Performance patches in Go 1.11

Aliaksandr Valialkin, VictoriaMetrics

About me

- I like Go and performance optimizations
- I’m the author of fasthttp, fastjson, fastrpc, fastrand, quicktemplate and many other projects - see https://github.com/valyala/
- Now I work on the fastest time series DB - VictoriaMetrics. It is written in Go

Agenda

- Compiler and runtime optimizations
- Math/big optimizations (aka ‘crypto-optimizations’)
- Standard library optimizations
- Arch-specific optimizations

Compiler and runtime optimizations

[109918](#) cmd/compile:
refactor inlining parameters; inline panic

What is inlining?

- Inlining is the process of embedding function code into the place of function call
- Inlining eliminates function call overhead
- Inlining opens additional optimization opportunities for the compiler
- Inlining may improve performance

What is inlining?

- But sometimes inlining may hurt performance
- Big functions’ inlining may lead to binary size bloat and bad performance if the resulting binary code stops fitting CPU instruction cache
- So it is better to inline small functions
- Go compiler performs basic inlining

Inline functions with panic

- Go 1.11 may inline functions with panic()
- Panic may be implicit:
  - If slice element is accessed without explicit or provable bounds check, then the compiler translates **`a[i]`** into

```
if i < 0 || i >= len(a) {

    panic("out of bounds access")

}

a[i]
```

Inline functions with panic

- Go 1.11 may inline functions with panic()
- Panic may be implicit:
  - If struct field is accessed via struct pointer without explicit or provable nill check, then the compiler may translate **`foo.bar`** for **`foo *T`** into

```
if foo == nil {
    panic(“nil dereference”)
}
foo.bar
```

Inline functions with panic

- Go 1.11 is able to inline the following function:

```
type T struct {
    N int
}
func f(a []int, b *T) {
    b.N = a[2]
}
```

110055 cmd/compile:
optimize map-clearing range idiom

Optimize map-clearing range idiom

- Go 1.11 now detects and optimizes the following code

```
func clearMap(m map[K]V) {
    for k := range m {
        delete(m, k)
    }
}
```

Optimize map-clearing range idiom

- It is substituted by something like:

```
func clearMap(m map[K]V) {
    // fastClearMap is an optimized function
    // for clearing the map. It preserves m capacity
    // in order to reduce overhead during
    // subsequent additions into the map
    fastClearMap(m)
}
```

109517 cmd/compile: optimize append(x, make([]T, y)...) slice extension

Optimize append(x, make([]T, y)...)

- Previously the following code was frequently used for fast slice extension:

```
func growSlice(a []T, itemsToAdd int) []T {
    newSize := len(a) + itemsToAdd
    for cap(a) < newSize {
        a = append(a[:cap(a)], T{})
    }
    return a[:newSize]
}
```

Optimize append(x, make([]T, y)...)

- Now this code may be substituted by simpler and faster code:

```
func growSlice(a []T, itemsToAdd int) []T {
    return append(x, make([]T, itemsToAdd)...)
}
```

- Previously such code wasn’t optimal, since Go performed an unnecessary allocation for `make([]T, itemsToAdd)`.

91557 cmd/compile:
avoid extra mapaccess in "m[k] op= r"

Avoid extra mapaccess in “m[k] op= r”

- Now the following code works faster:

```
func countWords(words []string) map[string]int {
    m := make(map[string]int)
    for _, w := range words {
        m[w] += 1  // This line works faster in Go 1.11
    }
    return m
}
```

100838 cmd/compile:
avoid mapaccess at m[k]=append(m[k]..

Avoid mapaccess at m[k]=append(m[k]...)

- Now the following code works faster:

```
func groupWordsByLen(words []string) map[int][]string {
    m := make(map[int][]string)
    for _, w := range words {
        wLen := len(w)
        // The following line works faster in Go 1.11
        m[wLen] = append(m[wLen], w)
    }
    return m
}
```

84055 cmd/compile/internal/ssa: update regalloc in loops

Update regalloc in loops

- Improves performance for the following code by using better register allocation inside loops:

```
for ... {
    if hard_case {
        call()
    }
    // simple case, without call
}
```

[100718](100718) cmd/compile:

specialize Move up to 79B on amd64

Specialize Move up to 79B on amd64

- Improves performance when copying structs and arrays with sizes from 32 bytes to 79 bytes
- Benchmark results:

```
CopyFat24-4    0.80ns ± 0%  0.40ns ± 0%   -50.00%  (p=0.001 n=8+9)
CopyFat32-4    2.01ns ± 0%  0.40ns ± 0%   -80.10%  (p=0.000 n=8+8)
CopyFat64-4    2.87ns ± 0%  0.40ns ± 0%   -86.07%  (p=0.000 n=8+10)
```

Bounds check elimination (BCE) improvements

What is bounds check elimination (BCE)?

