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:

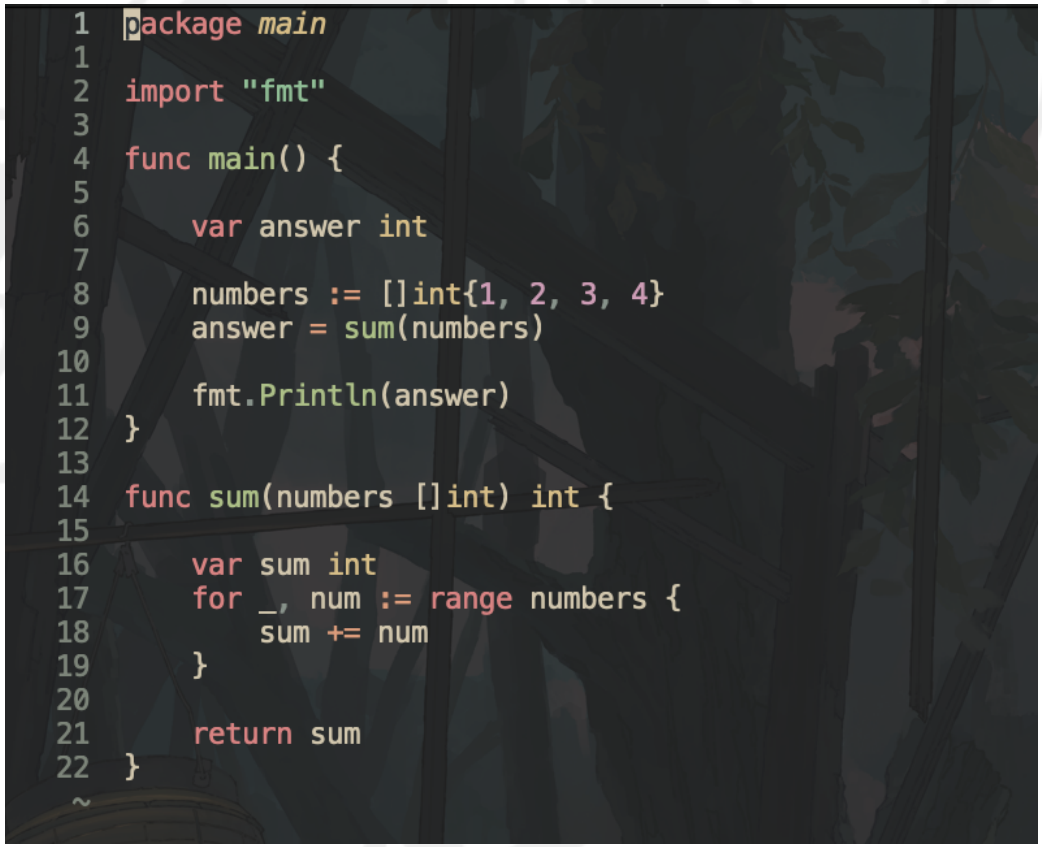
- I/O operations
- Synchronizations
- Heap allocations
- GC Pressure

