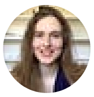
print(“lol”) doubled the speed of my Go function



Ludi Rehak · Follow

4 min read · 7 hours ago

Here is a Go function, `if_max()` . It finds the max value of an array of integers. Simple enough:

```
func if_max(values []int) int {
    maxV := values[0]
    for _, v := range values[1:] {
        if v > maxV {
            maxV = v
            continue
        }
    }
    return maxV
}
```

Next is another function, `if_max_lol()` , that is identical to `if_max()` , except that it adds a single `print("lol”)` statement:

```
func if_max_lol(values []int) int {
    maxV := values[0]
    for _, v := range values[1:] {
        if v > maxV {
            maxV = v
            continue
        }
        print("lol") // <---------- the new line
    }
    return maxV
}
```

I benchmark them on my M1 Apple Pro on an **increasing** array. Following standard practice, I use the `benchstat` tool to compare their speeds. Full code here.

```
goos: darwin
goarch: arm64
pkg: github.com/ludi317/max/blog
        │   if_max    │             if_max_lol             │
        │   sec/op    │   sec/op     vs base               │
Fn-10    63.51µ ± 2%   31.69µ ± 0%  -50.10% (p=0.002 n=6)
```

`if_max_lol()` runs in half the time of `if_max()`. What on Earth?!

Time to look at the disassembly. I generate a cpu profile by rerunning the benchmarks with an extra arg: `-test.cpuprofile cpu.out`. I feed the profile into `go tool pprof -http=”:” max.test cpu.out` and click on `View > Source` in my browser with great haste. Screenshots of the two functions of interest below:

/Users/lrehak/go/src/github.com/ludi317/max/blog/max_test.go

```
Total:       9.76s     10.09s (flat, cum) 54.36%
     2           .          .
     3           .          .           import (
     4           .          .                   "testing"
     5           .          .           )
     6           .          .
     7           .          .           func if_max(values []int) int {
     8           .          .                   maxV := values[0]
     9       9.34s      9.64s                   for _, v := range values[1:] {
    10       420ms      450ms                           if v > maxV {
             420ms      450ms 1000efc90:                      CMP R4, R3
```

/Users/lrehak/go/src/github.com/ludi317/max/blog/max_test.go

```
Total:       8.26s      8.38s (flat, cum) 45.15%
    17           .          .
    18           .          .           func if_max_lol(values []int) int {
    19           .          .                   maxV := values[0]
    20       8.06s      8.18s                   for _, v := range values[1:] {
    21       170ms      170ms                           if v > maxV {
             100ms      100ms 1000efd54:                      CMP R4, R3
              70ms       70ms 1000efd58:                      BLT -6(PC)
```

This view highlights that the comparison in `if_max_lol()` spends CPU time executing a branch instruction (**B**ranch if **L**ess **T**han): `BLT -6(PC)`.

So, the compiler saw the `print("lol")` statement in the source code, and generated an extra conditional branch instruction. This, in turn, invokes the branch predictor. What's that? Here is ChatGPT's explanation:

> *Branch prediction is a technique used in modern microprocessors to improve the throughput of their instruction pipelines by guessing the outcome of a conditional branch instruction before it's actually executed. Let's delve deeper into why it's needed and how it works.*
>
> *The Need for Branch Prediction:*
>
> *Microprocessors use pipelines to process multiple instructions simultaneously at different stages. Think of this like an assembly line in a factory, where different stages of a product are being worked on simultaneously. Each instruction goes through various stages like fetching, decoding, executing, and writing results.*
>
> *When a conditional branch instruction (like "if A is greater than B, jump to instruction X") comes into the pipeline, the CPU needs to know which instruction to fetch next: the one right after the branch or the one at the target of the jump (instruction X in the example). If the CPU waits until the branch instruction is fully executed to know the outcome, the entire pipeline would stall, leading to inefficiencies.*
>
> *How Branch Prediction Works:*
>
> *To avoid these stalls, the CPU makes an educated guess or "predicts" the outcome of the branch. This prediction allows the CPU to speculatively fetch and execute subsequent instructions.*
>
> *Challenges:*
>
> *The main challenge with branch prediction is ensuring accuracy. An incorrect prediction (a "misprediction") causes the pipeline to be flushed, leading to wasted cycles and reduced performance. As such, a significant amount of research and engineering effort is dedicated to improving branch predictor accuracy. Despite these efforts, no predictor is perfect, especially in the face of complex software behaviors.*

In this case, the array is increasing. As long as the predictor guesses "true" `if v > maxV`, it predicts with perfect accuracy. This allows it to keep its pipeline full and execute the next instruction without delay, maximizing its efficiency.

