How to optimize Go code for really high performance

NewStore tech talk
Berlin, 2016-02-15

Björn “Beorn” Rabenstein, Production Engineer, SoundCloud Ltd.

Prometheus

Half a million samples per second!

Prologue: Avoidance Strategies

Don’t use Go, use Assembly.

https://golang.org/doc/asm

http://goroutines.com/asm

```
// add.go
package main

import "fmt"

func add(x, y int64) int64

func main() {
    fmt.Println(add(2, 3))
}

// add_amd64.s
TEXT ·add(SB),NOSPLIT,$0
    MOVQ x+0(FP), BX
    MOVQ y+8(FP), BP
    ADDQ BP, BX
    MOVQ BX, ret+16(FP)
    RET
```

Don’t use Go, use C/C++.

Compile time for Prometheus server

1.4.2	0:03.44
1.5.3	0:08.57
1.6rc2	0:08.29

CGO

Allows the use of highly optimized C/C++ libraries. But use it judiciously.

- ❏ Loss of certain advantages of the Go build environment.
- ❏ Per-call overhead (~150ns) for various reasons.
- ❏ Need to shovel input and output data back and forth to stay “safe”.

http://jmoiron.net/blog/go-performance-tales/

CGO case studies

Best case: Use a highly optimized C library for encryption.

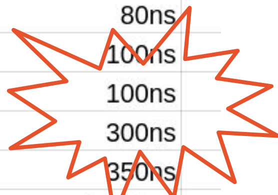
Worst case: LevelDB C++ implementation for Prometheus.

(500,000 samples per second times 150ns…)

Numbers every Go + App Engine dev should know

File Edit View Insert Format Data Tools Add-ons Help View only

	A	B
1	L1 cache reference	1ns
2	Branch mispredict	5ns
3	L2 cache reference	7ns
4	Mutex lock/unlock	25ns
5	Linear seek in 100 long slice @ F1	80ns
6	Main memory reference	100ns
7	Go sync.Mutex lock/unlock @ F1	100ns
8	Go channel send/receive @ F1	300ns
9	Go native map retrieval @ F1	350ns
10	Go native map insertion @ F1	1.000ns
11	Call method reflectively @ F1	1.000ns
12	Compress 1kb /w cheap compression algorithm	3.000ns
13	Send 2kb over 1GBPS network	20.000ns
14	Read 1Mb sequentially from memory	250.000ns
15	Roundtrip within datacenter	500.000ns
16	Memcache access @ F1	3.000.000ns
17	Datastore put @ F1	6.000.000ns
18	Datastore get @ F1	6.000.000ns
19	Datastore query @ F1	6.500.000ns
20	Disk seek	10.000.000ns
21	Read 1MB sequentially from disk	20.000.000ns
22	Roundtrip over the atlantic	150.000.000ns
23		



gccgo

No “CGO wall”. But other issues…

Wait for the next Go release.

Prometheus BenchmarkFuzzChunkType1

1.4.2	2.05 s
1.5.3	1.75 s
1.6rc2	1.68 s

Chapter 1: Measure Everything

Benchmarks are easy

And benchmark allocations, too.

```
$ go test -bench=. -cpu=1,4,16 -benchmem
```

```
func BenchmarkAddAndWrite(b *testing.B) {
    c := NewCounter()
    for i := 0; i < b.N; i++ {
        if i%1000 == 0 {
            c.Write(&dto)
            continue
        }
        c.Add(42)
    }
}
```

pprof.

Runtime and allocation profiling.

```
import _ "net/http/pprof"
```

```
$ go tool pprof http://localhost:9090/debug/pprof/profile
(pprof) web
```

```
$ go tool pprof http://localhost:9090/debug/pprof/heap
(pprof) web
```

runtime.goexit
0 of 38.20s(81.42%)

github.com/prometheus/prometheus/storage/local.func·029
0.06s(0.13%)
of 0.46s(0.98%)

github.com/prometheus/prometheus/retrieval.func·008
0.01s(0.02%)
of 0.56s(1.19%)

github.com/prometheus/prometheus/retrieval.(*target).scrape
0.06s(0.13%)
of 2.16s(4.60%)

github.com/prometheus/prometheus/storage/local.(*persistence).checkpointSeriesMapAndHeads
0.01s(0.02%)
of 30.17s(64.30%)

github.com/prometheus/prometheus/storage/local/index.(*LabelPairFingerprintIndex).LookupSet
0 of 0.40s(0.85%)

github.com/matttproud/golang_protobuf_extensions/pbutil.ReadDelimited
0 of 0.48s(1.02%)

