Micro-Optimizing Go Code

George Tankersley
@gtank__

Code: https://github.com/gtank/blake2s

This is a story of getting a little carried away

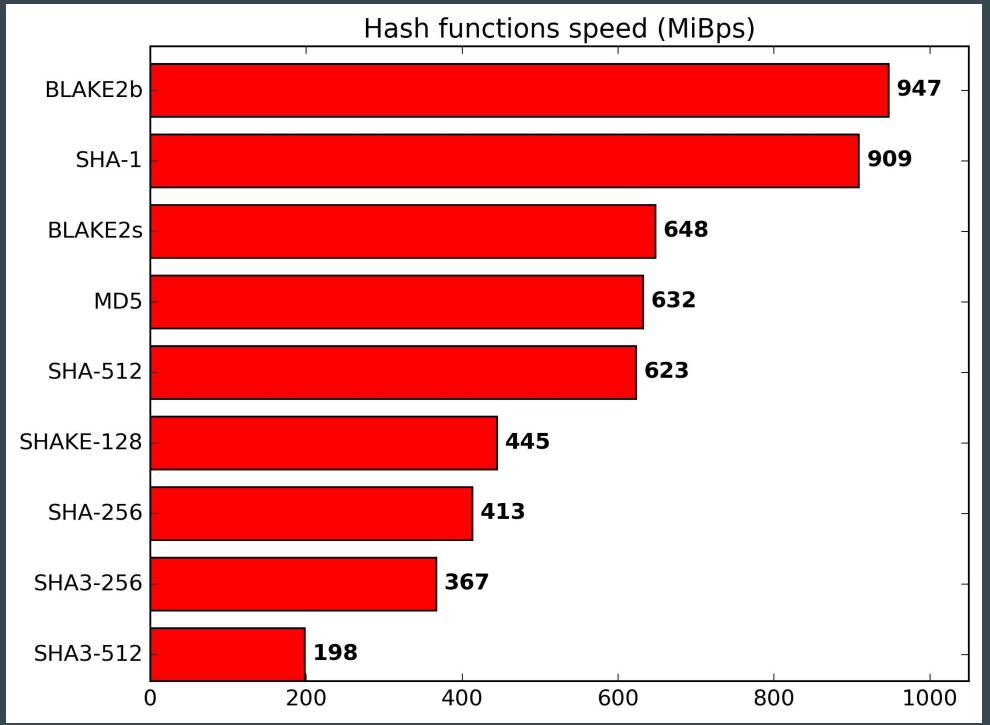
```
name           old time/op    new time/op     delta
Hash8Bytes-4     971ns ± 4%     392ns ± 1%   -59.66%  (p=0.008)
Hash1K-4        10.2µs ±11%     3.1µs ± 3%   -69.26%  (p=0.008)
Hash8K-4        77.0µs ± 4%    23.4µs ± 1%   -69.60%  (p=0.008)

name           old speed      new speed       delta
Hash8Bytes-4  8.24MB/s ± 4%  20.41MB/s ± 1%  +147.65%  (p=0.008)
Hash1K-4       101MB/s ±10%    327MB/s ± 3%  +224.31%  (p=0.008)
Hash8K-4       106MB/s ± 4%    350MB/s ± 1%  +228.73%  (p=0.008)
```

BLAKE2

BLAKE2 is *awesome*

From the paper:

- **Faster than MD5**
- Immune to length extension attacks
- FEATURES! Parallelism, tree hashing, prefix-MAC, personalization, etc



Single-core serial implementation, Skylake

BLAKE2 is *under-specified*

No one implements all of it. Not even RFC7693:

> Note: [The BLAKE2 paper] defines additional variants of BLAKE2 with features such as salting, personalized hashes, and tree hashing. These OPTIONAL features use fields in the parameter block that are not defined in this document.

Two cryptographers implementing an unspecified algorithm.

Photo by Zach Weinersmith, circa 2009.

The BLAKE2 Algorithm (Abridged)

1. Initialize parameters
2. Split input data into fixed-size blocks
3. Scramble the bits around
4. Update internal state
5. Finalize & output

Hash functions in Go

```
type Hash interface {
    // Write (via the embedded io.Writer) adds more data to the 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
}
```

Hash functions in Go

```
type Hash interface {
    // Write (via the embedded io.Writer) adds more data to the 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
}
```

Block padding. Tree modes?

Mutating finalize()

Needs key

Arbitrary parameter but affects hash output

Differs by BLAKE2 variant

Benchmarking

Tools of the trade

go bench	https://dave.cheney.net/2013/06/30/how-to-write-benchmarks-in-go
benchstat	https://godoc.org/golang.org/x/perf/cmd/benchstat
pprof	https://golang.org/pkg/runtime/pprof/

Tools of the trade

go bench https://dave.cheney.net/2013/06/30/how-to-write-benchmarks-in-go

benchstat https://godoc.org/golang.org/x/perf/cmd/benchstat

pprof https://golang.org/pkg/runtime/pprof/

And this awful bash one-liner:

```
DATE=`date -u +'%s' | tr -d '\n'`; BRANCH=`git
rev-parse --abbrev-ref HEAD`; for i in {1..8}; do
go test -bench . >> benchmark-$BRANCH-$DATE; done
```

Benchmarks

- Go has built-in support for benchmarking.
- You've seen `testing.T`, this is `testing.B`.
- I usually put benchmarks in my test files.

