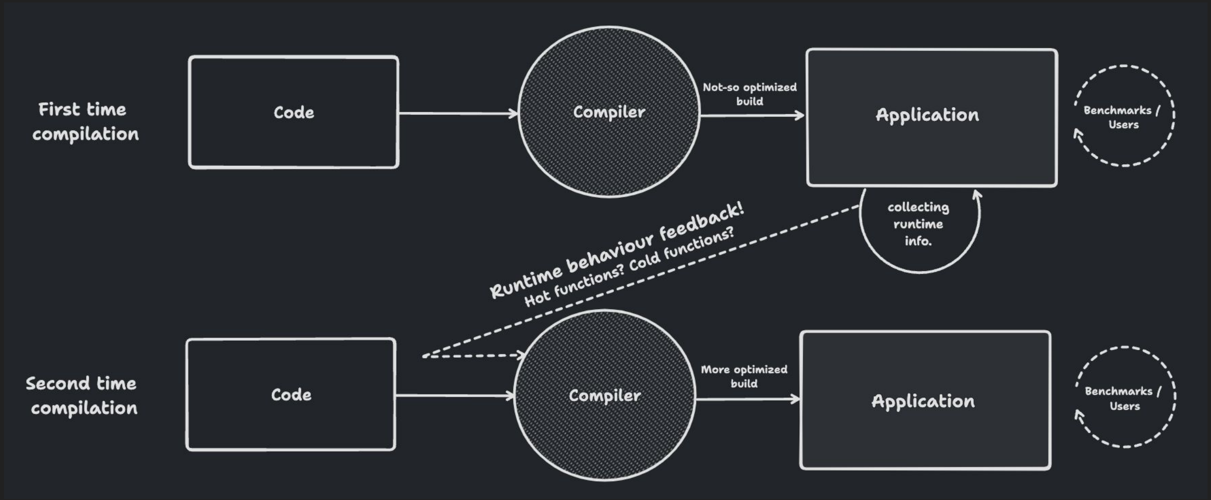
- What if compilers look at your application in runtime and learn?
- Or in other words,
  - Your application runs in runtime
  - You collect various number and metrics about its behaviour in runtime
  - Feed that information to the compiler next time you compile your code




Looks like a feedback-loop, doesn't it?

# Feedback-Driven Optimization (FDO)

- Teach compilers how and where to optimize your code on the basis of "feedback"
- Feedback?
  - Benchmarks
  - User Traffic



# Early days of FDO - Instrumentation-based

- The compiler introduces extra lines of code in between your code during compilation
    - Lines of code? - Start/Stop Timers, Call Counters, etc.
    - Track and instrument the behaviour of your code in runtime.

- A bunch of benchmarks are run against your application.
- A bunch of information gets instrumented.
- This information becomes the feedback for the next build by the compiler.

# Looks solid on paper, but is it really that good?

- Code is much more bloated with all those compiler-introduced instrumentations.
- The extra benchmarking step just makes the build process slower and boring.
- What if the benchmarks don’t resemble the reality of how your code runs in Production?
  - Leads to wrongfully assumed optimizations causing performance degradation instead.

# So what do we want? Let’s talk first principles

- Faster build times
- Realistic runtime data instead of benchmarks “pretending” to be real.
- Lighter executables with no extra lines of code for instrumentation.