github.com/prometheus/prometheus/storage/local.(*memorySeriesStorage).Append
0.02s(0.04%)
of 1.79s(3.82%)

github.com/prometheus/prometheus/storage/local.func·019
1.79s(3.82%)
of 30.01s(63.96%)

github.com/prometheus/prometheus/storage/local/index.(*LevelDB).Get
0 of 0.40s(0.85%)

github.com/prometheus/client_golang/model.Metric.FastFingerprint
0.01s(0.02%)
of 0.45s(0.96%)

github.com/prometheus/prometheus/storage/local.(*chunkDesc).firstTime
0.14s(0.3%)
of 9.09s(19.37%)

github.com/prometheus/prometheus/storage/local.(*chunkDesc).lastTime
0.15s(0.32%)
of 3.24s(6.91%)

github.com/prometheus/prometheus/storage/local.doubleDeltaEncodedChunk.marshal
0.18s(0.38%)
of 0.88s(1.88%)

github.com/prometheus/prometheus/storage/local/codable.Metric.MarshalBinary
0.08s(0.17%)
of 1.62s(3.45%)

github.com/prometheus/prometheus/storage/local.(*doubleDeltaEncodedChunk).firstTime
0.58s(1.24%)
of 1.31s(2.79%)

github.com/prometheus/prometheus/storage/local.(*doubleDeltaEncodedChunk).lastTime
0.04s(0.085%)
of 0.84s(1.79%)

sync.(*Mutex).Unlock
0.60s(1.28%)
of 1.07s(2.28%)

github.com/prometheus/prometheus/storage/local/codable.encodeString
0.02s(0.04%)
of 1.29s(2.75%)

runtime.mapassign1
0.39s(0.83%)
of 0.48s(1.02%)

sync.(*Mutex).Lock
0.11s(0.23%)
of 5.99s(12.77%)

sync/atomic.AddUint32
0.53s(1.13%)

bytes.(*Buffer).WriteString
0.07s(0.15%)
of 0.54s(1.15%)

github.com/prometheus/prometheus/storage/local/codable.EncodeVarint
0.71s(1.51%)
of 13.78s(29.37%)

runtime.deferreturn
1.93s(4.11%)
of 5.24s(11.17%)

sync/atomic.CompareAndSwapUint32
5.87s(12.51%)

runtime.deferproc
0.77s(1.64%)
of 2.04s(4.35%)

bufio.(*Writer).Write
0.71s(1.51%)
of 1.94s(4.13%)

github.com/prometheus/prometheus/storage/local/codable.putBuf
0.63s(1.34%)
of 6.36s(13.55%)

runtime.freedefer
0.93s(1.98%)
of 1.74s(3.71%)

runtime.getg
0.36s(0.77%)

syscall.Syscall
0.82s(1.75%)

System
0 of 8.71s(18.56%)

runtime.convT2E
0.52s(1.11%)
of 4.49s(9.57%)

sync.(*Pool).Put
0.55s(1.17%)
of 1.27s(2.71%)

runtime.acquirem
0.66s(1.41%)

runtime.releasem
0.94s(2.00%)

runtime.memmove
1.91s(4.07%)

runtime.onM
0.56s(1.19%)

runtime.newdefer
1.68s(3.58%)

runtime.futex
1.38s(2.94%)

runtime.deferproc_m
0.69s(1.47%)

runtime.atomicload
0.67s(1.43%)

findrunnable
0.56s(1.19%)

runtime.epollwait
0.48s(1.02%)

runtime.newobject
0.37s(0.79%)
of 4.38s(9.34%)

runtime.mallocgc
3.56s(7.59%)
of 4.17s(8.89%)

runtime.gomcache
0.36s(0.77%)

runtime.writebarrierptr
1.85s(3.94%)

