

Fear is Necessary for Optimisation

A guide to writing fast, optimised Go programs

Briheet Siñgh Yadav

<https://github.com/briheet>

March 22, 2026

About Me

Briheet Singh Yadav

Software Engineer — Backend / Low Latency Systems

- ▶ Backend & low latency systems at work
- ▶ Gen-AI infra for media
- ▶ **Go** daily, **Rust** for fun, **C++** for competitive programming
- ▶ **Nix** for nix-darwin / home-manager setup. Deployments ;)

Find me

Discord: zenin108 — X: @brsy_10 — GitHub: briheet

Overview

- ▶ Theory + hands-on examples
- ▶ Tools, tricks, and ways to write **better** Go code
- ▶ We will **analyze** code and make it better step by step
- ▶ Find problems first, then solve them

Disclaimer

There are hundreds of optimisation tricks — we cover the important ones here. Pair this with a good understanding of Go.

Things That Bother Us

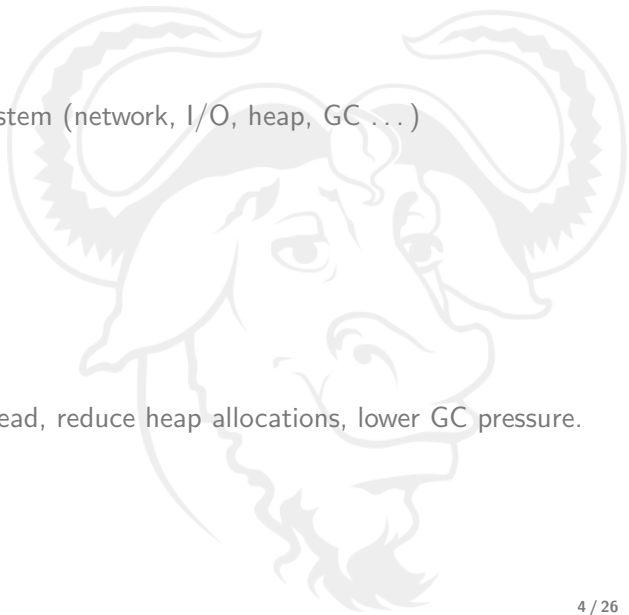
Why optimise at all?

Latency = delay in a system (network, I/O, heap, GC ...)

Things that bug us:

- ▶ I/O operations
- ▶ Synchronizations
- ▶ Heap allocations
- ▶ GC Pressure

Goal: Reduce I/O overhead, reduce heap allocations, lower GC pressure.



A Simple Go Program

Where does everything happen?

```
package main

import "fmt"

func main() {

    var answer int64

    numbers := []int64{1, 2,
        3, 4}
    answer = sum(numbers)

    fmt.Println(answer)
}

func sum(numbers []int64)
int64 {

    var sum int64
    for _, num := range
        numbers {
        sum += num
    }
}
```

```
1 package main
2 import "fmt"
3
4 func main() {
5
6     var answer int
7
8     numbers := []int{1, 2, 3, 4}
9     answer = sum(numbers)
10
11    fmt.Println(answer)
12 }
13
14 func sum(numbers []int) int {
15
16    var sum int
17    for _, num := range numbers {
18        sum += num
19    }
20
21    return sum
22 }
~
```


Go Assembler: MOVD

Go pseudo-instructions vs ARM64

```
MOVD 16(R28), R16 ; ldr x16, [x28,#16]
```

Go implements its own assembler pseudo-instructions for portability:

- ▶ 64-bit: `ldr`, `str`, `stur` ⇒ `MOVD`
- ▶ 32-bit: `str`, `stur`, `ldrsw` ⇒ `MOVW`
- ▶ 32-bit unsigned: `ldr` ⇒ `MOVWU`

See Rob Pike's talk: Design of the Go Assembler

MOV (bitmask immediate): Move (bitmask immediate): an alias of ORR (immediate).

MOV (inverted wide immediate): Move (inverted wide immediate): an alias of **MOVN**.