- For **`a[i]`** go checks whether **`i`** exceeds **`a`** bounds by adding the following guard code before each **`a[i]`**:

```
if i < 0 || i >= len(a) {
    panic(“out of bounds access”)
}
```

- This code sometimes becomes redundant if a similar check already exists before **`a[i]`**
- Detection with subsequent removal of such guard code is called BCE
- BCE improves performance

BCE improvements

- Go 1.11 contains many patches for detecting and eliminating more bounds checks comparing to previous go versions. Here are a few of such patches:
    - 104037 cmd/compile: in prove, complete support for OpIsInBounds/OpIsSliceInBounds
    - 100277 cmd/compile: in prove, add transitive closure of relations
    - 100278 cmd/compile: in prove, infer unsigned relations while branching
    - 104038 cmd/compile: implement loop BCE in prove
    - 104041 cmd/compile: in prove, detect loops with negative increments
    - 105635 cmd/compile: teach prove to handle expressions like len(s)-delta
    - 109776 cmd/compile: simplify shifts using bounds from prove pass
    - 102601 cmd/compile: teach prove about relations between constants
    - 102602 cmd/compile: derive len/cap relations in factsTable.update

BCE improvements example 1

- The following code works faster in go 1.11:

```
func HasSuffix(s, suffix string) bool {
    return len(s) >= len(suffix) && s[len(s)-len(suffix):] == suffix
}
```

- Because the compiler became smart enough to eliminate redundant bounds check for `s[len(s)-len(suffix):]`

BCE improvements example 2

- The following code works faster in go 1.11:

```
x := 0
for i := len(a); i > 0; i-- {
    x += int(a[i-1])
}
return x
```

- Because the compiler is smart enough to detect that `a[i-1]` cannot go out of bounds inside the loop

BCE improvements example 3

- The following code works faster in go 1.11:

```
if i < 0 || i >= len(b) {
    return
}
for j := 0; j < i; j++ {
    b[j]++
}
```

- Because the compiler removes redundant bounds check at `b[j]`

105257 cmd/compile:
in escape analysis, propagate loop depth to field

What is escape analysis?

- Working with heap variables is usually slower than working with stack variables due to garbage collector overhead
- Initially all the variables go to heap, since it is unsafe to store certain variables on stack
- Go compiler tries detecting which variables don’t escape from the stack scope and may be safely put on stack
- This process is called **escape analysis**

Escape analysis improvements in Go 1.11

- Now the following code works faster:

```
type T struct { x int }

func f(t *T) {
        var y *int
        for i := 0; i < 2; i++ {
                y = &t.x
                *y = 1
        }
}
```

- Because the compiler puts `t` on stack instead of heap

80144 runtime:
use private futexes on Linux

Use private futexes on Linux

- Previously Go unnecessarily used shared futexes for synchronization primitives.
- Now code with heavy use of synchronization primitives should work slightly faster on Linux.
- See https://lwn.net/Articles/229668/ for differences between shared and private futexes.

Stack copy optimizations

Stack copy optimizations

- Go copies goroutine stack each time the stack must grow beyond its’ capacity
- Initial goroutine stack capacity is 2Kb
- When it outgrows 2Kb, go copies the stack into a new memory with bigger size
- Stack copies may become a bottleneck for a program with frequently created goroutines with deep call stacks. For instance, busy web server with a lot of middleware

Stack copy optimizations

- Go 1.11 contains the following optimizations for stack copy:
  - 94029 runtime: speed up stack copying a little
  - 104737 runtime: avoid calling adjustpointers unnecessarily
  - 108945 runtime: add fast version of getArgInfo
  - 109716 runtime: iterate over set bits in adjustpointers
  - 109001 runtime: allow inlining of stackmapdata
  - 104175 cmd/compile: shrink liveness maps
- These optimizations collectively improve stack copy performance by ~2X

math/big optimizations

math/big optimizations

- math/big is mostly used in public key cryptography
- Public key cryptography is used in https during session establishing
- So, math/big optimizations usually speed up https

99838 math/big:
reduce amount of copying
in Montgomery multiplication

What is Montgomery modular multiplication?