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

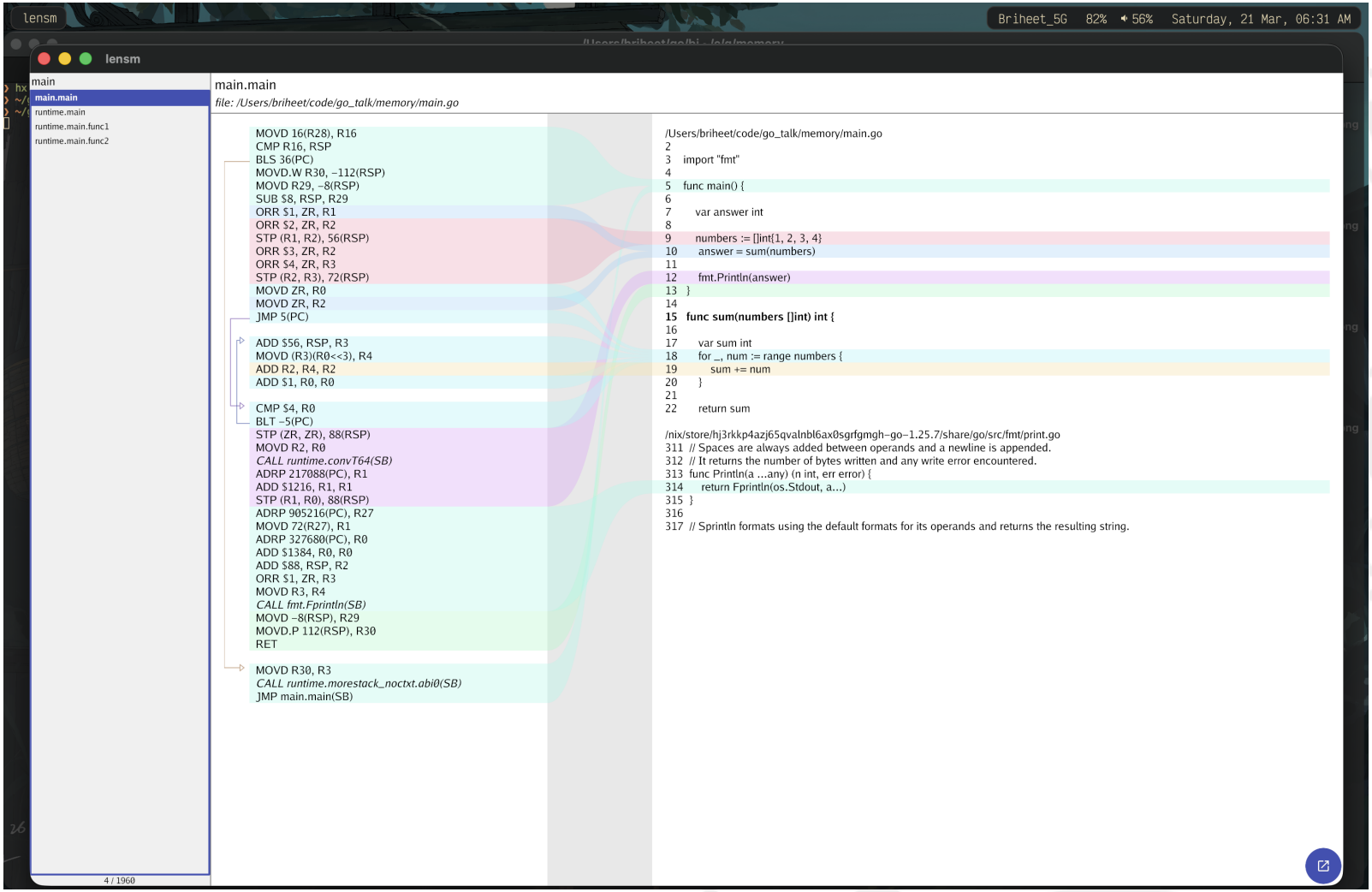
A Simple Go Program

Where does everything happen?

```go
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
    }
```



Reading Assembly with lensm



Our simple program does *a lot* under the hood.
Tool: `lensm` — a GUI to read Go assembly.

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` $\Rightarrow$ `MOVD`
- 32-bit: `str`, `stur`, `ldrsw` $\Rightarrow$ `MOVW`
- 32-bit unsigned: `ldr` $\Rightarrow$ `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).

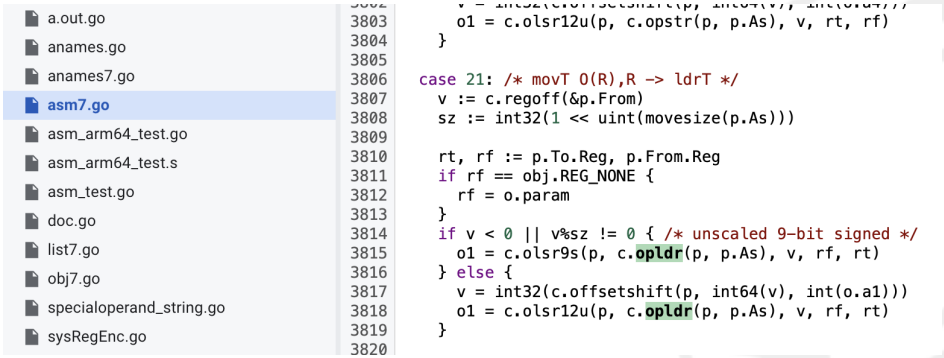
MOV (wide immediate): Move (wide immediate): an alias of MOVZ.