MOV (register): Move (register): an alias of ORR (shifted register).

MOV (to/from.SP): Move between register and stack pointer: an alias of ADD (immediate).

MOV (wide immediate): Move (wide immediate): an alias of **MOVZ**.

MOVK: Move wide with keep.

MOVN: Move wide with NOT.

MOVZ: Move wide with zero.

Inside asmout: MOVD → LDR

```
a.out.go 3803
anames.go 3804
anames7.go 3805
asm7.go 3806
asm_arm64_test.go 3807
asm_arm64_tests 3808
asm_test.go 3809
doc.go 3810
list7.go 3811
obj7.go 3812
specialoperand_string.go 3813
sysRegEnc.go 3814
3815
3816
3817
3818
3819
3820
```

```
o1 = c.olsr12u(p, c.opstr(p, p.As), v, rt, rf)
}
case 21: /* movt 0(R),R -> ldrt */
v := c.regoff(gp.From)
sz := int32(1 << uint(movesize(p.As)))
rt, rf := p.To.Reg, c.From.Reg
if rf == obj.REG_NONE {
rf = 0.param
}
if v < 0 || v&sz != 0 { /* unscaled 9-bit signed */
o1 = c.olsr9s(p, c.opldr(p, p.As), v, rf, rt)
} else {
v = int32(c.offsetshift(p, int64(v), int(o.a1)))
o1 = c.olsr12u(p, c.opldr(p, p.As), v, rf, rt)
}
}
```

Switch in asmout

```
a.out.go 7190 // www.llvm.org/
anames.go 7197 // scaled 12-bit unsigned immediate offset.
anames7.go 7198 // unscaled 9-bit signed immediate offset.
asm7.go 7199 // pre/post-indexed load.
asm_arm64_test.go 7200 // and the 12-bit and 9-bit are distinguished in olsr12u and olsr9s.
asm_arm64_test.a 7200 func (c *ctxt7) olsr12u(p *obj.Prog, a obj.As) uint32 {
asm_test.go 7201 switch a {
doc.go 7202 case AMOVQ:
list7.go 7203 return LDSTR(3, 0, 1) /* sim0<12 | Renc5 | Rt */
obj7.go 7204 case AMOVM:
specialoperand_string.go 7205 return LDSTR(2, 0, 2)
sysRegEnc.go 7206 return LDSTR(2, 0, 2)
7207
7208 case AMOVL:
7209 return LDSTR(2, 0, 1)
7210
7211 case AMOVB:
7212 return LDSTR(1, 0, 2)
7213
7214
```

Base opcodes from opldr

```
a.out.go 7246
anames.go 7247 c.ctx.DIagn"bad spstore %v/vw", a, pl
anames7.go 7248 return 0
asm7.go 7249 }
asm_arm64_test.go 7250
asm_arm64_tests 7251 /*
asm_test.go 7252 * load/store register (scaled 12-bit unsigned immediate) C3.3.13
doc.go 7253 * these produce 64-bit values (when there's an option)
list7.go 7254 */
obj7.go 7255 func (c *ctxt7) olsr12u(p *obj.Prog, o uint32, v int32, rs, rt int16) uint32 {
specialoperand_string.go 7256 if v < 0 || v >= (int32) 1
sysRegEnc.go 7257 c.ctx.DIagn"offset out of range: %v/vw", v, pl
7258 o |= uint32((v&0xfff) << 3)
7259 o |= uint32((v >= 0) << 5)
7260 o |= uint32((rt & 3) << 1)
7261 o |= 5 << 24
7262 return o
7263 }
7264
7265
```

olsr12u encodes the immediate and sets the unsigned-offset bit.