- Montgomery modular multiplication is an optimized algorithm for calculating **a * b (mod q)**, where **a** and **b** - big integers and **q** - big prime number
- See boring details at https://en.wikipedia.org/wiki/Montgomery_modular_multiplication
- Modular multiplication is the core of many public key cryptography algorithms

Reduce amount of copying in Montgomery multiplication

- Benchmark results:

```
name                    old time/op   new time/op   delta
RSA2048Decrypt-8        1.73ms ± 2%   1.55ms ± 2%   -10.19%  (p=0.000 n=10+10)
RSA2048Sign-8           2.17ms ± 2%   2.00ms ± 2%    -7.93%  (p=0.000 n=10+10)
3PrimeRSA2048Decrypt-8  1.10ms ± 2%   0.96ms ± 2%   -13.03%  (p=0.000 n=10+9)
```

99615 math/big:
implement Atkin's ModSqrt for 5 mod 8 primes

What is Atkin’s ModSqrt for 5 mod 8 algorithm?

- This is an algorithm for fast calculating **sqrt(x) (mod q)** for Atkin’s prime **q = 2^n+1 (mod 2^(n+1))** where **n = 2**
- **q=13** is the first such Atkin’s prime, since **13 (mod 8) = 5**.
- Boring details are available at https://ieeexplore.ieee.org/document/6504967/
- It looks like the algorithm covers 1/7 or 14% of all the primes
- So the patch speeds up **ModSqrt** calculation in 14% cases

Implement Atkin's ModSqrt for 5 mod 8 primes

- This improves performance for big.Int.ModSqrt in 14% cases:

```
ModSqrt231_5Mod8-4  1.03ms ± 2%  0.36ms ± 5%  -65.06%  (p=0.008 n=5+5)
```

105075 math/big: specialize Karatsuba implementation for squaring

What is squaring?

- Squaring is just **X * X** calculation. Simple. Isn’t it?
- This is simple for small integers. What about big integers with thouthands of decimal digits?
- Squaring becomes complicated and slows down for big integers
- Squaring is used in more complex algorithms such as **a pow b** calculation
- These algorithms are used in more powerful algorithms for public key cryptography
- There are special algorithms that may improve squaring performance for big integers

What is Karatsuba squaring?

- Karatsuba multiplication is an optimized multiplication algorithm for big integers
- Boring details may be found at https://www.hindawi.com/journals/jam/2014/107109/ .
- Squaring is a special case for multiplication
- So, Karatsuba squaring is just a special case for Karatsuba multiplication

Specialize Karatsuba implementation for squaring

- Improves performance for **x*x** where **x** - big integer:

```
NatSqr/500-4   81.9µs ± 1%  67.0µs ± 1%  -18.25%  (p=0.000 n=48+48)
NatSqr/800-4    161µs ± 1%   140µs ± 1%  -13.29%  (p=0.000 n=47+48)
NatSqr/1000-4   245µs ± 1%   207µs ± 1%  -15.17%  (p=0.000 n=49+49)
```

78755 math/big: implement Lehmer's extended GCD algorithm

Boring details about extended GCD algorithm

- Extended GCD is an extension to the Euclidean algorithm for finding the following numbers:
  - Greatest common divisor for two integers, **a** and **b**, i.e. **gcd(a, b)**. For instance gcd(10, 15)=5, because 10/5=2 and 15/5=3. *Remember, how you did this in elementary school? :)*
  - Integer coefficients, **x** and **y**, such that **ax + by = gcd(a,b)**. For instance, 10*(-1) + 15*1 = 5
- **GCD(a, b) = 1** if **a** and **b** have no common divisors, i.e. if they are co-prime
- Co-prime numbers are frequently used in public key cryptography
- Cryptographers like calculating **x = 1 / a (mod b)** such that **ax = 1 (mod b)** aka modular multiplicative inverse . This is essentially **x** coefficient from the extended GCD
- It is used in RSA algorithm, which may be used in https handshake

What is the Lehmer’s GCD algorithm?