# Easy enough

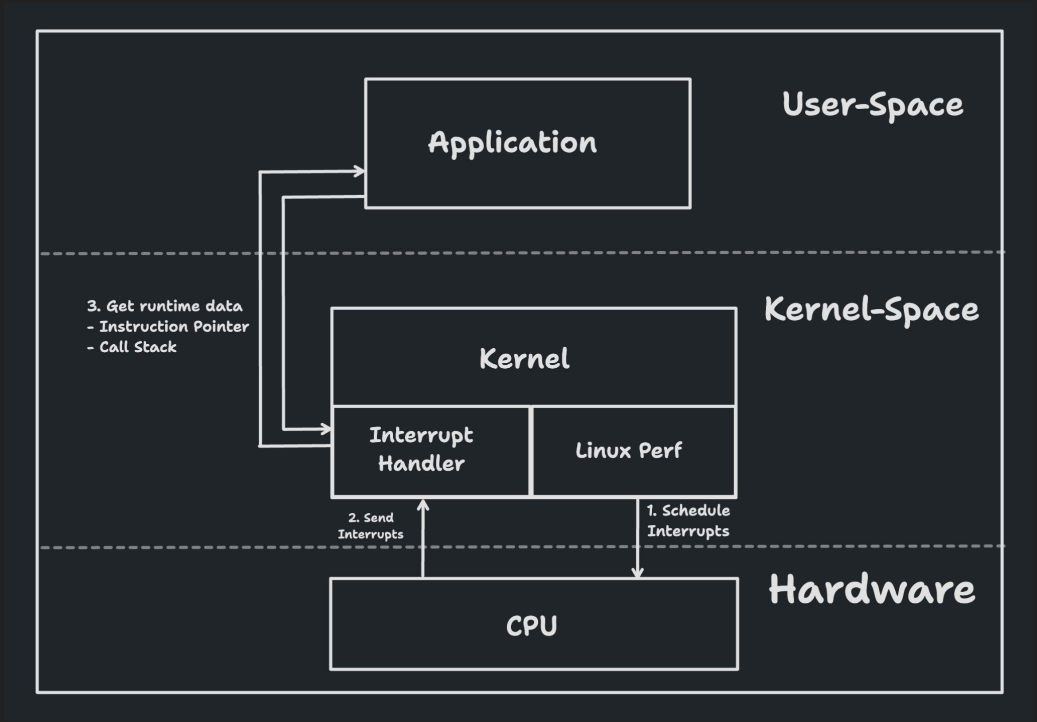
- Use actual behaviour of your code as the feedback to your compiler!
- Faster builds times
  - Saves you from running benchmarks during compilation.
- More realistic
  - No more pretentious benchmarks.

# Profiling!

- Tracks the runtime behaviour of your code.
- No need for those extra lines of code to be inserted during compilation.
- Sample-Based Profiling (Ackchuallllyy!!)

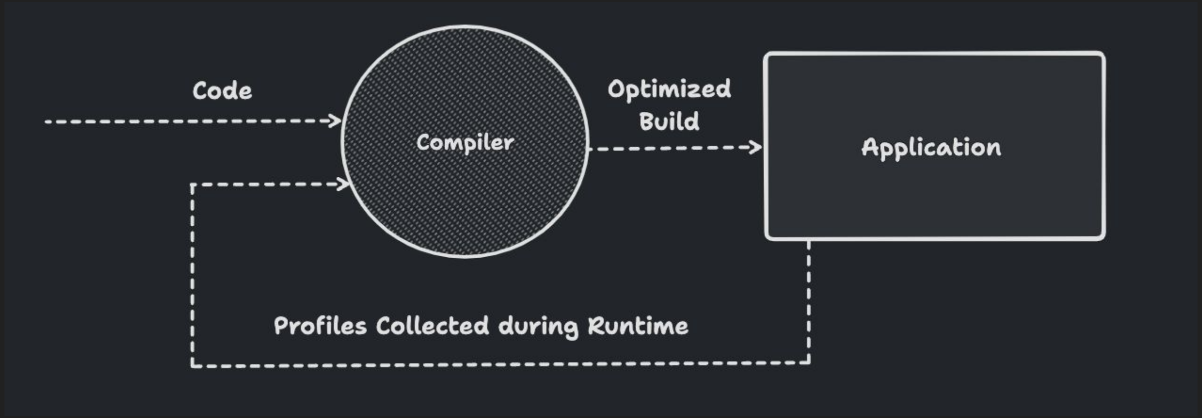
# How does it work then?

Kernel uses programmable events and interrupts to poke your application for runtime information.



# Enter Profile-Guided Optimization - PGO

As the name suggests,
Compiler “optimizations” which are “guided” by the “profiles” of your code collected during its runtime.