R28 = Current Goroutine

Go ABI on ARM64

- › add2lsw
- › api
- › arm
- › buildid
- › cgo
- › compile
- › internal
- › testdata
- › README.md
- › **abi-internal.md**
- › default.gpp
- › doc.go
- › main.go
- › profile.sh
- › script_test.go
- › testdata
- › cover
- › dot
- › dotpack
- › fix
- › go
- › gofmt
- › internal
- › archive

The `mt7` floating-point control word is not used by Go on arm64.

arm64 architecture

The arm64 architecture uses `R0` – `R15` for integer arguments and results.

It uses `F0` – `F15` for floating-point arguments and results.

Rationale: 16 integer registers and 16 floating-point registers are more than enough for passing arguments and results for practically all functions (see Appendix). While there are more registers available, using more registers provides little benefit. Additionally, it will add overhead on code paths where the number of arguments are not statically known (e.g. reflect call), and will consume more stack space when there is only limited stack space available to fit in the `nosplit` limit.

Registers `R16` and `R17` are permanent scratch registers. They are also used as scratch registers by the linker (Go linker and external linker) in trampolines.

Register `R18` is reserved and never used. It is reserved for the OS on some platforms (e.g. macOS).

Registers `R19` – `R25` are permanent scratch registers. In addition, `R27` is a permanent scratch register used by the assembler when expanding instructions.

Floating-point registers `F16` – `F31` are also permanent scratch registers.

Special-purpose registers are as follows:

Register	Call meaning	Return meaning	Body meaning
<code>RSP</code>	Stack pointer	Same	Same
<code>R30</code>	Link register	Same	Scratch (non-leaf functions)
<code>R29</code>	Frame pointer	Same	Same
<code>R28</code>	Current goroutine	Same	Same
<code>R27</code>	Scratch	Scratch	Scratch
<code>R26</code>	Closure context pointer	Scratch	Scratch
<code>R18</code>	Reserved (not used)	Same	Same
<code>ZR</code>	Zero value	Same	Same

Rationale: These register meanings are compatible with Go's stack-based calling convention.

```
MOVD 16(R28), R16 ; load g.stackguard0
CMP R16, RSP ; compare with stack pointer
BLS 36(PC) ; if too close, grow stack
```

R28 is reserved by the Go compiler as the current goroutine pointer (`g`).

Stack Guard & runtime.morestack

```
runtime.morestack_addr$SB
file: /nix/store/5141puzg5cvh6b8ndh9dgmgh-go-1.25.7/share/go/src/runtime/stack_arm64.s

MOVW RSP, R28
MOVW PC, R28
JMP runtime.morestack_addr$SB

runtime.morestack$SB
402 // the caller doesn't save LR on stack but passes it on s
403 // (Make it SPWRITE to stop unwinding. One issue 54532)
404 MOVW RSP, R28
405
406 MOVW SP, R28
407 // runtime.morestack$SB
408
409 // sp$Args stores return values from registers to a *interface{}.RegArgs in R28.
410 TEXT "sp$Args$SB",NOPLIT,30-0
```

Flow:

1. R28 = current goroutine (g)
2. MOVW R28, R16 loads g.stackguard0
3. CMP R16, SP — is the stack pointer too close to the guard?

The Goroutine Struct (g)

```
471 type g struct {
472     // Stack parameters.
473     // stack describes the actual stack memory: [stack.lo, stack.hi].
474     // stackguard0 is the stack pointer compared in the Go stack growth prologue.
475     // It is stack.lo+StackGuard normally, but can be StackPreempt to trigger a preemption.
476     // stackguard1 is the stack pointer compared in the //go:systemstack stack growth prologue.
477     // It is stack.lo+StackGuard on g0 and gsignal stacks.
478     // It is ~0 on other goroutine stacks, to trigger a call to morestackc (and crash).
479     stack    stack // offset known to runtime/cgo
480     stackguard0 uintptr // offset known to cmd/internal/obj/*
481     stackguard1 uintptr // offset known to cmd/internal/obj/*
482
483     _panic   *_panic // innermost panic
484     _defer   *_defer // innermost defer
485     m        *m      // current m
486     sched    gobuf
487     syscallsp uintptr // if status==Gsyscall, syscallsp = sched.sp to use during gc
488     syscallpc uintptr // if status==Gsyscall, syscallpc = sched.pc to use during gc
489     syscallbp uintptr // if status==Gsyscall, syscallbp = sched.bp to use in fpTraceback

```

```
455 }
456
457 // Stack describes a Go execution stack.
458 // The bounds of the stack are exactly [lo, hi),
459 // with no implicit data structures on either side.
460 type stack struct {
461     lo uintptr
462     hi uintptr
463 }
464
```

Offset +0, +8, +16 ...

