Analyzing and Improving Performance issues with Go applications

The goal of every software developer should be to design and implement a fun to use product - slow software is never fun to use. On the contrary, making your own software faster is always a joy - so let me tell you about some things I found fun and practical when reworking Go applications around performance improvements.
Lets start of with some tales of mine and why you should even read this: I wrote an article about a leetcode solution and how I made it significantly faster[1]. I created a programming language and made it 8 times faster afterwards[2]. I wrote a paper about garbage collection of programming languages with a friend and Go was featured in it[3]. Furthermore I implemented a just in time compiler and made a go runtime for a programming language 14 times faster[4]. I also am currently working on a JSON parser for Go thats already beating the `encoding/json` package [5].

Analyzing Applications

To find areas for improvement, we can use the package go provides for this exact purpose. This isn't the place to discuss specifics, but I recommend tinkering with the package.

```
package main;import p"runtime/pprof"
func main() {
    f, _ := os.Create("cpu.pprof")
    p.StartCPUProfile(f)
    defer p.StopCPUProfile()
    // logic here
}
```

An other way for analysation is to use `hyperfine` for comparing the performance of two binaries.

Performance Tips for Go

Lets look at some specific Go tips and tricks for low hanging, fast universal changes one can make to get better performance out of your existing code. The first specific tip is to start preallocating slices and maps with values determined with benchmarks:

```
// don't
a := []byte{}
m := map[string]int{}
// do
a := make([]byte, 0, 16)
m := make(map[string]int, 16)
```

[1] https://xnacly.me/posts/2023/leetcode-optimization/
[2] https://xnacly.me/posts/2023/language-performance/
[3] https://xnacly.me/papers/modern_algorithms_for_gc.pdf
[4] https://xnacly.me/papers/tree-walk-vs-go-jit.pdf
[5] https://github.com/xNaCly/libjson

The next tip is to carefully decide between the builder / buffer structures, depending on the API you require. I generally recommend using `strings.Builder`, its faster than `bytes.Buffer`

```
BenchmarkBuffer-4       0.0000603 ns/op
BenchmarkString-4       0.0000466 ns/op
BenchmarkBufferLarge-4  0.004109 ns/op
BenchmarkStringLarge-4  0.003431 ns/op
```

If these options are too slow for you, consider buffering in your own `[]byte` and use `*(*string)(unsafe.Pointer(&buf))` or `unsafe.String(unsafe.SliceData(buf), len(buf))` this reuses the memory already stored at `[]byte`. The latter option can half the time spent in string concatenation - however both approaches use the unsafe package which makes no guarantees for portability and compatibility [6].

```
BenchmarkArray-4      0.0000222 ns/op
BenchmarkArrayLarge-4 0.002167 ns/op
```

If you consider a function to be hot, e.g. it being called often and in loops, you should switch from a generic function to a specific function - if applicable.

```
func generic[T any](data any) (T, bool) {
    v, ok := data.(T);
    if !ok { var e T; return e, false }
    return v, ok
}
func specific(data any) (bool, bool) {
    switch data.(type) {
    case bool:
        return true, true
    default:
        return false, false
    }
}
```

The results are very situation dependent and require lots of benchmarking.

```
BenchmarkGeneric-4  0.0003623 ns/op
BenchmarkSpecific-4 0.0003494 ns/op
```

To minimize the usage of expensive syscalls and batch input/output actions, the `bufio` package should always be used for files and other file like structures. The final tip is to use `(*bytes.Reader).ReadByte` instead of `(*bytes.Reader).ReadRune`.

```
BenchmarkReadByte-4 0.0004150 ns/op
BenchmarkReadRune-4 0.0008462 ns/op
```

Tu sum up: always benchmark all changes and note their improvements. If you make a lot of long living copies, as is often the case with interpreters and parsers, either use an arena or pointers - copying can be expensive if there are a lot of those. Always search for fast paths, the goal is to always do less, look for early returns, such as edge cases for zero values and such.

[6] https://pkg.go.dev/unsafe