The benchmarks I'm using are here:

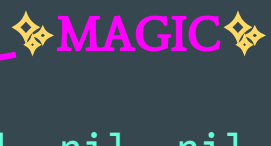
https://github.com/gtank/blake2s/blob/master/blake2s_test.go

Benchmarks

```
var emptyBuf = make([]byte, 8192)

func benchmarkHashSize(b *testing.B, size int) {
    b.SetBytes(int64(size))
    sum := make([]byte, 32)
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        digest, _ := NewDigest(nil, nil, nil, 32)
        digest.Write(emptyBuf[:size])
        digest.Sum(sum[:0])
    }
}

func BenchmarkHash8Bytes(b *testing.B) {
    benchmarkHashSize(b, 8)
}
```

Benchmarks

```
var emptyBuf = make([]byte, 8192)

func benchmarkHashSize(b *testing.B, size int) {
    b.SetBytes(int64(size))
    sum := make([]byte, 32)
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        digest, _ := NewDigest(nil, nil, nil, 32)
        digest.Write(emptyBuf[:size])
        digest.Sum(sum[:0])
    }
}

func BenchmarkHash8Bytes(b *testing.B) {
    benchmarkHashSize(b, 8)
}
```



$ go test -bench .

```
goos: linux
goarch: amd64
pkg: github.com/gtank/blake2
BenchmarkHash8Bytes-4   2000000     859 ns/op     9.31 MB/s
BenchmarkHash1K-4        200000    8822 ns/op   116.06 MB/s
BenchmarkHash8K-4         20000   66617 ns/op   122.97 MB/s
PASS
ok     github.com/gtank/blake2 6.613s
```

$ go test -bench .

```
goos: linux
goarch: amd64
pkg: github.com/gtank/blake2
BenchmarkHash8Bytes-4   2000000     859 ns/op     9.31 MB/s
BenchmarkHash1K-4        200000    8822 ns/op   116.06 MB/s
BenchmarkHash8K-4         20000   66617 ns/op   122.97 MB/s
PASS
ok     github.com/gtank/blake2 6.613s
```

pprof

(pprof) top5

```
Showing top 5 nodes out of 39
      flat  flat%   sum%        cum   cum%
    3480ms 54.12% 54.12%     5130ms 79.78%  blake2.(*Digest).compress
    1320ms 20.53% 74.65%     1600ms 24.88%  github.com/gtank/blake2.g
     280ms  4.35% 79.00%      280ms  4.35%  math/bits.RotateLeft32
     220ms  3.42% 82.43%      780ms 12.13%  runtime.mallocgc
     100ms  1.56% 83.98%      650ms 10.11%  runtime.makeslice
```

What’s their relationship though?

The round function, g()

```
func g(a, b, c, d, m0, m1 uint32) (uint32, uint32, uint32,
uint32) {
    a = a + b + m0
    d = bits.RotateLeft32(d^a, -16)
    c = c + d
    b = bits.RotateLeft32(b^c, -12)
    a = a + b + m1
    d = bits.RotateLeft32(d^a, -8)
    c = c + d
    b = bits.RotateLeft32(b^c, -7)
    return a, b, c, d
}
```

Inlining

Inlining

Inlining is copying the body of a function into the body of the caller.

Avoids function call overhead, which is substantial in Go.

Tradeoff between performance and binary size.

Inlining

The ***inliner*** is a component of the compiler with no* manual control.

It uses an AST visitor to calculate a complexity score vs a complexity budget.

Chasing the inliner is a flavor of optimization unique to Go.

*Except unofficial pragmas

Inlining

Functions accrue +1 cost for each node in the instruction tree

Slices are expensive! A slice node is +2 or +3 depending.

Function calls OK in most cases if we have budget for it.
But a call is +2 regardless.

Inlining

Some things are hard stops:

- Nonlinear control flow - for, range, select, break, defer, type switch
- Recover (but not panic)
- Certain runtime funcs and all non-intrinsic assembly [#17373]

Full details (as of go1.11) in inl.go

Results: $ benchstat baseline inlinable_g

```
name  old time/opnew time/opdelta
Hash8B-4  772ns  ± 2%  574ns  ± 0%  -25.71%  (p=0.000)
Hash1K-4  8.50µs ± 3%  5.20µs ± 2%  -38.80%  (p=0.000)
Hash8K-4  65.8µs ± 4%  39.3µs ± 2%  -40.25%  (p=0.000)

name  old speed  new speed  delta
Hash8B-4  10.4MB/s ± 2%  13.9MB/s ± 0%  +34.52%  (p=0.000)
Hash1K-4  121MB/s  ± 3%   197MB/s ± 2%  +63.36%  (p=0.000)
Hash8K-4  125MB/s  ± 4%   209MB/s ± 2%  +67.33%  (p=0.000)
```

How do we check?

