Common pitfalls in Go benchmarking

Go programmers have the good fortune of excellent testing and benchmarking tooling built into the standard library - in the `testing` package. However, *benchmarking is hard*. This isn't Go specific; it's just one of those things experienced developers learn over time.

This post lists some common benchmarking pitfalls Go programmers run into. It assumes basic familiarity with writing Go benchmarks; consult the testing package documentation if needed. While these pitfalls are presented in Go, they exist in any programming language or environment, so the lessons learned here are widely applicable.

Benchmarking the wrong thing

Let's say we want to benchmark the new sorting functionality available in the `slices` package starting with Go 1.21 [1]. Consider the following benchmark:

```
const N = 100_000

func BenchmarkSortIntsWrong(b *testing.B) {
  ints := makeRandomInts(N)
  b.ResetTimer()

  for i := 0; i < b.N; i++ {
    slices.Sort(ints)
  }
}

func makeRandomInts(n int) []int {
  ints := make([]int, n)
  for i := 0; i < n; i++ {
    ints[i] = rand.Intn(n)
  }
  return ints
}
```

Why is this benchmark wrong? Because it's not really testing what you think it's testing. `slices.Sort` sorts a slice *in-place*. After the first iteration, the `ints` slice is already sorted, so all subsequent iterations "sort" a sorted slice. This is not what we want to measure [2]. The right measurement would create a new random slice

for each iteration:

```
func BenchmarkSortInts(b *testing.B) {
  for i := 0; i < b.N; i++ {
    b.StopTimer()
    ints := makeRandomInts(N)
    b.StartTimer()
    slices.Sort(ints)
  }
}
```

On my machine, the second benchmark is almost 100x slower; this makes sense, because it actually sorts something on each iteration.

Forgetting to reset the timer

Note how the benchmarks mentioned above are careful to reset or stop/start the benchmarking timer around certain operations. This is on purpose, because the benchmark scaffold measures the run-time of the entire `Benchmark*` function, executing it many times and dividing the total execution time by `b.N` (much more details on this process later in the post).

Try it! Remove the `b.StopTimer()` and `b.StartTimer()` calls from the benchmarking loop in `BenchmarkSortInts` and compare the results - you'll notice the benchmark now reports noticeably slower per-op execution time because creating the slices of random integers also happens `b.N` times.

This can throw results off significantly in some cases. Even if we use the same technique to benchmark two approaches, the errors don't necessarily cancel out. Due to Amdahl's law we can easily report the wrong speed-up when not isolating the computation we care about.

Fooled by the compiler

The trickiest benchmarking pitfall is dealing with compiler optimizations thwarting our results. The Go compiler doesn't give `Benchmark*` functions any preferential treatment and will try to optimize them and their contents just as it would any other Go code [3]. This produces wrong results and it's not even easy to know they are wrong! Consider this benchmark:

```
func isCond(b byte) bool {
  if b%3 == 1 && b%7 == 2 && b%17 == 11 && b%31 == 9 {
    return true
  }
  return false
```

```
}


func BenchmarkIsCondWrong(b *testing.B) {
  for i := 0; i < b.N; i++ {
    isCond(201)
  }
}
```

We're trying to benchmark the `isCond` function, but the compiler is fooling us (at least in recent Go versions):

```
BenchmarkIsCondWrong-8        1000000000     0.2401 ns/op
```

Sub-nanosecond operation times are always a bit suspect, although not entirely out of the question given the simplicity of the `isCond` function. There are two serious errors here, however:

We use a constant input to `isCond`; since `isCond` is a simple function that is likely to be inlined into the benchmark function, the compiler can theoretically constant-propagate its input through its contents and replace the code with the compile-time computed answer.

Even if we used a non-constant for the input, the result of `isCond` is unused, so the compiler may optimize it away.

And in fact, in this example the contents of the benchmark loop are optimized away entirely; looking at the disassembly, we see this:

```
    JMP     BenchmarkIsCondWrong_pc7
BenchmarkIsCondWrong_pc4:
    INCQ    CX
BenchmarkIsCondWrong_pc7:
    CMPQ    416(AX), CX
    JGT     BenchmarkIsCondWrong_pc4
    RET
```

`CX` holds `i`, and `416(AX)` accesses `b.N`. This is just an empty loop! Now the 0.24 nanoseconds per iteration make sense [4].

Here's an even more confounding manifestation of the same problem:

```
func countCond(b []byte) int {
  result := 0
  for i := 0; i < len(b); i++ {
    if isCond(b[i]) {
      result++
```

```
    }
  }
  return result
}

func BenchmarkCountWrong(b *testing.B) {
  inp := getInputContents()
  b.ResetTimer()
  for i := 0; i < b.N; i++ {
    countCond(inp)
  }
}

func getInputContents() []byte {
  n := 400000
  buf := make([]byte, n)
  for i := 0; i < n; i++ {
    buf[i] = byte(n % 32)
  }
  return buf
}
```

Now there are certainly no constant propagation issues, and the result of `isCond` is clearly used inside `countCond`. But we've committed the same error! While the result of `isCond` is used, the result of `countCond` is not, and therefore the compiler will do something like:

Inline `isCond` into `countCond`

Inline `countCond` into the benchmark function

Realize that the loop body in `coundCond` isn't producing side effects or any result used outside it

Hollow out the loop body in `countCond`, leaving only an empty loop

This process results in confusing benchmark results because the loop remains in place, and thus if we grow the input, the benchmarked execution time will grow accordingly! One of the oldest tricks in the benchmarking book is to change the input size and observe how the run-time is affected; if it's not moving, something is wrong. If it's growing roughly in line with the asymptotic complexity of the code under test, it's at least passing a simple smoke test. But in this case the heuristic fails due to the specific optimizations performed.