- Lehmer’s GCD algorithm is a modern optimized version of the ancient algorithm from Euclide. Read boring details at https://en.wikipedia.org/wiki/Lehmer%27s_GCD_algorithm :)
- Lehmer’s GCD algorithm outperforms Euclidean algorithm for big integers
- Big integers are used in public key cryptography
- Public key cryptography is used in https
- So, Lehmer’s GCD algorithm should improve https performance

Improve performance for extended GCD algorithm

- Improves performance of **`big.Int.ModInverse`** used in https handshake
- Benchmark results:

```
name                            old time/op    new time/op    delta
GCD100x100/WithXY-4               19.3µs ± 0%     3.9µs ± 1%  -79.58%  (p=0.008 n=5+5)
GCD100x1000/WithXY-4              22.8µs ± 1%     7.5µs ±10%  -67.00%  (p=0.008 n=5+5)
GCD100x10000/WithXY-4             75.1µs ± 2%    30.5µs ± 2%  -59.38%  (p=0.008 n=5+5)
GCD100x100000/WithXY-4             542µs ± 2%     267µs ± 2%  -50.79%  (p=0.008 n=5+5)
GCD1000x1000/WithXY-4              329µs ± 0%      42µs ± 1%  -87.12%  (p=0.008 n=5+5)
GCD1000x10000/WithXY-4             607µs ± 9%     123µs ± 1%  -79.70%  (p=0.008 n=5+5)
GCD1000x100000/WithXY-4           3.64ms ± 1%    0.93ms ± 1%  -74.41%  (p=0.016 n=4+5)
GCD10000x10000/WithXY-4           7.44ms ± 1%    1.00ms ± 0%  -86.58%  (p=0.008 n=5+5)
GCD10000x100000/WithXY-4          37.3ms ± 0%     7.3ms ± 1%  -80.45%  (p=0.008 n=5+5)
GCD100000x100000/WithXY-4          505ms ± 1%      56ms ± 1%  -88.92%  (p=0.008 n=5+5)
```

74851 math/big: speed-up addMulVVW on amd64

Speed up addMulVVW on amd64

- This is arch-specific patch for **GOARCH=amd64**, which improves performance for **a += b * c** calculations on big integers
- Improves performance for https handshake:

```
RSA2048Decrypt-8         1.61ms ± 1%  1.38ms ± 1%  -14.13%  (p=0.000 n=10+10)
RSA2048Sign-8            1.93ms ± 1%  1.70ms ± 1%  -11.86%  (p=0.000 n=10+10)
3PrimeRSA2048Decrypt-8    932µs ± 0%   828µs ± 0%  -11.15%  (p=0.000 n=10+10)

HandshakeServer/RSA-8   901µs ± 1%    777µs ± 0%  -13.70%  (p=0.000 n=10+8)
HandshakeServer/ECDHE-8 1.01ms ± 1%   0.90ms ± 0% -11.53%  (p=0.000 n=10+9)
```

Standard library optimizations

97255 strings:
speed-up replace for byteStringReplacer

Improve performance for strings.Replace

- Benchmark results:

```
Escape-6           34.2µs ± 2%  20.8µs ± 2%  -39.06%  (p=0.000 n=10+10)
EscapeNone-6       7.04µs ± 1%  1.05µs ± 0%  -85.03%  (p=0.000 n=10+10)


ByteStringMatch-6    1.59µs ± 2%    1.17µs ± 2%  -26.35%  (p=0.000 n=10+10)
HTMLEscapeNew-6       390ns ± 2%     337ns ± 2%  -13.62%  (p=0.000 n=10+10)
HTMLEscapeOld-6       621ns ± 2%     603ns ± 2%   -2.95%  (p=0.000 n=10+9)
```

101715 regexp:
use sync.Pool to cache
regexp.machine objects

Use sync.Pool to cache regexp.machine objects

- Removes lock contention when a single regexp is used from concurrently running goroutines
- Benchmark results:

```
BenchmarkMatchParallelShared-4   361    77.9          -78.42%
```

102235 compress/flate: optimize huffSym

Improve gzip decompression speed

- Benchmark results:

```
name                              old time/op    new time/op    delta
Decode/Digits/Huffman/1e4-6          278µs ± 1%     240µs ± 2%  -13.72%  (p=0.000 n=10+10)
Decode/Digits/Huffman/1e5-6         2.38ms ± 1%    2.05ms ± 1%  -14.12%  (p=0.000 n=10+10)
Decode/Digits/Huffman/1e6-6         23.4ms ± 1%    19.9ms ± 0%  -14.69%  (p=0.000 n=9+9)
Decode/Twain/Huffman/1e4-6           316µs ± 2%     267µs ± 3%  -15.30%  (p=0.000 n=9+10)
Decode/Twain/Huffman/1e5-6          2.62ms ± 0%    2.22ms ± 0%  -15.24%  (p=0.000 n=10+10)
Decode/Twain/Huffman/1e6-6          25.7ms ± 1%    21.8ms ± 0%  -15.19%  (p=0.000 n=10+10)
Decode/Twain/Compression/1e4-6       272µs ± 1%     250µs ± 4%   -8.20%  (p=0.000 n=9+10)
Decode/Twain/Compression/1e5-6      2.01ms ± 0%    1.84ms ± 1%   -8.57%  (p=0.000 n=9+10)
Decode/Twain/Compression/1e6-6      19.1ms ± 0%    17.4ms ± 1%   -8.75%  (p=0.000 n=9+10)
```

107715 net:
add support for splice(2)
in (*TCPConn).ReadFrom on Linux

What is splice(2)?