MOVK: Move wide with keep.

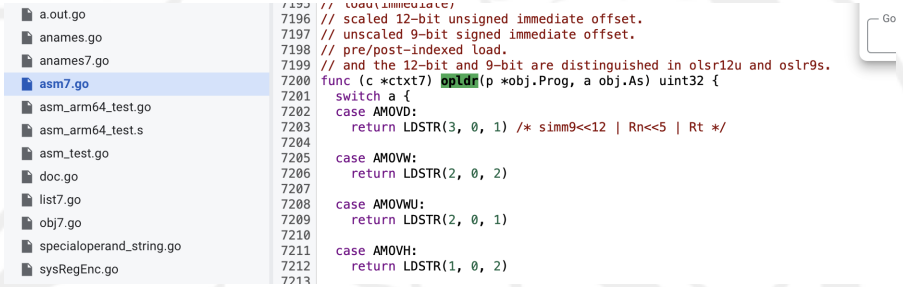
MOVN: Move wide with NOT.

MOVZ: Move wide with zero.

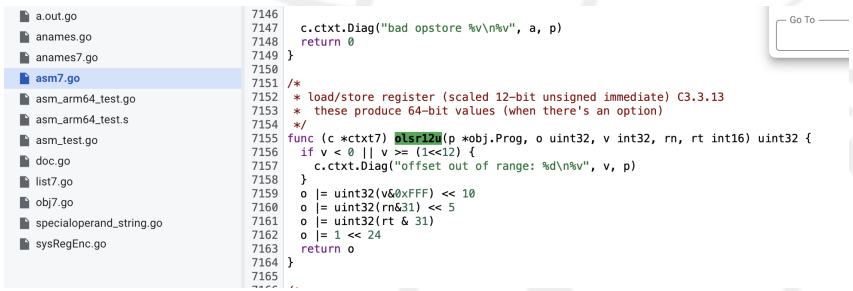
Inside asmout: MOVD → LDR



Switch in `asmout`



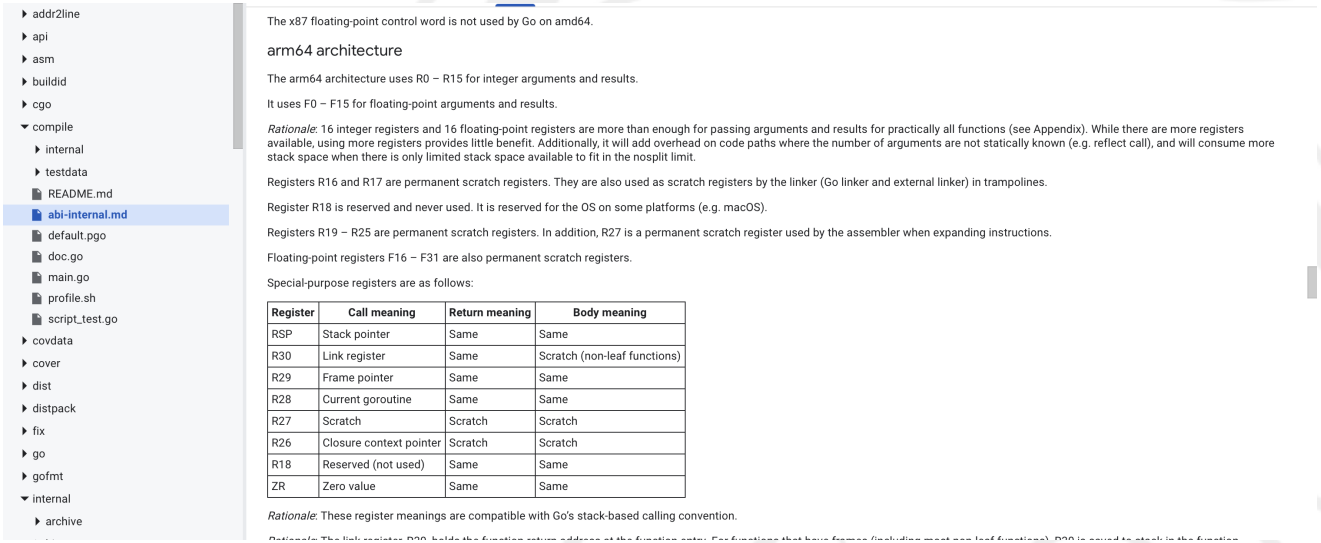
Base opcodes from `opldr`



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