Prometheus Alerts Graph Status Help

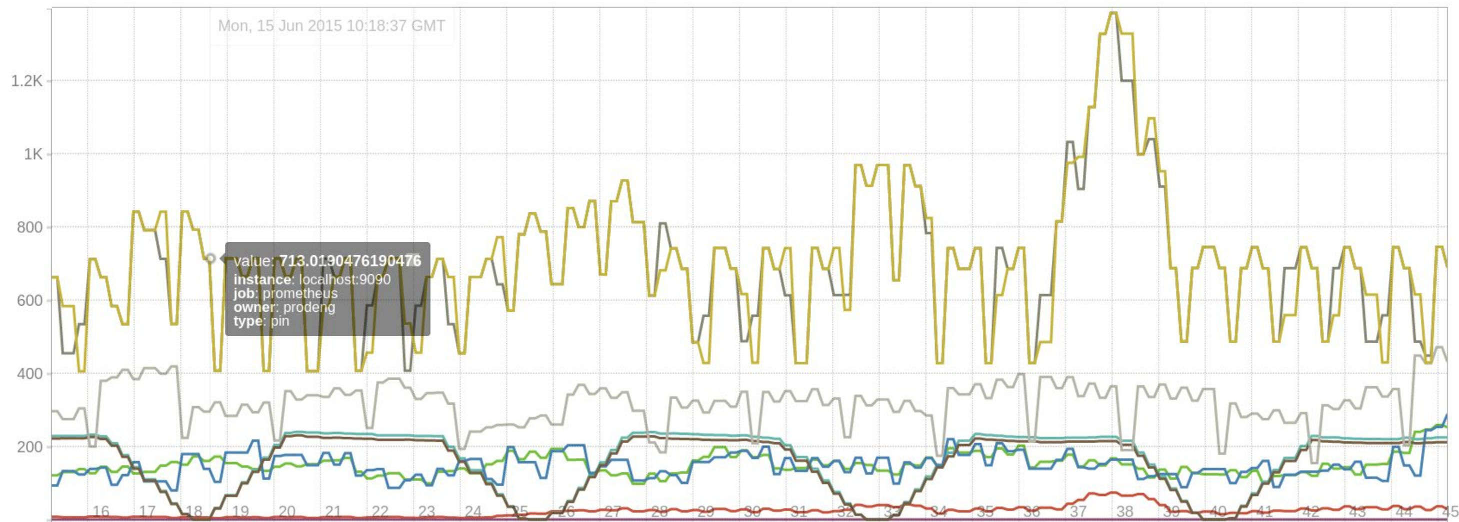
rate(prometheus_local_storage_chunk_ops_total[2m]) Execute

Load time: 145ms
Resolution: 7s

- Insert Metric at Cursor -

Graph Console

− 30m + ◀◀ Until ▶▶ Res. (s) stacked



{instance="localhost:9090",job="prometheus",owner="prodeng",type="unpin"}
{instance="localhost:9090",job="prometheus",owner="prodeng",type="transcode"}
{instance="localhost:9090",job="prometheus",owner="prodeng",type="pin"}
{instance="localhost:9090",job="prometheus",owner="prodeng",type="persist"}
{instance="localhost:9090",job="prometheus",owner="prodeng",type="load"}
{instance="localhost:9090",job="prometheus",owner="prodeng",type="evict"}
{instance="localhost:9090",job="prometheus",owner="prodeng",type="drop"}
{instance="localhost:9090",job="prometheus",owner="prodeng",type="create"}
{instance="localhost:9090",job="prometheus",owner="prodeng",type="clone"}

Chapter 2: Allocation Churn

Perhaps don't care?

GC has improved over time.

GC times of a really busy Prometheus server

	pause time percentage	pause time per GC
1.4.2	2.8%	750ms
1.5.3	0.9%	220ms
1.6rc2	0.04%	9ms

(taken from debug.GCStats.{NumGC,PauseTotal} and processed with Prometheus)

Code with care

```
for i := 0; i < b.N; i++ {
	var (
		result uint64
		hash   = fnv.New64a()
	)

	for labelName, labelValue := range labels {
		hash.Write([]byte(labelName))
		hash.Write([]byte{separatorByte})
		hash.Write([]byte(labelValue))
		result ^= hash.Sum64()
	}
}
```

759 ns/op 256 B/op 16 allocs/op

```
var (
	result uint64
	hash   = fnv.New64a()
	buf    bytes.Buffer
)

for i := 0; i < b.N; i++ {
	for labelName, labelValue := range labels {
		buf.WriteString(labelName)
		buf.WriteByte(separatorByte)
		buf.WriteString(labelValue)
		hash.Write(buf.Bytes())
		result ^= hash.Sum64()
		hash.Reset()
		buf.Reset()
		result = 0
	}
}
```

358 ns/op 0 B/op 0 allocs/op

sync.Pool

```
bufPool := sync.Pool{New: func() interface{} {
  return make([]byte, 0, 3*chunkLenWithHeader)
}}

func writeChunks(w io.Writer, chunks []chunk) error {
      b := bufPool.Get().([]byte)
      defer func() {
            p.bufPool.Put(b)
      }()
      // Why not 'defer p.bufPool.Put(b)'?
      writeSize := len(chunks) * chunkLenWithHeader
      if cap(b) < writeSize {
            b = make([]byte, writeSize)
      }
      b = b[:writeSize]
      // ... (Encode chunks into b.)
      if _, err := w.Write(b); err != nil {
                  return err
            }
      }
      return nil
}
```

“An appropriate use of a Pool is to manage a group of temporary items silently shared among and potentially reused by concurrent independent clients of a package. Pool provides a way to amortize allocation overhead across many clients.

