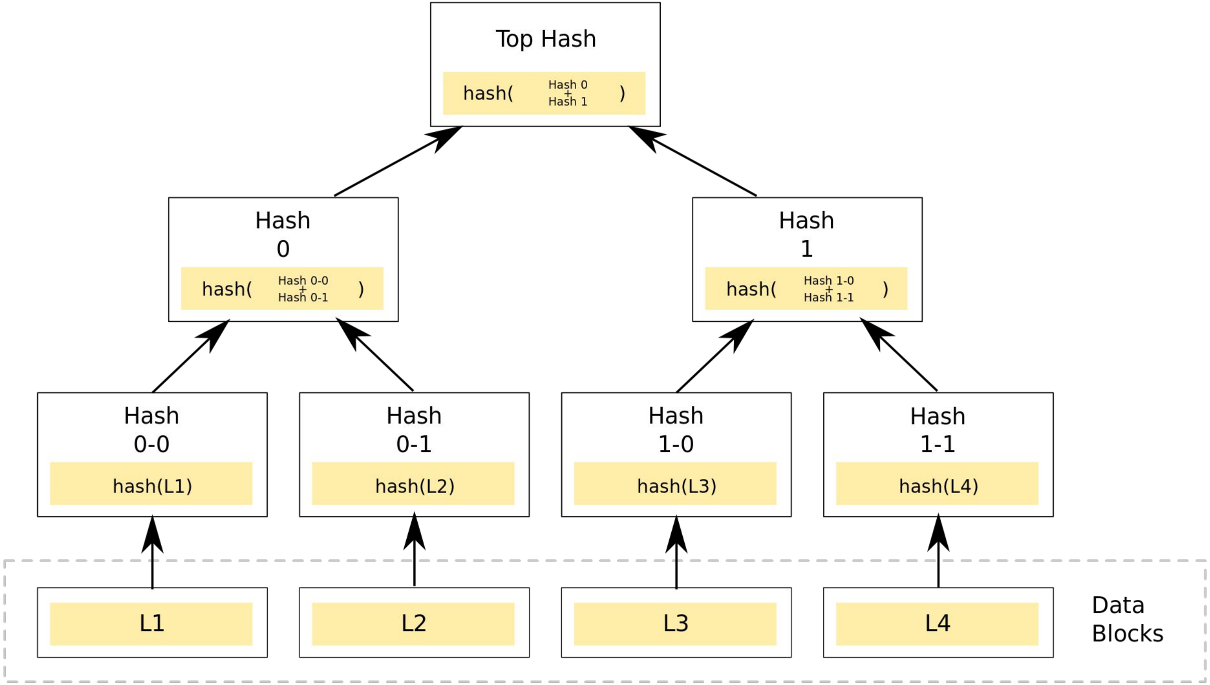
Concurrent Merkle Trees

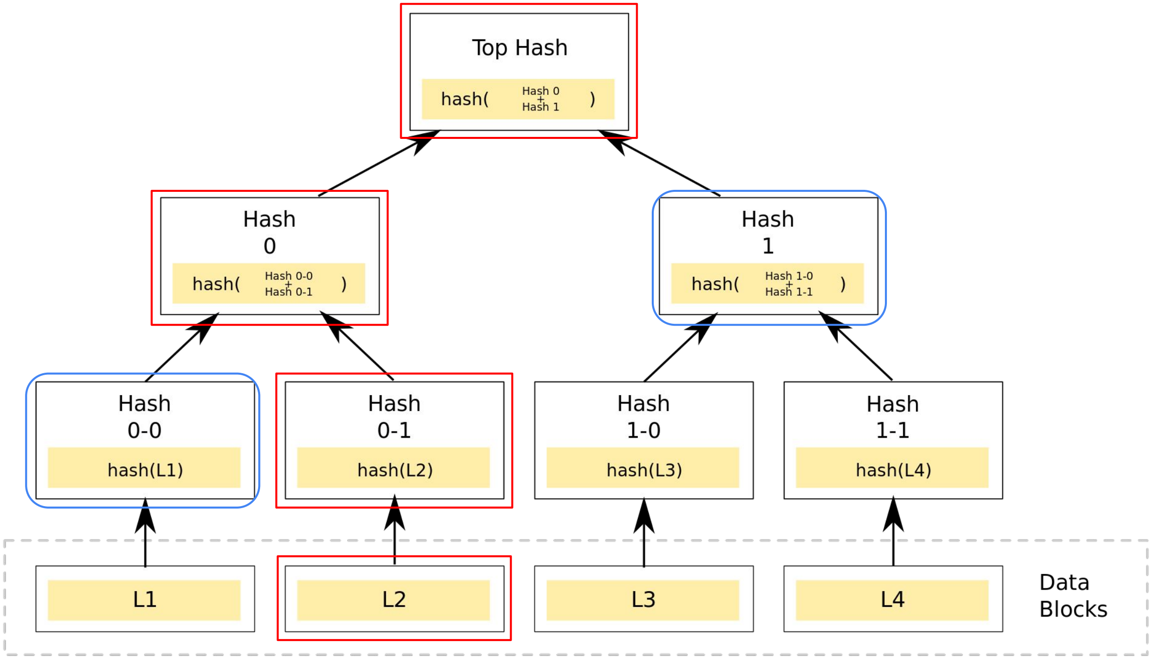
Gautam Singh (CS21BTECH11018)
K Vivek Kumar (CS21BTECH11026)