Talk is cheap, let’s walk in code

# A very simple server

- POST markdown files at /render
- Get a rendered markdown in response

```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) {
	src, err := io.ReadAll(r.Body)
	if err != nil {
		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 {
		http.Error(w, "Malformed markdown", http.StatusBadRequest)
		return
	}

	if _, err := io.Copy(w, &buf); err != nil {
		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 the code with “-m” gcflag to show escape analysis and inlining decisions

```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) {
	src, err := io.ReadAll(r.Body)
	if err != nil {
		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 {
		http.Error(w, "Malformed markdown", http.StatusBadRequest)
		return
	}

	if _, err := io.Copy(w, &buf); err != nil {
		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))
}
```

```
go build -gcflags -m .
```

```
ubuntu@ip-172-31-93-221:~/pgo$ go build -gcflags -m .
# pgo-stuff
./main.go:21:23: inlining call to markdown.XHTMLOutput
./main.go:22:23: inlining call to markdown.Typographer
./main.go:23:19: inlining call to markdown.Linkify
./main.go:24:18: inlining call to markdown.Tables
./main.go:33:22: inlining call to io.Copy
./main.go:40:17: inlining call to http.HandleFunc
./main.go:41:12: inlining call to log.Printf
./main.go:42:31: inlining call to http.ListenAndServe
./main.go:40:17: inlining call to http.(*ServeMux).HandleFunc
```

```go

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

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

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

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

All of these are getting inlined

# But if you notice carefully

```go
func render(w http.ResponseWriter, r *http.Request) {  1 usage
    src, err := io.ReadAll(r.Body)
    if err != nil {
        http.Error(w,  error: "Internal Server Error", http.StatusInternalServerError)
        return
    }

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

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

These didn't
get inlined

# Inlining could’ve been useful here

- The act of just calling a function itself tends to have an overhead
  - Setting up a new stack dedicated to the function’s scope
  - Returning back to the caller
  - Pass-by-value performance overhead.
- io.ReadAll() is getting called everytime render() gets called
- For every request, render() gets called showing some “hot”-ness.

# Let’s run and profile the program

```
ubuntu@ip-172-31-93-221:~/pgo$ go build -gcflags -m -o main.nopgo main.go
# command-line-arguments
./main.go:21:23: inlining call to markdown.XHTMLOutput
./main.go:22:23: inlining call to markdown.Typographer
./main.go:23:19: inlining call to markdown.Linkify
./main.go:24:18: inlining call to markdown.Tables
./main.go:33:22: inlining call to io.Copy
./main.go:40:17: inlining call to http.HandleFunc
./main.go:41:12: inlining call to log.Printf
./main.go:42:31: inlining call to http.ListenAndServe
./main.go:40:17: inlining call to http.(*ServeMux).HandleFunc
./main.go:13:13: leaking param: w
```

Building the server

```
ubuntu@ip-172-31-93-221:~/pgo$ ./main.nopgo
2024/04/14 17:24:56 Serving on port 8080...
```

Running the server

```
ubuntu@ip-172-31-93-221:~/pgo$ go run github.com/prattmic/markdown-pgo/load@latest
```

Executing the load

```
ubuntu@ip-172-31-93-221:~/pgo$ curl -o cpu.nopgo.pprof "http://localhost:8080/debug/pprof/profile?
seconds=30"
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 43423    0 43423    0     0   1440      0 --:--:--  0:00:30 --:--:-- 11119
```

Collecting the profiles against the load

```
cp cpu.nopgo.pprof default.pgo
```

# We have all these files now

```
ubuntu@ip-172-31-93-221:~/pgo$ ls -atrl
total 8508
-rw-rw-r--  1 ubuntu ubuntu     574 Apr 14 14:07 go.mod
-rw-rw-r--  1 ubuntu ubuntu    2128 Apr 14 14:07 go.sum
-rw-rw-r--  1 ubuntu ubuntu    1455 Apr 14 14:07 README.md
-rw-rw-r--  1 ubuntu ubuntu     860 Apr 14 17:05 main.go
drwxr-xr-x 10 ubuntu ubuntu    4096 Apr 14 17:05 ..
-rwxrwxr-x  1 ubuntu ubuntu 8596200 Apr 14 17:24 main.nopgo
-rw-rw-r--  1 ubuntu ubuntu   43423 Apr 14 17:26 cpu.nopgo.pprof
-rw-rw-r--  1 ubuntu ubuntu   43423 Apr 14 17:28 default.pgo
drwxrwxr-x  2 ubuntu ubuntu    4096 Apr 14 17:28 .
```

# Now, let's compile with pgo