An example of good use of a Pool is in the fmt package, which maintains a dynamically-sized store of temporary output buffers. The store scales under load (when many goroutines are actively printing) and shrinks when quiescent.

On the other hand, a free list maintained as part of a short-lived object is not a suitable use for a Pool, since the overhead does not amortize well in that scenario. It is more efficient to have such objects implement their own free list.”

Free list

Can be implemented with a channel.

```
type FooFreeList chan *Foo

func (fl FooFreeList) Get() *Foo {
  select {
  case f := <-fl:
    return f
  default:
    return NewFoo()
  }
}

func (fl FooFreeList) Put(f *Foo) {
  f.Reset()
  select {
  case fl <- f:
    return f
  default: // Let it go.
}

fl := FooFreeList(make(chan *Foo, maxFreeListItems))
```

Avoid fragmentation

There are many kinds of memory...

Escape analysis

```
var p *int

func BenchmarkEscape(b *testing.B) {
	for i := 0; i < b.N; i++ {
		j := i
		p = &j
	}
}

$ go test -gcflags=-m -bench=Something -benchmem
[...]
./alloc_test.go:100: moved to heap: j
./alloc_test.go:101: &j escapes to heap
[...]
29.7 ns/op             16 B/op          1 allocs/op
```

Chapter 3:
Lock Contention

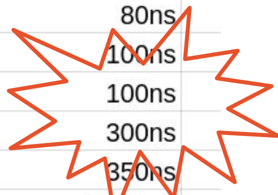
```go
type Counter struct {
    value int
    mtx   sync.Mutex
}

func (c *Counter) Add(v int) {
    c.mtx.Lock()
    defer c.mtx.Unlock()
    c.value += v
}

func (c *Counter) Get() int {
    c.mtx.Lock()
    defer c.mtx.Unlock()
    return c.value
}
```

Numbers every Go + App Engine dev should know

File Edit View Insert Format Data Tools Add-ons Help View only

L1 cache reference

	A	B
1	L1 cache reference	1ns
2	Branch mispredict	5ns
3	L2 cache reference	7ns
4	Mutex lock/unlock	25ns
5	Linear seek in 100 long slice @ F1	80ns
6	Main memory reference	100ns
7	Go sync.Mutex lock/unlock @ F1	100ns
8	Go channel send/receive @ F1	300ns
9	Go native map retrieval @ F1	350ns
10	Go native map insertion @ F1	1.000ns
11	Call method reflectively @ F1	1.000ns
12	Compress 1kb /w cheap compression algorithm	3.000ns
13	Send 2kb over 1GBPS network	20.000ns
14	Read 1Mb sequentially from memory	250.000ns
15	Roundtrip within datacenter	500.000ns
16	Memcache access @ F1	3.000.000ns
17	Datastore put @ F1	6.000.000ns
18	Datastore get @ F1	6.000.000ns
19	Datastore query @ F1	6.500.000ns
20	Disk seek	10.000.000ns
21	Read 1MB sequentially from disk	20.000.000ns
22	Roundtrip over the atlantic	150.000.000ns
23		



Mutexes are for parallelism

150ms without contention, but…

ns/op	1 Goroutine	10 Goroutines	100 Goroutines
GOMAXPROCS=1	150	160	190
GOMAXPROCS=4	150	730	570
GOMAXPROCS=16	150	1100	1100

Check out `func (*B) RunParallel` and `func (*B) SetParallelism` to implement parallel benchmarks.

How to avoid lock contention

1. Don't optimize prematurely.
2. Design your code appropriately. ("Do not communicate by sharing memory; share memory by communicating" etc.)
3. Look at `sync/atomic` for low-level cases.

Atomic counter

```
type Counter struct {
    value int64
}

func (c *Counter) Add(v int64) {
    atomic.AddInt64(&c.value, v)
}

func (c *Counter) Get() int64 {
    return atomic.LoadInt64(&c.value)
}
```

Atomic counter 2

```
type Counter int64

func (c *Counter) Add(v int64) {
    atomic.AddInt64(c, v)
}

func (c *Counter) Get() int64 {
    return atomic.LoadInt64(c)
}
```

Atomic counter benchmarked

ns/op	1 Goroutine	10 Goroutines	100 Goroutines
GOMAXPROCS=1	15	14	15
GOMAXPROCS=4	14	45	44
GOMAXPROCS=16	14	47	45

Chapter 4:
Hashes, Maps, and Hashmaps

Lookup in a hashmap is O(1), right?