Merkle Trees



Merkle Trees: Updating a Leaf

- To update node L2, we have to update all nodes in red (Hash 0-1, Hash 0, Top Hash)
- Merkle proof is in blue (Hash 0-0, Hash 1).
  - Sometimes called “co-path”.
  - Used to verify the existence of a leaf by rebuilding the merkle root and checking equality.
- Both update and verify are of logarithmic complexity.



Concurrency conditions required for a continuously updating Merkle Tree

1. Concurrency in appending a data leaf

   Potential Issues: Order of receiving data leaves

2. Concurrency in correcting a data leaf and involved hash nodes

   Potential Issues: Delay in between identifying the false data leaf and updating the involved hash nodes

Concurrent Merkle Trees: Naive Implementation (C/C++)

```
    for (unsigned int l = 0, n = 0; l < blocks.size(); l = l + 2, n++) {
        if (l != blocks.size() - 1) { // checks for adjacent block
            nodes[n] = new Node(md5(blocks[l]->hash_val + blocks[l+1]->hash_val));
            nodes[n]->mom = blocks[l]; // assign children
            nodes[n]->dad = blocks[l+1];
        } else {
            nodes[n] = blocks[l];
        }
    }
    //std::cout << "\n";
    blocks = nodes;
    nodes.clear();

  }

  this->root = blocks[0];
}
```

Fig. 3: The serial implementation used in defining the Merkle tree.

```
    #pragma omp parallel
    {
        #pragma omp for ordered
        for (unsigned int l = 0, n = 0; l < blocks.size(); l = l + 2, n++) {
            if (l != blocks.size() - 1) { // checks for adjacent block
                nodes[n] = new Node(md5(blocks[l]->hash_val + blocks[l+1]->hash_val));
                nodes[n]->mom = blocks[l]; // assign children
                nodes[n]->dad = blocks[l+1];
            } else {
                nodes[n] = blocks[l];
            }
        }

    }
    //std::cout << "\n";
    blocks = nodes;
    nodes.clear();

  }

  this->root = blocks[0];
}
```

Fig. 4: The parallel implementation used in defining the Merkle tree.

Source: Parallelized C Implementation of a Merkle Tree | SpringerLink

Concurrent Merkle Trees: Using Changelogs (Rust/TS)

- How to validate the proofs?
  - Use changelogs!
- After previous updates, keep a record of the old values in the Merkle tree.
- For new update, search for the (old) proof in the changelogs and build from there.

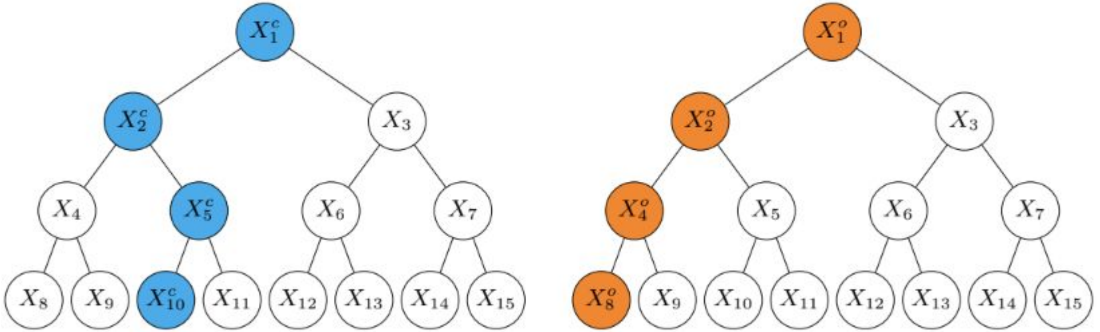


Figure 3: The cyan path and the orange path modify the same tree, but once one of the changes is locked in, the proof to the other change will be invalid.