R28 = Current Goroutine

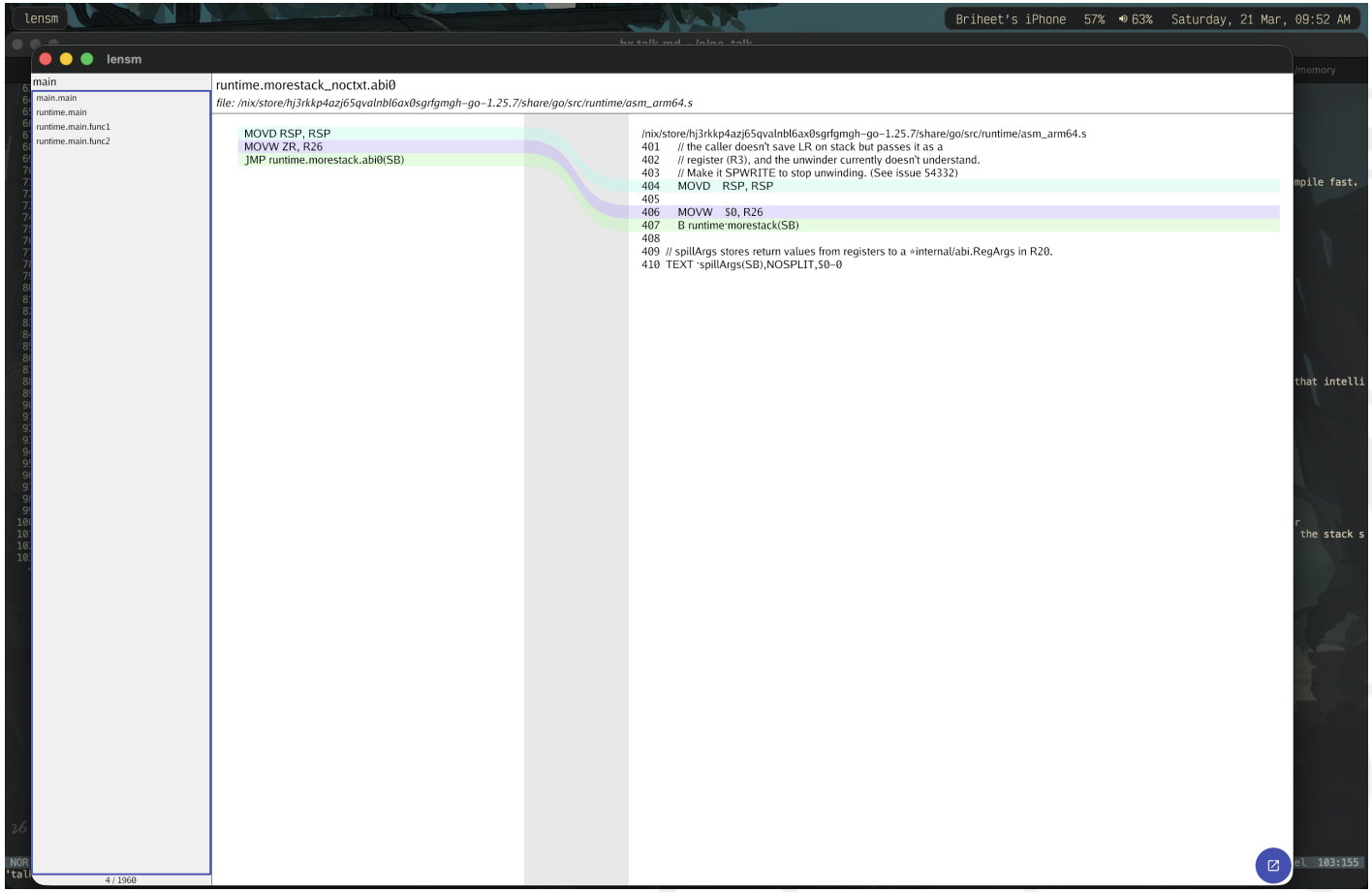
Go ABI on ARM64




```
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



Flow:

1. R28 = current goroutine (g)
2. `MOVD 16(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<<3 = index × 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
```