+16 on R28 gives stackguard0.

Stack Frame & Slice Building

```
MOVD .W R30, -112(RSP) ; save return address
MOVD R29, -8(RSP) ; save frame pointer
SUB $8, RSP, R29 ; set up frame pointer

ORR $1, ZR, R1 ; R1 = 1
ORR $2, ZR, R2 ; R2 = 2
STP (R1, R2), 56(RSP) ; store [1,2] at SP+56
ORR $3, ZR, R2 ; R2 = 3
ORR $4, ZR, R3 ; R3 = 4
STP (R2, R3), 72(RSP) ; store [3,4] at SP+72
```

Slice data lives on the stack: offsets 56–88 = 32 bytes = 4×8 -byte ints.

The Inlined Sum Loop

```
MOVD ZR, R0          ; i = 0
MOVD ZR, R2          ; sum = 0
JMP 5(PC)            ; jump to
                    condition

ADD $56, RSP, R3     ; base addr
MOVD (R3)(R0<<3), R4 ; load elem
ADD R2, R4, R2       ; sum += elem
ADD $1, R0, R0       ; i++
CMP $4, R0           ; i < 4?
BLT -5(PC)          ; loop back
```

```
sum := 0
for i := 0; i < 4; i++ {
    sum += numbers[i]
}
```

$R0 \ll 3 = \text{index} \times 8$
(each int64 is 8 bytes)

Jump-before-body pattern prevents executing the body before the first check.

The fmt.Println Trap: Interface Boxing

```
ADD $1, R0, R0
CMP $4, R0
BLT -5(PC)
STP (ZR, ZR), 88(RSP)
MOVD R2, R0
CALL runtime.convT64(SB)
ADRP 217088(PC), R1
ADD $1216, R1, R1
STP (R1, R0), 88(RSP)
ADRP 905216(PC), R27
```

```
MOVD 16828(R2), R38
CMP R1R, R3P
R1, 21(PC)
MOVD W R1R, -48(RSP)
MOVD R2R, -32(RSP)
SUB $8, RSP, R29
CMP $216, R0
PCS 5(PC)
ADRP 38204(PC), R1
ADDP $784, R1, R1
ADDP 908-<3, R1, R1
JMP 38(PC)

MOVD R0, 568(RSP)
ADRP 112648(PC), R27
MOVD 126(R27), R1
ORR $8, ZR, R0
MOVD ZR, R2
CALL runtime.convT64(SB)
MOVD 568(RSP), R1
MOVD R1, R0
MOVD R0, R1

MOVD R1, R0
MOVD -8(RSP), R29
MOVD $P-48(RSP), R30
RET
```

```
runtime.convT64(SB)
297 // See go.dev/issue748.
298 //
299 //go:linkname convT64
300 func convT64(val any) (x unsafe.Pointer) {
301     if val == nil {
302         x = unsafe.Pointer(&runtime.convT64Nil)
303     } else {
304         x = runtime.convT64(val)
305     }
306     return
307 }
308 //
309 // convT64 should be an internal detail.
310 }
```

`fmt.Println(answer)` causes
`runtime.convT64`

The `convT64` implementation —
allocates at runtime!

Why?

`fmt.Println(a ...any)` takes `any` (interface) — so the value gets boxed, causing a heap allocation.

Part 2: CPU Profiling

Measure, don't guess

CPU Profiling

Measuring where your program spends CPU time.
Records which functions consume the most cycles.

