Performance optimization sins

Real-life examples

VICTORIA
METRICS

Aliaksandr Valialkin
VictoriaMetrics founder and core developer

Hello

- I’m Aliaksandr Valialkin (aka @valyala)
- The author of fasthttp, fastjson, quicktemplate, etc.
- I’m fond of performance optimizations
- VictoriaMetrics founder and core developer

When do we need performance optimizations?

- Slow programs
- High costs
- Benchmark games

Performance optimization sins

- Better performance isn’t free
- The price may be too high

Performance optimization sins

- Fasthttp
- Fastjson
- VictoriaMetrics
- Standard Go packages
- CPUs

https://github.com/valyala/fasthttp

Fasthttp

- Response buffering
- Deferring HTTP headers' parsing
- Slices instead of maps for (key -> value) entries
- RequestCtx re-use
- DNS caching

Fasthttp response buffering

- Standard HTTP protocol flow

```go
func ProcessHTTPConn(c net.Conn) {
    for {
        req, closed := readRequest(c)
        if closed {
            return
        }
        resp := processRequest(req)
        writeResponse(c, resp)
    }
}
```

Fasthttp response buffering

- Protocol flow optimized for HTTP pipelining

```go
func ProcessPipelinedHTTPConn(c net.Conn) {
    br := bufio.NewReader(c)
    bw := bufio.NewWriter(c)
    for {
        req, closed := readRequest(br)
        if closed {
            bw.Flush()
            return
        }
        resp := processRequest(req)
        writeResponse(bw, resp)
        if br.Buffered() == 0 {
            // Send buffered responses to client, since no more buffered requests
            bw.Flush()
        }
    }
}
```

Fasthttp response buffering benefits

- Reduces the number of **recv** syscalls, since multiple small requests may be read from conn via a single **recv** syscall
- Reduces the number of **send** syscalls, since multiple responses may be flushed to conn via a single **send** syscall

Fasthttp response buffering sins

- Delays responses for slow pipelined requests
- Doesn’t provide any speedup for real-world HTTP servers, since modern web-browsers don’t use HTTP pipelining due to historical bugs. See https://en.wikipedia.org/wiki/HTTP_pipelining#Implementation_status
- Provides 2x speedup for Techempower plaintext benchmark. See https://github.com/TechEmpower/FrameworkBenchmarks/issues/4410

Deferring HTTP headers’ parsing

- Fasthttp defers parsing of HTTP headers until they are needed
- It just searches for the “end of headers” marker in the input buffer and puts unparsed headers into a byte slice
- The byte slice is parsed into HTTP headers structure on the first access

Deferring HTTP headers’ parsing

- The following code skips parsing HTTP headers:

```
fasthttp.ListenAndServe(":8080", func(ctx *fasthttp.RequestCtx) {
    ctx.WriteString("Hello, world!")
})
```

Deferring HTTP headers’ parsing

The following code parses HTTP headers:

```
fasthttp.ListenAndServe(":8080", func(ctx *fasthttp.RequestCtx) {
    clientAddr := ctx.Request.Header.Peek("X-Forwarded-For")
    fmt.Fprintf(ctx, "client addr: %q", clientAddr)
})
```

Deferring HTTP headers' parsing benefits

- If request handler doesn't access HTTP headers, then CPU time is saved

Deferring HTTP headers' parsing sins

- Real-world HTTP handlers usually consult HTTP headers
- Techempower plaintext benchmark is the only user that benefits from this optimization :)

Slices instead of maps for (key -> value)

- Fasthttp uses slices instead of maps for storing the following entities:
  - HTTP headers
  - Query args
  - Cookies

Slices instead of maps for (key -> value)

- (key->value) slice structs:

```
type kv struct {
  key   []byte
  value []byte
}

type sliceMap []kv
```

Slices instead of maps for (key -> value)