- Splice is a Linux system call for fast data transfer. See `` `man 2 splice` `` or http://man7.org/linux/man-pages/man2/splice.2.html
- Splice avoids data copy between kernel space and user space. This is sometimes called **zero-copy**
- See https://lwn.net/Articles/178199/ for boring details about splice internals

Add splice(2) in TCPConn.ReadFrom on Linux

- Go 1.11 transparently uses splice when **io.Copy** is used for copying data between two tcp connections
- See details at https://acln.ro/articles/go-splice
- Benchmark results:

benchmark	old ns/op	new ns/op	delta
BenchmarkTCPReadFrom/8192-4	5219	4779	-8.43%
BenchmarkTCPReadFrom/16384-4	8708	8008	-8.04%
BenchmarkTCPReadFrom/32768-4	16349	14973	-8.42%
BenchmarkTCPReadFrom/65536-4	35246	27406	-22.24%
BenchmarkTCPReadFrom/131072-4	72920	52382	-28.17%
BenchmarkTCPReadFrom/262144-4	149311	95094	-36.31%
BenchmarkTCPReadFrom/524288-4	306704	181856	-40.71%
BenchmarkTCPReadFrom/1048576-4	674174	357406	-46.99%

116378 net/http:
remove an allocation in ServeMux

Remove an allocation in ServeMux

- This is just classical optimization based on reducing memory allocations
- Improves performance for **net/http.ServeMux**:

```
name                    old time/op    new time/op    delta
ServeMux_SkipServe-4      74.2µs ± 2%    60.6µs ± 1%   -18.31%   (p=0.000 n=10+9)

name                    old alloc/op   new alloc/op   delta
ServeMux_SkipServe-4      2.62kB ± 0%   0.00kB ±NaN%  -100.00%  (p=0.000 n=10+10)

name                    old allocs/op  new allocs/op  delta
ServeMux_SkipServe-4         180 ± 0%        0 ±NaN%  -100.00%  (p=0.000 n=10+10)
```

Arch-specific optimizations

Arch-specific optimizations

- Go may build programs for many architectures, not only for **GOARCH=amd64**
- **GOARCH=arm** becomes popular. It is used in smartphones and in energy effective servers. See https://blog.cloudflare.com/arm-takes-wing/ .
- Go 1.11 has patches significantly optimizing performance for **GOARCH=arm**

Performance patches for arm

- 98095 runtime: use vDSO for clock_gettime on linux/arm
  - This improves **time.Now()** performance by 2.5x
  - The patch substitutes system call with access to vDSO - a shared memory visible to all the processes, where system time may be read

- 83799 runtime: improve arm64 memmove implementation
  - **Memmove** is the most frequently executed code in the majority of programs
  - **Memmove** just copies byte slice from one place to another
  - The patch improves **memove** performance by up to 2x

Performance patches for arm

- There are pending patches for arm, which may still be merged into go 1.11:
    - *99755 crypto/elliptic: implement P256 for arm64*. Improves https handshake performance **by up to 10x**
    - *81877 cmd/compile: improve atomic add intrinsics with ARMv8.1 new instruction*. Improves **atomic.Add*** performance **by up to 50X**
    - *107298 crypto/aes: implement AES-GCM AEAD for arm64.* Improves https performance **by up to 10x**

There are many other (performance) patches went into Go 1.11…

Interested readers may go to

https://go-review.googlesource.com/q/status:merged+project:go

Summary

Summary

- Each new Go version contains performance improvements
- Programs built with Go 1.11 should work faster comparing to builds with previous Go versions
- The future Go versions will be optimized further
- Everybody may take part in this process
- This is fun and is easier than optimizing gcc / llvm, since the majority of the Go compiler, runtime and standard library is written in Go

Useful links

- https://go-review.googlesource.com/q/status:open - patches (with review comments) that may go into the next Go version
- https://go-review.googlesource.com/q/status:merged - patches recently merged into Go
- https://golang.org/doc/contribute.html - how to contribute to Go. It’s easy - just do it! :)

Questions?