$ go test -gcflags="-m=2" 2>&1 | grep "too complex"

```
[...]
./blake2s.go:272:6: cannot inline g: function too
complex: cost 133 exceeds budget 80
./blake2s.go:284:6: cannot inline NewDigest:
function too complex: cost 332 exceeds budget 80
./blake2s.go:340:6: cannot inline (*Digest).Sum:
function too complex: cost 100 exceeds budget 80
[...]
```

The round function, g()

```
func g(a, b, c, d, m0, m1 uint32) (uint32, uint32, uint32, uint32)
{
    a = a + b + m0
    d = bits.RotateLeft32(d^a, -16)
    c = c + d
    b = bits.RotateLeft32(b^c, -12)
    a = a + b + m1
    d = bits.RotateLeft32(d^a, -8)
    c = c + d
    b = bits.RotateLeft32(b^c, -7)
    return a, b, c, d
}
```

The round function, g()

```
func g(a, b, c, d, m0, m1 uint32) (uint32, uint32, uint32, uint32)
{
    a = a + b + m0
    d = ((d ^ a) >> 16) | ((d ^ a) << (32 - 16))
    c = c + d
    b = ((b ^ c) >> 12) | ((b ^ c) << (32 - 12))
    a = a + b + m1
    d = ((d ^ a) >> 8) | ((d ^ a) << (32 - 8))
    c = c + d
    b = ((b ^ c) >> 7) | ((b ^ c) << (32 - 7))
    return a, b, c, d
}
```

$ go test -gcflags="-m=2" 2>&1 | grep "too complex"

```
[...]
./blake2s.go:270:6: cannot inline g: function too
complex: cost 81 exceeds budget 80
./blake2s.go:282:6: cannot inline NewDigest:
function too complex: cost 332 exceeds budget 80
./blake2s.go:338:6: cannot inline (*Digest).Sum:
function too complex: cost 100 exceeds budget 80
[...]
```

The round function, g()

```
func g(a, b, c, d, m0, m1 uint32) (uint32, uint32, uint32, uint32)
{
    a = a + b + m0
    d = ((d ^ a) >> 16) | ((d ^ a) << (32 - 16))
    c = c + d
    b = ((b ^ c) >> 12) | ((b ^ c) << (32 - 12))
    a = a + b + m1
    d = ((d ^ a) >> 8) | ((d ^ a) << (32 - 8))
    c = c + d
    b = ((b ^ c) >> 7) | ((b ^ c) << (32 - 7))
    return a, b, c, d
}
```

Change the API!

```
func g(a, b, c, d, m1 uint32) (uint32, uint32, uint32, uint32)
{
    // a = a + b + m0
    d = ((d ^ a) >> 16) | ((d ^ a) << (32 - 16))
    c = c + d
    b = ((b ^ c) >> 12) | ((b ^ c) << (32 - 12))
    a = a + b + m1
    d = ((d ^ a) >> 8) | ((d ^ a) << (32 - 8))
    c = c + d
    b = ((b ^ c) >> 7) | ((b ^ c) << (32 - 7))
    return a, b, c, d
}
```

Change the API!

```
func g(a, b, c, d, m1 uint32) (uint32, uint32, uint32, uint32)
{
    // a = a + b + m0
    d = ((d ^ a) >> 16) | ((d ^ a) << (32 - 16))
    c = c + d
    b = ((b ^ c) >> 12) | ((b ^ c) << (32 - 12))
    a = a + b + m1
    d = ((d ^ a) >> 8) | ((d ^ a) << (32 - 8))
    c = c + d
    b = ((b ^ c) >> 7) | ((b ^ c) << (32 - 7))
    return a, b, c, d
}
```

$ go test -gcflags="-m=2" 2>&1 | grep "can inline"

```
./blake2s.go:270:6: can inline g with cost 74 as:
func(uint32, uint32, uint32, uint32, uint32)
(uint32, uint32, uint32, uint32) { d = (d ^ a) >>
16 | (d ^ a) << (32 - 16); c = c + d; b = (b ^ c)
>> 12 | (b ^ c) << (32 - 12); a = a + b + m1; d =
(d ^ a) >> 8 | (d ^ a) << (32 - 8); c = c + d; b
= (b ^ c) >> 7 | (b ^ c) << (32 - 7); return a,
b, c, d }
```

Under budget does not mean faster!