Source: [Solana Concurrent Merkle Tree Whitepaper]

Concurrent Merkle Trees: Using Changelogs (Rust/TS)

Hyperparameter of changelog buffer size (K).

- Time complexity depends on additional factor.
- Storage overheads.
- Still not used in Solana ecosystem.

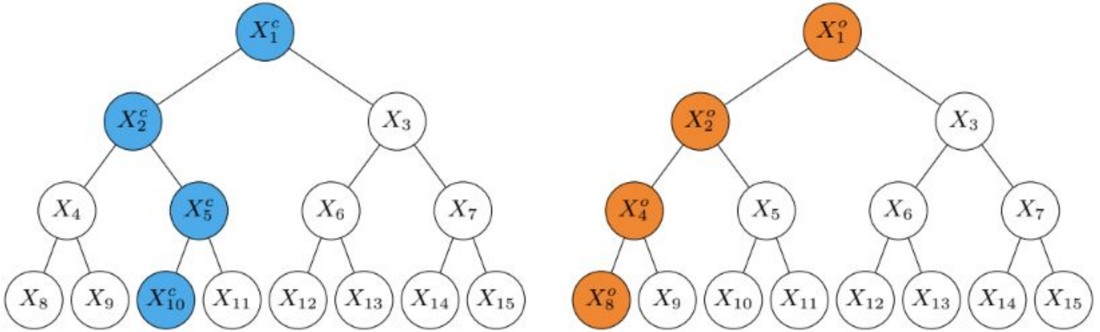
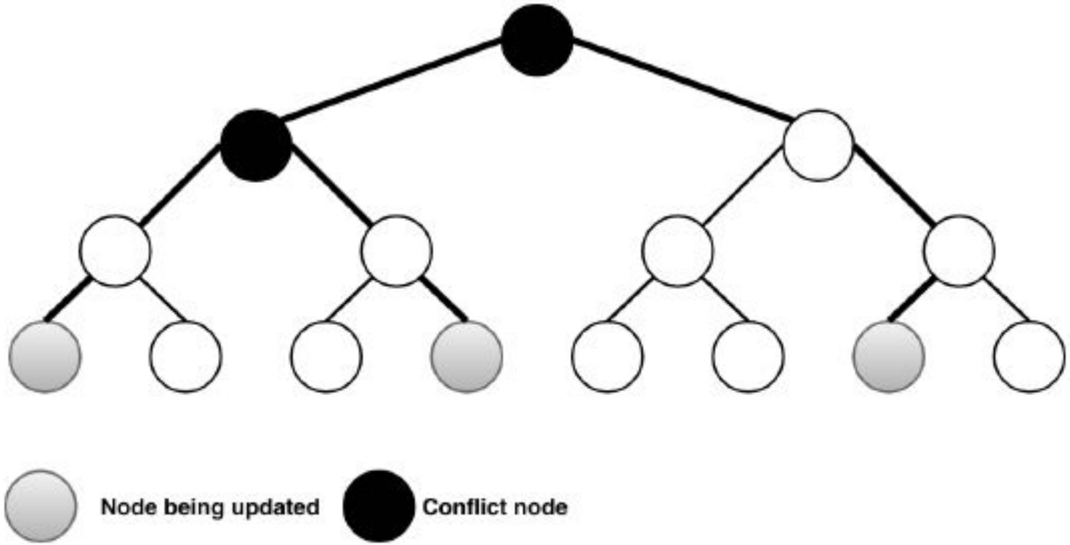
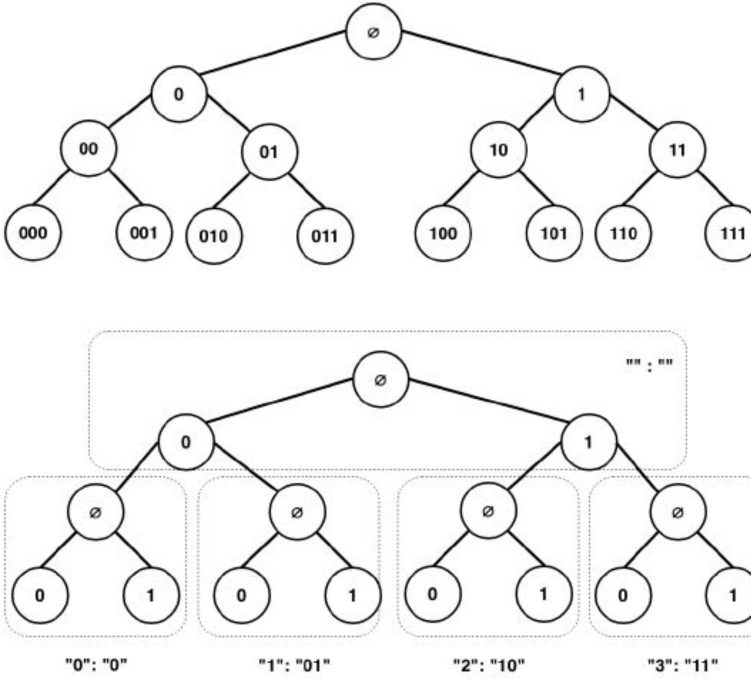