Results rounded to two significant digits to avoid the noise.

```
var dictionary map[string]string
```

	var dictionary map[string]string		hardware AES support	
BenchmarkMapInt	53 ns/op	*1.0	64 ns/op	*1.0
BenchmarkMapString1	56 ns/op	*1.1	59 ns/op	*0.9
BenchmarkMapString10	63 ns/op	*1.2	66 ns/op	*1.0
BenchmarkMapString100	110 ns/op	*2.1	80 ns/op	*1.3
BenchmarkMapString1000	510 ns/op	*9.6	330 ns/op	*5.2
BenchmarkMapString10000	2600 ns/op	*49	1300 ns/op	*20

An ordinary Prometheus metric

```
apiv3_incoming_http_request_latency_microseconds
_bucket{method="GET",path="/foos/bars/orders/sta
tus",status="200",le="5000",instance="ip-10-20-1
17-38.xy.bla.example.net:19928",job="api-v3",dep
loy="canary",zone="xy",owner="billing-team"}
```

250 characters… or an int64 hash value.

fnv64a

So easy, the whole code fits on this slide.

```
const (
    offset64 = 14695981039346656037
    prime64  = 1099511628211
)

func hashNew() uint64 {
    return offset64
}
```

```
func hashAdd(h uint64, s string) uint64 {
    for i := 0; i < len(s); i++ {
        h ^= uint64(s[i])
        h *= prime64
    }
    return h
}

func hashAddByte(h uint64, b byte) uint64 {
    h ^= uint64(b)
    h *= prime64
    return h
}
```

```go
func LabelsToSignature(labels map[string]string) uint64 {
	if len(labels) == 0 {
		return emptyLabelSignature
	}

	labelNames := make([]string, 0, len(labels))
	for labelName := range labels {
		labelNames = append(labelNames, labelName)
	}
	sort.Strings(labelNames)

	sum := hashNew()
	for _, labelName := range labelNames {
		sum = hashAdd(sum, labelName)
		sum = hashAddByte(sum, SeparatorByte)
		sum = hashAdd(sum, labels[labelName])
		sum = hashAddByte(sum, SeparatorByte)
	}
	return sum
}
```

And now make it faster…

```
func LabelsToFastSignature(labels map[string]string) uint64 {
    if len(labels) == 0 {
        return emptyLabelSignature
    }

    var result uint64
    for labelName, labelValue := range labels {
        sum := hashNew()
        sum = hashAdd(sum, labelName)
        sum = hashAddByte(sum, SeparatorByte)
        sum = hashAdd(sum, labelValue)
        result ^= sum
    }
    return result
}
```

Epilogue:
The Price of Abstraction

The Go Hash interface

```
type Hash interface {
	// Write (via the embedded io.Writer interface) adds more data to the running hash.
	// It never returns an error.
	io.Writer

	// Sum appends the current hash to b and returns the resulting slice.
	// It does not change the underlying hash state.
	Sum(b []byte) []byte

	// Reset resets the Hash to its initial state.
	Reset()

	// Size returns the number of bytes Sum will return.
	Size() int

	// BlockSize returns the hash's underlying block size.
	// The Write method must be able to accept any amount
	// of data, but it may operate more efficiently if all writes
	// are a multiple of the block size.
	BlockSize() int
}

type Hash64 interface {
	Hash
	Sum64() uint64
}
```

Before and after…

```go
var b bytes.Buffer // Could retrieve those
h := fnv.New64a()  // from a sync.Pool.

var result uint64
for labelName, labelValue := range labels {
     b.WriteString(labelName)
     b.WriteByte(SeparatorByte)
     b.WriteString(labelValue)
     h.Write(b.Bytes())
     result ^= h.Sum64()
     h.Reset()
     b.Reset()
}
return result
```

```go
var result uint64
for labelName, labelValue := range labels {
     sum := hashNew()
     sum = hashAdd(sum, labelName)
     sum = hashAddByte(sum, SeparatorByte)
     sum = hashAdd(sum, labelValue)
     result ^= sum
}
return result
```

Naive hash: 759 ns/op
Allocation free hash (left): 358 ns/op
Inlined hash (right): 157 ns/op

The End