Don’t be this guy*

```
// This function is written to ensure it inlines.
func mul64x64(lo, hi, a, b uint64) (ol uint64, oh uint64) {
	t1 := (a>>32)*(b&0xFFFFFFFF) + ((a & 0xFFFFFFFF) * (b & 0xFFFFFFFF) >> 32)
	t2 := (a&0xFFFFFFFF)*(b>>32) + (t1 & 0xFFFFFFFF)
	ol = (a * b) + lo
	cmp := ol < lo
	oh = hi + (a>>32)*(b>>32) + t1>>32 + t2>>32 + uint64(*(*byte)(unsafe.Pointer(&cmp)))
	return
}
```

*me

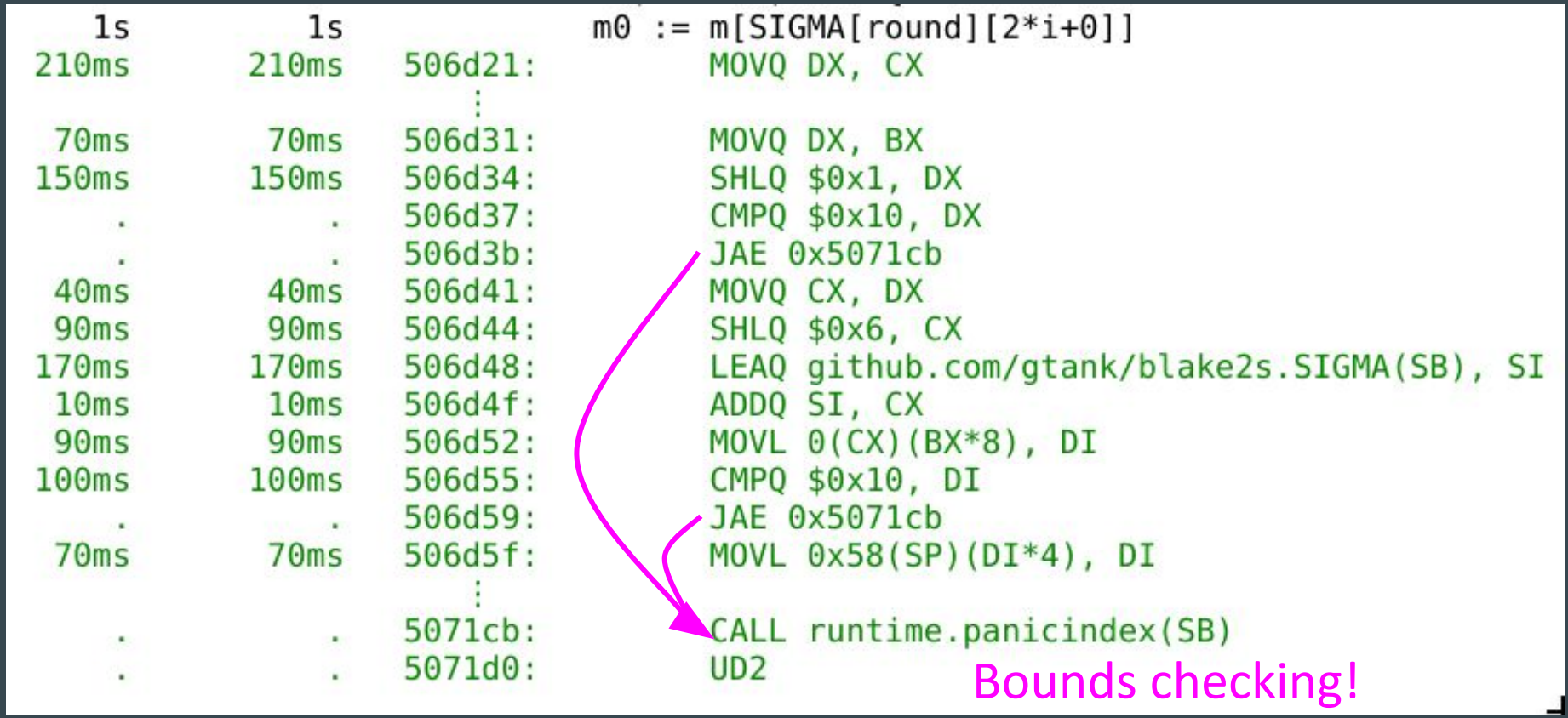
What’s next?

Back to pprof

```
(pprof) top5
Showing top 5 nodes out of 41
  flat  flat%   sum%    cum   cum%
3650ms 61.55% 61.55% 4700ms 79.26%  blake2s.(*Digest).compress
1010ms 17.03% 78.58% 1010ms 17.03%  blake2s.g (inline)
 230ms  3.88% 82.46%  790ms 13.32%  runtime.mallocgc
 110ms  1.85% 84.32%  110ms  1.85%  runtime.memclrNoHeapPointers
 110ms  1.85% 86.17%  110ms  1.85%  runtime.nextFreeFast (inline)
```

We need a more granular view...