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

```
MOVD 16(R28), R16
CMP R16, RSP
BLS 23(PC)
MOVD.W R30, -48(RSP)
MOVD R29, -8(RSP)
SUB $8, RSP, R29
CMP $256, R0
BCS 5(PC)
ADRP 385024(PC), R1
ADD $2784, R1, R1
ADD R0<<3, R1, R1
JMP 10(PC)

MOVD R0, 56(RSP)
ADRP 1126400(PC), R27
MOVD 128(R27), R1
ORR $8, ZR, R0
MOVD ZR, R2
CALL runtime.mallocgc(SB)
MOVD 56(RSP), R3
MOVD R3, (R0)
MOVD R0, R1

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

```
/nix/store/hj3rkkp4azj65qvalnbl6ax0sgrfgmgh-go-1.25.7/share/go/src/runtime/iface.go
397 // See go.dev/issue/67401.
398 //
399 //go:linkname convT64
400 func convT64(val uint64) (x unsafe.Pointer) {
401     if val < uint64(len(staticuint64s)) {
402         x = unsafe.Pointer(&staticuint64s[val])
403     } else {
404         x = mallocgc(8, uint64Type, false)
405         *(*uint64)(x) = val
406     }
407     return
408 }
409
410 // convTstring should be an internal detail,
```

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

```go
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
18      t.Run("basic test", func(t *testing.T) {
17
16          num := 2
15          got := calculateResult(num)
14          want := "item-0,item-1,"
13
12          if got != want {
11              t.Errorf("got %s, want %s", got, want)
10          }
 9      })
 8  }
 7
 6  func BenchmarkCalculateResult(b *testing.B) {
 5
 4      num := 100_100
 3      for b.Loop() {
 2          calculateResult(num)
 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)
```

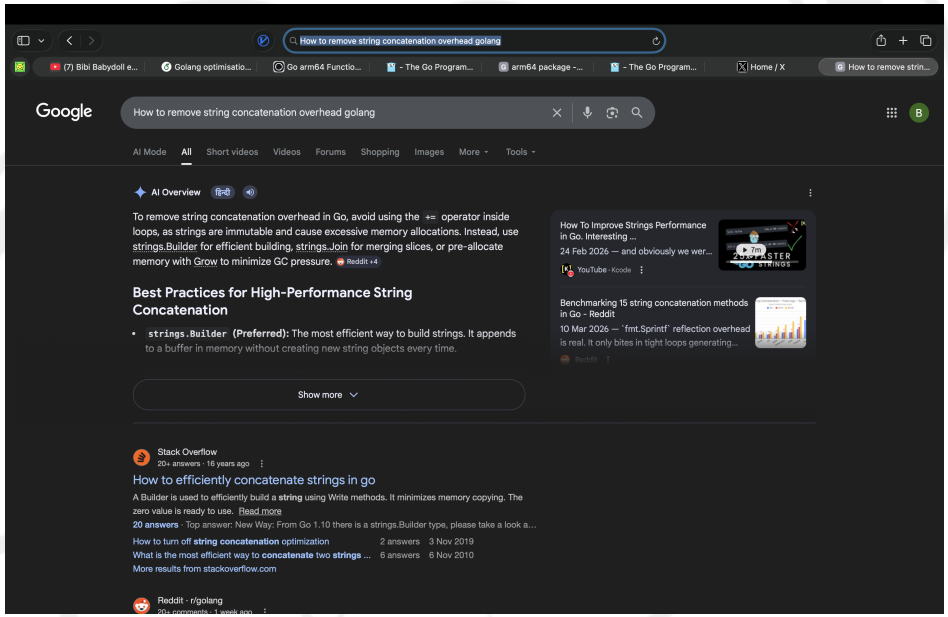
`fmt.Sprintf` itself is only 170ms — the real problem is string concatenation:

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

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

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



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

```go
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

```go
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