Tools

```
go test -bench, pprof, benchstat, lensm
```

The Victim Program

```
func calculateResult(num int) string {  
    var result string  
    for i := range num {  
        result += fmt.Sprintf("item-%d,", i)  
    }  
    return result  
}
```

```
$ go run main.go  
elapsed: 2.649227084s  
length: 100101
```

2.6 seconds for string building. Yikes.

Writing Tests & Benchmarks

```
> go test -v ./...
=== RUN   TestCalculateResult
=== RUN   TestCalculateResult/basic_test
--- PASS: TestCalculateResult (0.00s)
    --- PASS: TestCalculateResult/basic_test (0.00s)
PASS
ok      cpu      0.865s
> cat helper_test.go
package main

import (
    "testing"
)

func TestCalculateResult(t *testing.T) {
    t.Run("basic test", func(t *testing.T) {
        num := 2
        got := calculateResult(num)
        want := "item-0,item-1,"
        if got != want {
            t.Errorf("got %s, want %s", got, want)
        }
    })
}
> cat helper.go
package main

import (
    "fmt"
)

func calculateResult(num int) string {
    var result string
    for i := range num {
        result += fmt.Sprintf("item-%d,", i)
    }
    return result
}
>
```

Test passes

```
26 package main
25
24 import (
23     "testing"
22 )
21
20 func TestCalculateResult(t *testing.T) {
19     t.Run("basic test", func(t *testing.T) {
18         num := 2
17         got := calculateResult(num)
16         want := "item-0,item-1,"
15         if got != want {
14             t.Errorf("got %s, want %s", got, want)
13         }
12     })
11 }
10
9 }
8 }
7
6 func BenchmarkCalculateResult(b *testing.B) {
5     num := 100_100
4     for b.Loop() {
3         calculateResult(num)
2     }
1 }
27 }
~
```

Benchmark setup

Always write tests first so optimisations don't break correctness.

Benchmarking Flags

```
go test -bench=. -benchmem -benchtime=3s \  
-count=10 -cpuprofile=cpu.pprof \  
-memprofile=mem.pprof > benchmark.txt
```

- ▶ `-bench=.` — run all benchmarks
- ▶ `-benchmem` — include memory allocation stats
- ▶ `-benchtime=3s` — each benchmark runs for 3 seconds
- ▶ `-count=10` — repeat 10 times (profilers are non-deterministic)
- ▶ `-cpuprofile` / `-memprofile` — save profiles for pprof

Baseline: The Ugly Numbers

BenchmarkCalculateResult -12 2 2545ms 54GB/op 329k allocs/op

```
> go tool pprof cpu.pprof
File: cpu.test
Type: cpu
Time: 2026-03-22 14:19:45 IST
Duration: 47.64s, Total samples = 76.73s (161.07%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top5
Showing nodes accounting for 58.17s, 75.81% of 76.73s total
Dropped 176 nodes (cum <= 0.38s)
Showing top 5 nodes out of 88
      flat flat% sum% cum cum%
 23.63s 30.80% 30.80% 23.63s 30.80% runtime.madvise
 12.11s 15.78% 46.58% 12.11s 15.78% runtime.pthread_cond_wait
 10.61s 13.83% 60.41% 10.61s 13.83% runtime.pthread_kill
   6.29s  8.20% 68.60%   6.29s  8.20% runtime.pthread_cond_signal
   5.53s  7.21% 75.81%   5.53s  7.21% runtime.kevent
(pprof) list calculateResult
Total: 76.73s
ROUTINE ======  cpu.calculateResult in /Users/briheet/code/go_talk/cpu/helper.go
      0      6.14s (flat, cum)  8.00% of Total
      .      .      7:func calculateResult(num int) string {
      .      .      8:
      .      .      9:   var result string
      .      .     10:   for i := range num {
      .      6.14s     11:       result += fmt.Sprintf("item-%d,", i)
      .      .      12:   }
      .      .      13:
      .      .      14:   return result
      .      .      15:}
(pprof) █
```

pprof shows 6 seconds on the result line — and that's only 8% of the program!