What's going on here?



runtime.panicindex

```
$ go run bce.go
panic: runtime error: index out of range

goroutine 1 [running]:
main.demo(...)
        /home/gtank/bce.go:9 main.main()
        /home/gtank/bce.go:5 +0x11
exit status 2
```

Another view: $ go test -gcflags="-d=ssa/check_bce/debug=1"

```
[...]
./blake2s.go:199:11: Found IsInBounds
./blake2s.go:199:24: Found IsInBounds
./blake2s.go:200:11: Found IsInBounds
./blake2s.go:200:24: Found IsInBounds
```

```
197 ▸   for round := 0; round < RoundCount; round++ {
198 ▸   ▸   for i := 0; i < 8; i++ {
199 ▸   ▸   ▸   m0 := m[SIGMA[round][2*i+0]]
200 ▸   ▸   ▸   m1 := m[SIGMA[round][2*i+1]]
201 ▸   ▸   ▸   switch i {
```

Bounds check elimination, normally

```
func (bigEndian) PutUint32(b []byte, v uint32) {
    _ = b[3] // early bounds check to guarantee safety below
    b[0] = byte(v >> 24)
    b[1] = byte(v >> 16)
    b[2] = byte(v >> 8)
    b[3] = byte(v)
}
```

Optimizing that table lookup

Combination of several “old-school” optimization techniques:

- Propagate constants
- Unroll loops
- Reuse previously-allocated local variables

In pursuit of a specific thing:

- Bounds-Check Elimination (further reading)

Results - worth it?

Much, much faster!

Macros would be nice here.

```
// This round structure is several steps removed from the spec and
// reference implementation. We unrolled the loops and calculated the
// offsets from the permutation table entry for each round, then directly
// mapped it to the correct word of the input block. This is a tradeoff:
// the doubly-indirect lookups were horrible for performance, but it's not
// at all obvious what this code is doing anymore.

// Round 0 w/ precomputed permutation offsets
v0, v4, v8, v12 = g(v0+v4+m0, v4, v8, v12, m1)
v1, v5, v9, v13 = g(v1+v5+m2, v5, v9, v13, m3)
v2, v6, v10, v14 = g(v2+v6+m4, v6, v10, v14, m5)
v3, v7, v11, v15 = g(v3+v7+m6, v7, v11, v15, m7)

v0, v5, v10, v15 = g(v0+v5+m8, v5, v10, v15, m9)
v1, v6, v11, v12 = g(v1+v6+m10, v6, v11, v12, m11)
v2, v7, v8, v13 = g(v2+v7+m12, v7, v8, v13, m13)
v3, v4, v9, v14 = g(v3+v4+m14, v4, v9, v14, m15)

// Round 1
v0, v4, v8, v12 = g(v0+v4+m14, v4, v8, v12, m10)
v1, v5, v9, v13 = g(v1+v5+m4, v5, v9, v13, m8)
v2, v6, v10, v14 = g(v2+v6+m9, v6, v10, v14, m15)
v3, v7, v11, v15 = g(v3+v7+m13, v7, v11, v15, m6)

v0, v5, v10, v15 = g(v0+v5+m1, v5, v10, v15, m12)
v1, v6, v11, v12 = g(v1+v6+m0, v6, v11, v12, m2)
v2, v7, v8, v13 = g(v2+v7+m11, v7, v8, v13, m7)
v3, v4, v9, v14 = g(v3+v4+m5, v4, v9, v14, m3)

// Round 2
v0, v4, v8, v12 = g(v0+v4+m11, v4, v8, v12, m8)
v1, v5, v9, v13 = g(v1+v5+m12, v5, v9, v13, m0)
v2, v6, v10, v14 = g(v2+v6+m5, v6, v10, v14, m2)
v3, v7, v11, v15 = g(v3+v7+m15, v7, v11, v15, m13)
```

Results: $ benchstat inlinable_g eliminate_bounds_checks

```
name           old time/op    new time/op    delta
Hash8Bytes-4     574ns ± 0%     420ns ± 2%  -26.90%  (p=0.000)
Hash1K-4        5.20µs ± 2%    2.91µs ± 3%  -44.09%  (p=0.000)
Hash8K-4        39.3µs ± 2%    21.5µs ± 4%  -45.16%  (p=0.000)

name           old speed      new speed      delta
Hash8Bytes-4  13.9MB/s ± 0%  19.1MB/s ± 2%  +36.81%  (p=0.000)
Hash1K-4       197MB/s ± 2%   352MB/s ± 3%  +78.87%  (p=0.000)
Hash8K-4       209MB/s ± 2%   380MB/s ± 4%  +82.40%  (p=0.000)
```

One more bounds check...?

The internal hash state is a slice, but it's always of fixed size.

Can we eliminate these?

Well, not as we expect.

SSA bounds check output

```
./blake2s.go:308:14: Found IsInBounds
./blake2s.go:309:14: Found IsInBounds
./blake2s.go:310:14: Found IsInBounds
./blake2s.go:311:14: Found IsInBounds
./blake2s.go:312:14: Found IsInBounds
./blake2s.go:313:14: Found IsInBounds
./blake2s.go:314:14: Found IsInBounds
```

Corresponding to these lines in compress():

```
307 ▸    d.h[0] = d.h[0] ^ v0 ^ v8
308 ▸    d.h[1] = d.h[1] ^ v1 ^ v9
309 ▸    d.h[2] = d.h[2] ^ v2 ^ v10
310 ▸    d.h[3] = d.h[3] ^ v3 ^ v11
311 ▸    d.h[4] = d.h[4] ^ v4 ^ v12
312 ▸    d.h[5] = d.h[5] ^ v5 ^ v13
313 ▸    d.h[6] = d.h[6] ^ v6 ^ v14
314 ▸    d.h[7] = d.h[7] ^ v7 ^ v15
```

Sure, why not?