```
ubuntu@ip-172-31-93-221:~/pgo$ go build -gcflags -m -pgo=auto -o main.withpgo main.go
# command-line-arguments
./main.go:14:24: inlining call to io.ReadAll
./main.go:21:23: inlining call to markdown.XHTMLOutput
./main.go:22:23: inlining call to markdown.Typographer
./main.go:23:19: inlining call to markdown.Linkify
./main.go:24:18: inlining call to markdown.Tables
./main.go:40:17: inlining call to http.HandleFunc
./main.go:41:12: inlining call to log.Printf
./main.go:42:31: inlining call to http.ListenAndServe
./main.go:40:17: inlining call to http.(*ServeMux).HandleFunc
```

The compiler decided to inline io.ReadAll

Probably some more internal inlining happened as well

```
ubuntu@ip-172-31-93-221:~/pgo$ ls -atrl | grep "main.*pgo"
-rwxrwxr-x  1 ubuntu ubuntu 8596200 Apr 14 17:24 main.nopgo
-rwxrwxr-x  1 ubuntu ubuntu 8781539 Apr 14 17:30 main.withpgo
```

Due to more inlining the execute withpgo is slightly bigger as well

# Let’s load test the old and new binaries

Old one

```
ubuntu@ip-172-31-93-221:~/pgo$ ./main.nopgo
2024/04/14 17:37:25 Serving on port 8080...
```

```
ubuntu@ip-172-31-93-221:~/pgo$ go test github.com/prattmic/markdown-pgo/load
-bench=. -count=40 -source $(pwd)/README.md > nopgo.txt
```

New one

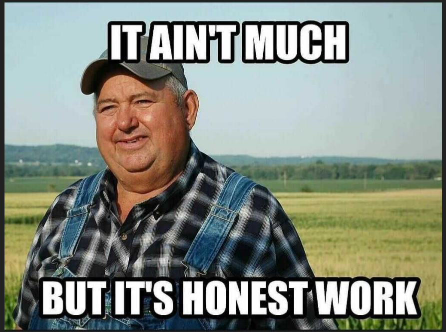
```
ubuntu@ip-172-31-93-221:~/pgo$ ./main.withpgo
2024/04/14 17:40:43 Serving on port 8080...
```

```
ubuntu@ip-172-31-93-221:~/pgo$ go test github.com/prattmic/markdown-pgo/load
-bench=. -count=40 -source $(pwd)/README.md > withpgo.txt
```

# Let’s compare the performances

```
ubuntu@ip-172-31-93-221:~/pgo$ benchstat nopgo.txt withpgo.txt
goos: linux
goarch: amd64
pkg: github.com/prattmic/markdown-pgo/load
cpu: Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz
       │  nopgo.txt  │               withpgo.txt                │
       │   sec/op    │    sec/op      vs base                   │
Load     306.4µ ± 1%   300.6µ ± 1%  -1.88% (p=0.000 n=40)
```

~2% increase in performance with no changes to the code



Let’s get our hands dirty?

# Conclusion

- We explored the process of compilation
- How compilation can be made more effective by feeding it runtime data.
- The way instrumentation-based FDO works.
- How Sampling-Profiles-based PGO works (more effectively).
- Got our hands dirty with playing with Profile-Guided Optimization.

To find these slides and the associated content

https://github.com/yashvardhan-kukreja/conf42-golang-pgo

# References - The real Gs

- Go dev blog on PGO
- An exhaustive list of compiler optimizations
- Example of the code referred from here
- Dive into Profiling with Go
- Understand PGO v/s FDO

# Let’s connect

- Twitter @yashkukreja98
- GitHub @yashvardhan-kukreja
- LinkedIn @yashvardhan-kukreja
- My blog @yash-kukreja-98.medium.com


Thanks for your time folks!

Feel free to raise any questions