Keeping compiler optimizations in benchmarks under control

There are two main techniques in Go to thwart compiler optimizations where we don't want them.

The first is using the runtime.KeepAlive function. Originally introduced for fine-grained control of finalizers, it's also useful as a very-low-overhead way to tell the compiler "I really need this value, even if you can prove I do not". Here's how we can fix our `countCond` benchmark with its help:

```
func BenchmarkCountKeepAlive(b *testing.B) {
  inp := getInputContents()
  b.ResetTimer()
  result := 0
  for i := 0; i < b.N; i++ {
    result += countCond(inp)
  }
  runtime.KeepAlive(result)
}
```

Running the benchmark makes it clear that there's more work going into each iteration now:

```
BenchmarkCountWrong-8              12481      95911 ns/op
BenchmarkCountKeepAlive-8           4143     285527 ns/op
```

Another way is without needing a special function, but is slightly more dangerous; we can use a global exported value to collect the result:

```
var Sink int


func BenchmarkCountSink(b *testing.B) {
  inp := getInputContents()
  b.ResetTimer()
  for i := 0; i < b.N; i++ {
    Sink += countCond(inp)
  }
}
```

Even though our benchmark doesn't use `Sink`, it's very difficult for the compiler to eliminate because it's a package-level exported value that can be used pretty much anywhere in the program. I did say it's slightly more dangerous, though, because theoretically the compiler *could* prove it's redundant using more sophisticated cross-package analysis.

Which brings us the to ultimate point: when in doubt, always double check the assembly generated for the benchmark function, and ensure that the compiler wasn't too clever.

Ongoing work in the Go standard library

Addressing the issue of compiler optimizations in benchmarks is something the Go team is interested in. There are currently two active proposals in discussion:

Issue 61179: adding a low-overhead "keep this value" function in the `testing` package; this would be similar to `runtime.KeepAlive`, but more targeted at benchmarks with a better name and clearer guarantees.

A more ambitious proposal in issue 61515 discusses changing the main benchmarking API of the `testing` package to facilitate safer benchmarking.

Misusing `b.N`

There are at least two common ways to misuse the benchmark repetition indicator `b.N`. Here's the first, completely forgetting the loop:

```
import (
  "crypto/rand"
  "testing"
)

func BenchmarkRandPrimeWrongNoLoop(b *testing.B) {
  rand.Prime(rand.Reader, 200)
}
```

From proper measurements, the `crypto/rand.Prime` invocation with length 200 should take around 1 ms on my machine, but this benchmark reports impossibly low times - fractions of a nanosecond.

The second is trickier:

```
func BenchmarkRandPrimeWrongUseI(b *testing.B) {
  for i := 0; i < b.N; i++ {
    rand.Prime(rand.Reader, i)
  }
}
```

Note how it's using `i` in each iteration (and thus indirectly `b.N`). On my machine, this benchmark **doesn't terminate**; what gives?

To understand what's going on here, we should first discuss how benchmarking in Go works in a bit more detail. The benchmarking harness invokes our benchmark functions several times. How many? This can be determined by flags like `-benchtime`, but the default is for one second. How does the harness know how many repetitions to pass to our benchmark function in order to get 1s of execution time? By trying it with low

repetitions first, measuring how long it takes, and then running with increasingly high repetitions until the intended duration is reached [5].

There are some nuances to the process: e.g. the number of repetitions doesn't grow too fast between attempts (at most 100x, as of Go 1.21), and it is capped at 1 billion. The key point is that it uses the execution time from the previous attempt to determine how many repetitions to run next.

With these insights in mind, let's examine the two misuses shown above.

Forgetting to loop until `b.N`. Each benchmark invocation only performs the tested operations once. No matter how many times the benchmark harness repeats the benchmark with increasing `b.N`, the function takes a millisecond to run! Therefore, eventually the benchmark harness hits its `N` limit of a billion invocations, and divides the execution time (1 ms) by this `N` to get a nonsensical result.

Using the value of `b.N` for something other than "how many times to repeat the benchmark". `rand.Prime` is very fast when its input length is small, but gets pretty slow for large inputs. The harness starts by running the function once to get its bearings, and then 100 times. For size 100 the run-time of `rand.Prime` is moderate, so the next time the harness can increase `b.N` by another factor of 100. But for higher inputs, `rand.Prime` also takes much longer. We end up with a quadratic run-time explosion! Our benchmark function isn't literally hanging - it *will* finish eventually, but it may take long minutes or hours.

It's worth mentioning that the proposal in issue 61515 would address this problem as well, by exposing a benchmarking API that would be much harder to misuse in this way. Another proposal that may affect this is add range over int, which will permit us to write `for range b.N` to iterate exactly `b.N` times, without an explicit iteration variable.

---

[1]	For background on this work and why the new generic sorting functions are faster, see this post. If you're reading this before Go 1.21's final release (in Aug 2023), you can download the release candidate or use gotip.
[2]	To be clear, it's also important to know how well sorting algorithms perform on sorted or almost-sorted inputs, but here we're talking about the more general case.
[3]	While it's possible to think of ways to treat `Benchmark*` specially, it's pretty hard to do it correctly because we very much *want* the compiler to optimize the functions we're benchmarking. After all, we want to know what their real run-time is. Drawing the line between "optimize here" and "don't touch there" is tricky!
[4]	The Go compiler doesn't currently optimize loops entirely away, unless it can prove they terminate.
[5]	This also has the side effect of providing whatever *warming* the benchmark needs w.r.t. caches (CPU caches, paged memory, file system caches, DNS caches, etc).