Replace slice with array.

Compiler is satisfied.

No more runtime bounds checks!

Side effect: makes an explicit copy an implicit one.

```
diff --git a/blake2s.go b/blake2s.go
index 7b0ccd0..a349bcf 100644
--- a/blake2s.go
+++ b/blake2s.go
@@ -69,7 +69,7 @@ func (p *parameterBlock) Marshal() []byte {

 // The internal state of the BLAKE2s algorithm.
 type Digest struct {
-       h      []uint32
+       h      [8]uint32
        t0, t1 uint32
        f0, f1 uint32

@@ -94,7 +94,7 @@ func initFromParams(p *parameterBlock) *Digest {
        h7 := IV7 ^ binary.LittleEndian.Uint32(paramBytes[28:32])

        d := &Digest{
-               h:    []uint32{h0, h1, h2, h3, h4, h5, h6, h7},
+               h:    [8]uint32{h0, h1, h2, h3, h4, h5, h6, h7},
                buf:  make([]byte, 0, BlockBytes),
                size: int(p.DigestSize),
        }
@@ -154,10 +154,6 @@ func (d *Digest) compress() error {
                return errors.New("blake2s: tried to compress when buffer wasn't full")
        }

-       if len(d.h) != 8 {
-               return errors.New("blake2s: internal hash state too large")
-       }
-
        // Split the buffer into 16x32-bit words.
        m0 := binary.LittleEndian.Uint32(d.buf[0*4 : 0*4+4])
        m1 := binary.LittleEndian.Uint32(d.buf[1*4 : 1*4+4])
@@ -328,8 +324,6 @@ func (d *Digest) finalize() ([]byte, error) {
        dCopy := *d
        dCopy.buf = make([]byte, cap(d.buf)) // want this zero-padded to BlockSize anyway
        copy(dCopy.buf, d.buf)
-       dCopy.h = make([]uint32, len(d.h))
-       copy(dCopy.h, d.h)

        // increment counter by size of pending input before padding
        dCopy.t0 += uint32(len(d.buf))
```

Results: $ benchstat eliminate_checks use_fixed_array

```
name           old time/op    new time/op    delta
Hash8Bytes-4     420ns ± 2%     373ns ± 2%  -11.03%  (p=0.000)
Hash1K-4        2.91µs ± 3%    2.87µs ± 3%     ~     (p=0.130)
Hash8K-4        21.5µs ± 4%    21.6µs ± 3%     ~     (p=0.536)

name           old speed      new speed      delta
Hash8Bytes-4  19.1MB/s ± 2%  21.4MB/s ± 2%  +12.37%  (p=0.000)
Hash1K-4       352MB/s ± 3%   357MB/s ± 3%     ~     (p=0.130)
Hash8K-4       380MB/s ± 4%   379MB/s ± 3%     ~     (p=0.536)
```

Results: $ benchstat eliminate_checks use_fixed_array

```
name           old time/op    new time/op    delta
Hash8Bytes-4     420ns ± 2%     373ns ± 2%  -11.03%  (p=0.000)
Hash1K-4        2.91µs ± 3%    2.87µs ± 3%     ~     (p=0.130)
Hash8K-4        21.5µs ± 4%    21.6µs ± 3%     ~     (p=0.536)

name           old speed      new speed      delta
Hash8Bytes-4  19.1MB/s ± 2%  21.4MB/s ± 2%  +12.37%  (p=0.000)
Hash1K-4       352MB/s ± 3%   357MB/s ± 3%     ~     (p=0.130)
Hash8K-4       380MB/s ± 4%   379MB/s ± 3%     ~     (p=0.536)
```

huh?

Benchmark

```
var emptyBuf = make([]byte, 8192)

func benchmarkHashSize(b *testing.B, size int) {
    b.SetBytes(int64(size))
    sum := make([]byte, 32)
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        digest, _ := NewDigest(nil, nil, nil, 32)
        digest.Write(emptyBuf[:size])
        digest.Sum(sum[:0])
    }
}
```

Allocations and copies

```
@@ -328,8 +324,6 @@ func (d *Digest) finalize() ([]byte, error) {
        dCopy := *d
        dCopy.buf = make([]byte, cap(d.buf)) // want this zero-pad
        copy(dCopy.buf, d.buf)
-       dCopy.h = make([]uint32, len(d.h))
-       copy(dCopy.h, d.h)
```

finalize() runs once each time you calculate a BLAKE2 sum.

We eliminated a make/copy there.

