How to hash a Merkle Tree

Potuz
Prysmatic Labs
$ cd gohash/tree
$ go test -bench=.  

goos: linux

goarch: amd64

pkg: github.com/prysmaticlabs/gohash/tree

cpu: Intel(R) Xeon(R) CPU @ 2.80GHz

<table>
<thead>
<tr>
<th>BenchmarkHash_1_minio-2</th>
<th>2462506</th>
<th>473.1 ns/op</th>
</tr>
</thead>
<tbody>
<tr>
<td>BenchmarkHash_1-2</td>
<td>3040208</td>
<td>391.3 ns/op</td>
</tr>
<tr>
<td>BenchmarkHash_4_minio-2</td>
<td>577078</td>
<td>1959 ns/op</td>
</tr>
<tr>
<td>BenchmarkHash_4-2</td>
<td>1954473</td>
<td>604.9 ns/op</td>
</tr>
<tr>
<td>BenchmarkHash_8_minio-2</td>
<td>298208</td>
<td>3896 ns/op</td>
</tr>
<tr>
<td>BenchmarkHash_8-2</td>
<td>1882191</td>
<td>624.8 ns/op</td>
</tr>
<tr>
<td>BenchmarkHash_16_minio-2</td>
<td>147230</td>
<td>7933 ns/op</td>
</tr>
<tr>
<td>BenchmarkHash_16-2</td>
<td>557485</td>
<td>1988 ns/op</td>
</tr>
<tr>
<td>BenchmarkHashLargeList_minio-2</td>
<td>10</td>
<td>105404666 ns/op</td>
</tr>
<tr>
<td>BenchmarkHashList-2</td>
<td>45</td>
<td>25368532 ns/op</td>
</tr>
</tbody>
</table>

PASS

ok  github.com/prysmaticlabs/gohash/tree  13.969s
SHA 256 Basics

- Break into 64 bytes chunks
- Schedule 64 dwords (4 bytes)
  - \( W_0, \ldots, W_{15} \) are the message
  - \( W_i, \ldots, W_{i+15} \) are computed in terms of \( W_{i-16}, \ldots, W_{i-1} \).
- Start with an incoming digest of 8 dwords \((a_0, \ldots, h_0)\)
- **Round** \(i\) takes 10 dwords \((a_1, \ldots, h_1; W_i, K_i)\) and returns \((a_{i+1}, \ldots, h_{i+1})\).
- incoming digest for next chunk: \((a_0, \ldots, h_0) + (a_{63}, \ldots, h_{63})\)
Message scheduling

\[ \sigma_0(W) = \text{ROR}_7(W) \land \text{ROR}_{18}(W) \land \text{SHR}_3(W) \]

\[ \sigma_1(W) = \text{ROR}_{17}(W) \land \text{ROR}_{19}(W) \land \text{SHR}_{10}(W) \]

- Compute 4 words at a time
- Can be done in parallel to rounds
- Does not depend on previously processed chunks.
Rounds

- **Incoming:**
  - Status 8 dwords: \((a_n, b_n, \ldots, g_n, h_n)\)
  - Constant \(K_n\)
  - Scheduled word \(W_n\)

- **Outcoming:**
  - Status 8 dwords: \((h_{n+1}, a_{n+1}, \ldots, f_{n+1}, g_{n+1})\)
  - Depends on previous steps.
  - Depends on Scheduled words
The last block contains the length of the message as a little endian \texttt{uint64}. This length occupies the last 64 bits of the last 512 bits (64 bytes block). A bit 1 is added after the last bit of the message, to signal its end.
Vectorization

- Word scheduling can be done in parallel
- AVX can schedule 4 dwords at a time
- AVX2 can schedule 8 dwords at a time
- AVX-512 can schedule 16 dwords at a time
- AVX-1024 ...
- SIMD instructions can be interleaved with arithmetic ones for better pipelining
- **Rounds have to be scalar**
func hash(message []byte) [32]byte

def hash(data: bytes) -> Bytes32

pub fn hash(input: &[u8]) -> [u8; HASH_LEN]
Merkle Trees

Section 2
Parallelization + Fixed Size blocks

- Each node is a **256bit** hash
- Each node is the digest of hashing the concatenation of its two children (**512bits**)
- Siblings can be computed in parallel
Implementations

def merkle_root(self) -> Root:
    if self._root is not None:
        return self._root
    self._root = merkle_hash(self.left.merkle_root(), self.right.merkle_root())
    return self._root

def merkle_tree(leaves: Sequence[Bytes32]) -> Sequence[Bytes32]:
    bottom_length = get_power_of_two_ceil(len(leaves))
    o = [Bytes32()] * bottom_length + list(leaves) + [Bytes32()] * (bottom_length - len(leaves))
    for i in range(bottom_length - 1, 0, -1):
        o[i] = hash(o[i * 2] + o[i * 2 + 1])
    return o
func NewUsing(data [][]byte, hash HashType, salt bool) (*MerkleTree, error) {
    ...
    for i := len(data) + branchesLen; i < len(nodes); i++ {
        nodes[i] = make([]byte, hash.HashLength())
    }
    // Branches
    for i := branchesLen - 1; i > 0; i-- {
        nodes[i] = hash.Hash(nodes[i*2], nodes[i*2+1])
    }

    tree := &MerkleTree{
        salt:  salt,
        hash:  hash,
        nodes: nodes,
        data:  data,
    }

    return tree, nil
}
The right way to hash a Merkle Tree
The padding block is known, so we can hardcode the scheduled words $W_0, \ldots, W_{63}$.

~20%-30% gain.
Vectorization

- AVX can hash 4 blocks at a time (128bit)
- AVX2 can hash 8 blocks at a time (256bit)
- AVX-512 can hash 16 blocks at a time
- AVX-1024...

- ARM NEON is faster than scalar hashing
- AVX-512 is faster than crypto extensions
Hasher signature

func hash(message [][]byte) [][][32]byte

def hash(data:bytes) -> Sequence[Bytes32]

pub fn hash(input: &[u8]) -> Vec<[u8; HASH_LEN]>

void hash(unsigned char* out,
             const unsigned char* in,
             uint64_t count)
Thank you!

Potuz
Prysmaticlabs
potuz@prysmaticlabs.com

https://github.com/prysmaticlabs/hashtree
https://github.com/prysmaticlabs/gohashtree