- How to add an entry to **sliceMap** and re-use memory

```
func (sm *sliceMap) Add(k, v []byte) {
    kvs := *sm
    if cap(kvs) > len(kvs) {
        kvs = kvs[:len(kvs)+1]
    } else {
        kvs = append(kvs, kv{})
    }
    kv := &kvs[len(kvs)-1]
    kv.key = append(kv.key[:0], k...)
    kv.value = append(kv.value[:0], v...)
    *sm = kvs
}
```

Slices instead of maps for (key -> value)

- How to get value for the given key from **sliceMap**

```
func (sm sliceMap) Get(k string) []byte {
    for i := range sm {
        kv := &sm[i]
        if string(kv.key) == k {
            return kv.value
        }
    }
    return nil
}
```

Slices instead of maps for (key -> value): benefits

- Usually the number of entries in headers, query args or cookies is quite small (less than 10)
- Slices provide better performance comparing to maps for this case
- Slices allow memory re-use
- Slices save the original order of added items

Slices instead of maps for (key->value): sins

- **sliceMap.Get** has **O(N)** complexity, while standard map has **O(1)** complexity
- **sliceMap** is vulnerable to memory fragmentation on re-use:
  - Write **hugeValue** into **sliceMap**
  - Now **sliceMap** occupies at least **len(hugeValue)** bytes of memory when re-used
- The value returned from **sliceMap.Get** is valid only until the **sliceMap** is re-used

RequestCtx re-use

```go
func processConn(c net.Conn, requestHandler RequestHandler) {
    // The ctx is re-used across requestHandler calls
    ctx := AcquireRequestCtx()
    defer ReleaseRequestCtx(ctx)
    for {
        ctx.Reset()
        readRequest(c, ctx)
        requestHandler(ctx)
        writeResponse(c, ctx)
    }
}
```

RequestCtx re-use: benefits

- Reduced memory allocations, since the **RequestCtx** memory is re-used
- Better performance, since **RequestCtx** is already in CPU cache on re-use

RequestCtx re-use: sins

- Easy to shoot in the foot by holding references to **RequestCtx** contents after returning from **RequestHandler**
- Possible memory fragmentation/calcification on **RequestCtx** re-use

DNS caching

- Fasthttp caches (**host -> IP**) entries for a minute

```go
func resolveHost(host string) net.IP {
    e := dnsCache[host]
    if e != nil && time.Since(e.resolveTime) < time.Minute {
        // Fast path - return the ip from cache.
        return e.ip
    }
    // Slow path - really resolve the host to ip and put it to cache
    ip := reallyResolveHost(host)
    dnsCache[host] = ip
    return ip
}
```

DNS caching benefits

- Reduced load on DNS subsystem
- Faster dials to remote hosts

DNS caching sins

- Breaks on frequent DNS changes (when Kubernetes restarts pods) https://github.com/valyala/fastjson

Fastjson

- Memory re-use
- Fast string unescaping
- Custom parser for integers and floats

Fastjson memory re-use

- **fastjson.Parser** owns all the JSON data structure
- **fastjson.Parser** re-uses the memory for JSON data structure

```
var p fastjson.Parser
v, err := p.Parse(`{"foo": "bar"}`) // v belongs to p
...
b := v.GetStringBytes("foo") // b also belongs to p
...
vv, err := p.Parse(`[1,2,3]`)
// vv overwrites v contents, so v and b become invalid
```

Fastjson memory re-use: benefits

- Reduced memory allocations
- Improved performance, since the memory remains in CPU caches across **Parse** invocations

Fastjson memory re-use: sins

- High memory usage after parsing random big JSON objects
- “Shoot in the foot” API - all the JSON structures returned by **Parser** cannot be used after the next **Parse** call. Recursively

Fast string unescaping

```go
func unescapeJSONString(s string) string {
	n := strings.IndexByte(s, '\\')
	if n < 0 {
		// Fast path - the string has no escape chars.
		return s
	}
	// Slow path - unescape every char in s
	return unescapeJSONStringSlow(s)
}
```