Allocations and copies

```
(pprof) top10
Showing nodes accounting for 5.44s, 87.46% of 6.22s total
Dropped 61 nodes (cum <= 0.03s)
Showing top 10 nodes out of 45
      flat  flat%   sum%        cum   cum%
     3.46s 55.63% 55.63%      3.46s 55.63%  github.com/gtank/blake2s.g (inline)
     0.82s 13.18% 68.81%      4.33s 69.61%  github.com/gtank/blake2s.(*Digest).compress
     0.37s  5.95% 74.76%      0.96s 15.43%  runtime.mallocgc
     0.17s  2.73% 77.49%      0.17s  2.73%  runtime.nextFreeFast (inline)
     0.16s  2.57% 80.06%      3.48s 55.95%  github.com/gtank/blake2s.(*Digest).Write
     0.12s  1.93% 81.99%      1.42s 22.83%  github.com/gtank/blake2s.(*Digest).finalize
     0.11s  1.77% 83.76%      0.83s 13.34%  runtime.makeslice
     0.10s  1.61% 85.37%      0.10s  1.61%  runtime.memmove
     0.07s  1.13% 86.50%      0.26s  4.18%  github.com/gtank/blake2s.(*parameterBlock).Marsha
     0.06s  0.96% 87.46%      0.06s  0.96%  encoding/binary.littleEndian.Uint32 (inline)
```

Allocations and copies

```
$ ag "make\(" blake2s.go
55: buf := make([]byte, 32)
98: buf:  make([]byte, 0, BlockSize),
325:dCopy.buf = make([]byte, cap(d.buf)) // want zero-padded to BlockSize anyway
343:out := make([]byte, dCopy.size)
390:params.Salt = make([]byte, SaltLength)
399:params.Personalization = make([]byte, SeparatorLength)
414:keyBuf := make([]byte, BlockSize)
```

Allocations and copies

```
$ ag "make\(" blake2s.go
55: buf := make([]byte, 32)
98: buf:  make([]byte, 0, BlockSize),
325:dCopy.buf = make([]byte, cap(d.buf)) // want zero-padded to BlockSize anyway
343:out := make([]byte, dCopy.size)
390:params.Salt = make([]byte, SaltLength)
399:params.Personalization = make([]byte, SeparatorLength)
414:keyBuf := make([]byte, BlockSize)
```

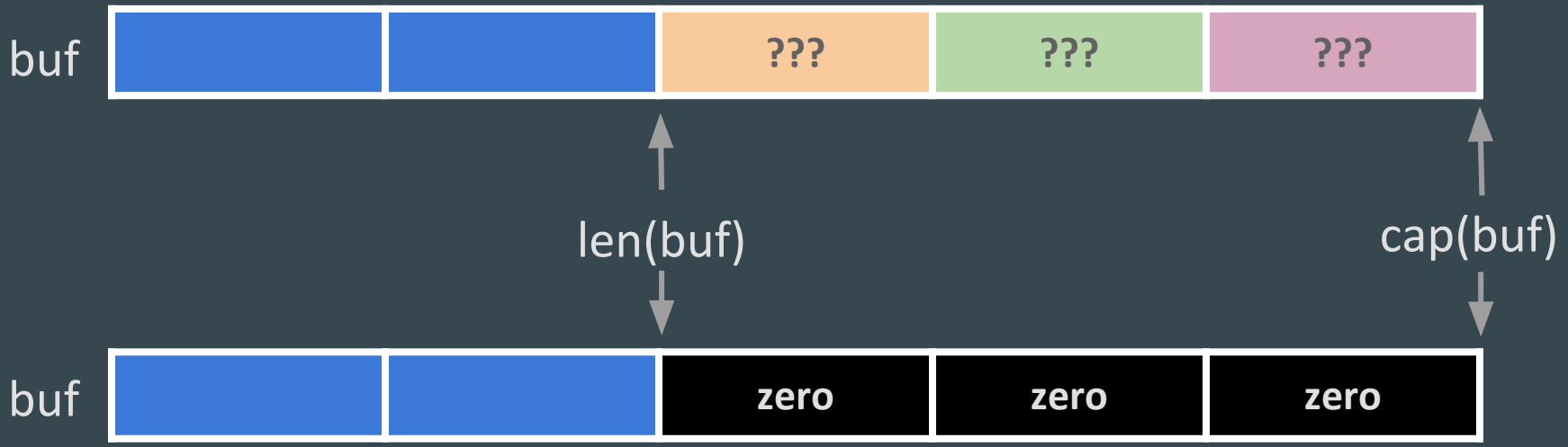
Copy ↓

```
func (d *Digest) finalize() ([]byte, error)
    if d.f0 != 0 {
        return nil, errors.New("blake2s: tr
    }

    // make copies of everything
    dCopy := *d
    dCopy.buf = make([]byte, cap(d.buf)) //
    copy(dCopy.buf, d.buf)
```