`cheat_max()` is a function that drops the `if v > maxV` comparison altogether. It takes a shortcut by explicitly avoiding the work of comparison.

```
func cheat_max(values []int) int {
    maxV := values[0]
    for _, v := range values[1:] {
        maxV = v
    }
    return maxV
}
```

```
goos: darwin
goarch: arm64
pkg: github.com/ludi317/max/blog
       │ if_max_lol  │           cheat_max             │
       │   sec/op    │   sec/op     vs base            │
Fn-10   31.70µ ± 3%   31.62µ ± 1%  ~ (p=0.145 n=6)
```

As the results show, `if_max_lol()` is about as fast as if it had no if-statement at all. Branch prediction effectively removes the if-statement.

Of course, this is just a toy example. If you wanted to know the max value of an increasing array, simply return the last element. These benchmarks serve to illustrate that very one-sided if-statements are where branching performance shines.

Even when the array contains **random** integers, branch prediction performs better than basic comparisons. There's no longer certainty about whether the kth element is a new max, but a different pattern emerges. The kth element has a `1/k` chance of being the new max. As long as the predictor predicts "false" for whether `v > maxV`, it has a `(k-1)/k` chance of being right, getting only more accurate as the array grows in size.

Instead of printing "lol", we can do something a little more useful to coax the compiler into generating a branch instruction, like finding the min value of the array.

```go
func if_max_min(values []int) (int, int) {
    minV := values[0]
    maxV := values[0]
    for _, v := range values[1:] {
         if v > maxV {
              maxV = v
              continue // generate branch instruction
         }
         if v < minV {
              minV = v
              continue // generate branch instruction
         }
    }
    return minV, maxV
}
```

How does `if_max_min()` compare against `if_max()` on a random array?

```
goos: darwin
goarch: arm64
pkg: github.com/ludi317/max
                  │ if_max_min  │              if_max            │
                  │   sec/op    │   sec/op     vs base           │
Fn/len=100000-10   64.98µ ± 1%   64.68µ ± 0%  ~ (p=0.132 n=6)
```

It's about as fast. But it gives you both the min and max of the array, whereas `if_max()` only gives the max. Branching allows it to deliver more information in the same amount of time.

Optimization

Computer Science

Go

Written by Ludi Rehak

298 Followers

Backend software engineer at Mist Systems

More from Ludi Rehak

Ludi Rehak in Towards Data Science

Are UUIDs really unique?

UUID stands for universally unique identifier. It looks like a 32-character sequence of letters and numbers separated by dashes. Some...

3 min read · May 28, 2017

-- 7

Ludi Rehak in Towards Data Science

Tuesday Birthday Problem

Below is a classic series of probability questions about children. They begin gently and grow steadily more difficult, to the point of…

4 min read · May 8, 2018

-- 2

Ludi Rehak

What I learned about the 1600s from Google's Books Ngram Viewer

How n-grams have trended over the centuries

5 min read · Sep 6, 2022

-- 1

Ludi Rehak

ChatGPT for Text Style Transfer: Part 2

Text style transfer (TST) is a NLP task that involves altering the style of text, while retaining its original meaning.

4 min read · Feb 26

--

See all from Ludi Rehak

Recommended from Medium

Diwakar Kashyap

Use of Golang

today i start learning Golang so everyday i will write a blog

3 min read · 4 days ago

--

Kevin Chisholm in Flutter

What’s new in Flutter 3.13

2D scrolling, faster graphics, Material 3 updates and more

12 min read · Aug 16

-- 20

Lists

It's never too late or early to start something

15 stories · 85 saves

General Coding Knowledge

20 stories · 235 saves

Now in AI: Handpicked by Better Programming

266 stories · 94 saves

Natural Language Processing

539 stories · 157 saves

Sumit Sagar

Mastering Go Channels - From Beginner to Pro

Go, often referred to as Golang, is a statically typed, compiled programming language that has become increasingly popular due to its...

8 min read · Aug 15

--

Dmitry Kruglov in Better Programming

The Architecture of a Modern Startup

Hype wave, pragmatic evidence vs the need to move fast

16 min read · Nov 7, 2022

-- 45

Arshlan

S1E5: Mastering the Concurrency in Go with Fan Out Design Pattern

In this series we have already learned: (Concurrency Design Patterns — YouTube)

4 min read · Aug 16

--

Ed Yu

Zig C/C++ Compiler — WTF is Zig C++

The power and complexity of Zig C/C++ Compiler in Zig

7 min read · Jul 20

-- 2

See more recommendations