Fast string unescaping: benefits

- Works fast for strings without escape chars

Fast string unescaping: sins

- Works slow on strings with escaped chars
- Uneven performance on mixed strings

Custom parser for integer and floats

- Fastjson doesn't use **strconv.ParseFloat** and **strconv.ParseInt**
- It uses custom functions optimized for speed

Custom parser for integer and floats: benefits

- Faster than the corresponding functions from **strconv**

Custom parser for integer and floats: sins

- It falls back to **strconv** for too big numbers -> slower performance
- It returns 0 for invalid numbers
- It may work improperly for corner cases

https://victoriametrics.com

VictoriaMetrics

- Hand-written protobuf parsing with memory re-use

Hand-written protobuf parsing

- VictoriaMetrics accepts protobuf via Prometheus remote write API
- Initially we used parser generated by standard protobuf generator
- It was allocating like a hell, so it had been rewritten to zero-alloc mode
- Now protobuf parsers re-use the provided structs

Hand-written protobuf parser

```
type TimeSeries struct {
        Labels  []Label
        Samples []Sample
}

type Label struct {
        Name  []byte
        Value []byte
}

type Sample struct {
        Value     float64
        Timestamp int64
}
```

Hand-written protobuf parser

- Diff examples:

```
- Name  string
+ Name  []byte

- m.Name = string(dAtA[iNdEx:postIndex])
+ m.Name = dAtA[iNdEx:postIndex]
```

Hand-written protobuf parser

- Diff examples:

```
- Labels  []*Label
+ Labels  []Label

- m.Labels = append(m.Labels, &Label{})
+ if cap(m.Labels) > len(m.Labels) {
+     m.Labels = m.Labels[:len(m.Labels)+1]
+ } else {
+     m.Labels = append(m.Labels, Label{})
+ }
```

Hand-written protobuf parsing: benefits

- Zero allocations
- Better performance

Hand-written protobuf parsing: sins

- The hand-written code is fragile
- It is hard to update the code if Prometheus remote write API changes
- The parsed struct cannot be referenced after the next Unmarshal call

Standard Go packages

Standard Go packages

- Pools for small numbers
- **sync.Pool** memory fragmentation

Pools for small numbers

- Go doesn’t allocate on **strconv.Itoa(smallNum)**:

```
func FormatInt(i int64, base int) string {
     if 0 <= i && i < nSmalls && base == 10 {
          return small(int(i))
     }
     _, s := formatBits(nil, uint64(i), base, i < 0, false)
     return s
}

func small(i int) string {
     if i < 10 {
          return digits[i : i+1]
     }
     return smallsString[i*2 : i*2+2]
}
```

Pools for small numbers: benefits

- Faster performance for small numbers (up to 99)
- Zero memory allocations for small numbers

Pools for small numbers: sins

- Uneven performance for mixed numbers
- Slower performance for bigger numbers

sync.Pool memory fragmentation

- sync.Pool allows re-using objects with the underlying memory
- Benefit: reduced memory allocations and improved performance
- Sin: possible high memory usage:
    - Suppose **sync.Pool** contains small byte slices
    - Put a big byte slice into **sync.Pool**
    - Now the big byte slice wastes memory when obtained from the pool, since only a small amount of allocated memory is really used

CPUs

CPUs

- Modern CPUs execute instructions in multi-stage pipeline
- This improves performance
- The pipeline is reset on each conditional branch, leading to delay
- CPU makers added prediction block and speculative execution, which may go beyond branches
- Benefit: CPU runs full speed until the first branch mispredict
- Sins: **Meltdown** and **Spectre**-like vulnerabilities

Conclusion

Conclusion

- Performance optimization is full of hard decisions:
    - Speed vs clarity
    - Speed vs simplicity
    - Speed vs nice API
    - Speed vs consistent performance for all the cases
    - Speed vs consistent memory usage
    - Speed vs precision
    - Speed vs lower vulnerability risk
- There is no silver bullet for making perfect decision
- Choose wisely

Questions?

Aliaksandr Valialkin
VictoriaMetrics founder and core developer