Struct →

```
type Digest struct {
    h       [8]uint32
    t0, t1 uint32
    f0, f1 uint32

    buf []byte

    // size is defint
    // return. Since
    size int
}
```

Zeroing the buffer



Pattern matching (link)

```
// Zero the unused portion of the buffer. This
triggers a specific optimization for memset, see
https://codereview.appspot.com/137880043

padBuf := d.buf[len(d.buf):cap(d.buf)]
for i := range padBuf {
   padBuf[i] = 0
}
dCopy.buf = d.buf[0:cap(d.buf)]
```

Results: $ benchstat use_fixed_array use_memset

```
name           old time/op    new time/op    delta
Hash8Bytes-4     627ns ± 0%     596ns ± 0%  -4.94%  (p=0.000)
Hash1K-4        4.28µs ± 0%    4.24µs ± 0%  -0.90%  (p=0.000)
Hash8K-4        31.4µs ± 0%    31.4µs ± 0%  -0.13%  (p=0.000)

name           old speed      new speed      delta
Hash8Bytes-4   12.8MB/s ± 0%  13.4MB/s ± 0%  +5.24%  (p=0.000)
Hash1K-4        239MB/s ± 0%   241MB/s ± 0%  +0.92%  (p=0.000)
Hash8K-4        261MB/s ± 0%   261MB/s ± 0%  +0.13%  (p=0.000)
```

Allocations and copies

```
$ ag "make\(" blake2s.go
55: buf := make([]byte, 32)
98: buf:  make([]byte, 0, BlockSize),
325:dCopy.buf = make([]byte, cap(d.buf)) // want zero-padded to BlockSize
343:out := make([]byte, dCopy.size)
390:params.Salt = make([]byte, SaltLength)
399:params.Personalization = make([]byte, SeparatorLength)
414:keyBuf := make([]byte, BlockSize)
```

Reuse the slice

It’s just slice reuse in an append API (like Sum)

```
// Sum appends the current hash to b and returns
// It does not change the underlying hash state.
func (d *Digest) Sum(b []byte) (out []byte) {
    // If there's capacity, reuse the b slice
    if n := len(b) + d.size; cap(b) >= n {
        out = b[:n]
    } else {
        out = make([]byte, n)
        copy(out, b)
    }

    err := d.finalize(out[len(b):])
    if err != nil {
        return out[:len(b)]
    }
    return out
}
```

sliceForAppend (link)

```
// sliceForAppend takes a slice and a requested number of bytes. It returns a
// slice with the contents of the given slice followed by that many bytes and a
// second slice that aliases into it and contains only the extra bytes. If the
// original slice has sufficient capacity then no allocation is performed.
func sliceForAppend(in []byte, n int) (head, tail []byte) {
    if total := len(in) + n; cap(in) >= total {
        head = in[:total]
    } else {
        head = make([]byte, total)
        copy(head, in)
    }
    tail = head[len(in):]
    return
}
```

Results: $ benchstat use_memset reuse_slices

```
name           old time/op    new time/op    delta
Hash8Bytes-4     310ns ± 1%     284ns ± 0%  -8.56%  (p=0.001)
Hash1K-4        2.73µs ± 2%    2.69µs ± 2%  -1.60%  (p=0.035)
Hash8K-4        20.7µs ± 3%    20.5µs ± 2%    ~     (p=0.234)

name           old speed      new speed      delta
Hash8Bytes-4  25.7MB/s ± 1%  28.1MB/s ± 0%  +9.36%  (p=0.001)
Hash1K-4       375MB/s ± 2%   381MB/s ± 2%  +1.63%  (p=0.038)
Hash8K-4       395MB/s ± 3%   399MB/s ± 2%    ~     (p=0.234)
```

Results: $ benchstat use_memset reuse_slices

```
name           old time/op    new time/op    delta
Hash8Bytes-4     310ns ± 1%     284ns ± 0%  -8.56%  (p=0.001)
Hash1K-4        2.73µs ± 2%    2.69µs ± 2%  -1.60%  (p=0.035)
Hash8K-4        20.7µs ± 3%    20.5µs ± 2%    ~     (p=0.234)

name           old speed      new speed      delta
Hash8Bytes-4   25.7MB/s ± 1%  28.1MB/s ± 0%  +9.36%  (p=0.001)
Hash1K-4        375MB/s ± 2%   381MB/s ± 2%  +1.63%  (p=0.038)
Hash8K-4        395MB/s ± 3%   399MB/s ± 2%    ~     (p=0.234)
```

Diminishing Returns

Diminishing Returns

These look like:

- Don’t allocate some trivial intermediate variables
- Unroll remaining fixed loops
- Copy small functions into this package to allow inlining them
- Hunt down the less significant bounds checks

Worth it? Not really.

Many hours of my life.

Library of techniques, not ~~always~~ best practices.

Extremely compiler version dependent.

Still not competitive with assembly.


Rust
EVANGELISM
STRIKE FORCE

Micro-Optimizing Go Code

•••

George Tankersley
@gtank__

Code: https://github.com/gtank/blake2s