Profiling Reveals: Memory is the Bottleneck

```
(pprof) list calculateResult
Total: 912.98GB
  912.70GB  912.97GB  11: result += fmt.Sprintf("item-%d,", i)
```

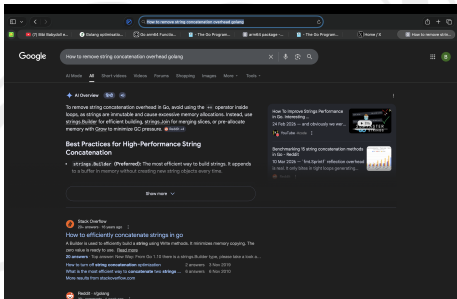
`fmt.Sprintf` itself is only 170ms — the real problem is **string concatenation**:

1. `runtime.convT64` — interface boxing for ...any
2. `runtime.concatstring2` — the `result +=` copies the entire string each time $\Rightarrow O(n^2)$

Fix #1: strings.Builder

```
func calculateResult(num int)
string {
var sb strings.Builder
for i := range num {
sb.WriteString(
fmt.Sprintf("item-%d
", i))
}
return sb.String()
}
```

```
$ go run .
elapsed: 11.362125ms
```



benchstat: v0 vs v1

Time: 2545ms → 4.5ms (-99.82%)

Memory: 52 GB → 7 MB (-99.99%)

Allocs: 330k → 200k (-39%)

Fix #2: Drop fmt.Sprintf

```
var itemBytes = []byte{0x69, 0x74, 0x65, 0x6d, 0x2d} // "item-"

func calculateResult(num int) string {
    var sb strings.Builder
    for i := range num {
        sb.Write(itemBytes)
        sb.WriteString(strconv.Itoa(i))
        sb.WriteByte(',')
    }
    return sb.String()
}
```

benchstat: v1 vs v2

Time: 4.5ms → 1.6ms (-64%)

Memory: 7.3 MB → 5.5 MB (-25%)

Allocs: 200k → 100k (-50%)

Fix #3: sync.Pool + Byte Buffer

```
var bufPool = sync.Pool{
    New: func() any {
        b := make([]byte, 0, 2*1024*1024)
        return &b
    },
}

func calculateResult(num int) string {
    bp := bufPool.Get().(*[]byte)
    buf := (*bp)[:0]
    var intBuf [20]byte
    for i := range num {
        buf = append(buf, itemBytes...)
        b := strconv.AppendInt(intBuf[:0], int64(i), 10)
        buf = append(buf, b...)
        buf = append(buf, commaByte)
    }
    result := string(buf)
    *bp = buf; bufPool.Put(bp)
    return result
}
```

Final Results: 3,077x Faster

```
$ benchstat benchmark.txt benchmark3.txt
```

```
CalculateResult-12  2545ms -> 0.83ms      -99.97%  
                   54 GB  -> 1 MB        -100.00%  
                   330k   -> 1 alloc   -100.00%
```

Version	Time	Memory	Allocs
v0: naive +=	2,545 ms	50.7 GB	329,600
v1: strings.Builder	4.5 ms	7 MB	200,000
v2: strconv + bytes	1.6 ms	5.5 MB	100,000
v3: sync.Pool	0.83 ms	1 MB	1

Key Takeaways

1. **Understand what the compiler does** — read the assembly
2. **Profile before optimising** — pprof, benchstat
3. **Avoid $O(n^2)$ patterns** — string concat, repeated copies
4. **Reduce heap allocations** — interface boxing, escape analysis
5. **Keep functions small & inlinable** — cost < 80
6. **Use the right tool** — strings.Builder, strconv.AppendInt, sync.Pool
7. **Always have tests** — optimise with confidence

Code

https://github.com/briheet/go_elixir_talk



Thank You!

Questions?

<https://github.com/briheet>