Figure 3: The cyan path and the orange path modify the same tree, but once one of the changes is locked in, the proof to the other change will be invalid.

Source: [Solana Concurrent Merkle Tree Whitepaper]

Concurrent Merkle Trees: Fine-Grained Locking (Go/Python)





Tree split among worker nodes (one subtree per AWS EC2 instance), race conditions are ONLY at conflict nodes.

Source: [Angela: A Sparse, Distributed, and Highly Concurrent Merkle Tree]

Concurrent Merkle Trees: Fine-Grained Locking (Go/Python)

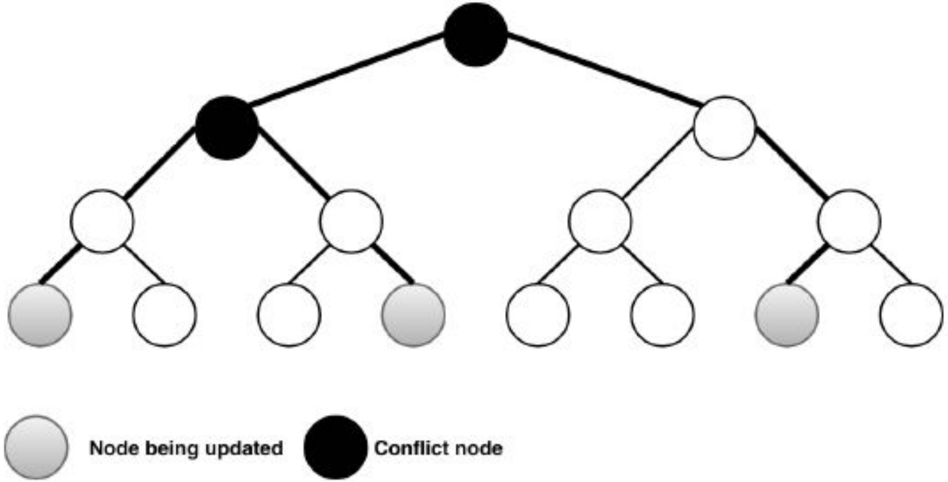
- Each update edge traversed only once.
  - Threads that arrive early at a conflict node return and are reused elsewhere.
- Lazy locking used here.
  - Threads lock only when they reach a conflict node.
- Overhead of maintaining conflict set.
  - We attempt to address this later!

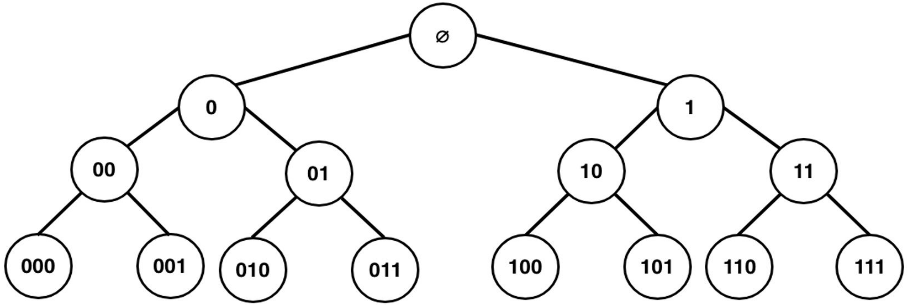
```
procedure update_Tree(Transaction T)
    while parent_exists() do
        if parent in Tree.conflicts then
            Tree.conflicts(parent).acquire_lock()
            defer Tree.conflicts(parent).release_lock()
            if parent has not been visited then
                Mark Node as visited
                End Thread
            else
                Process Update
            end if
        else
            Process Update
        end if
    end while
end procedure
```

Source: [Angela: A Sparse, Distributed, and Highly Concurrent Merkle Tree]

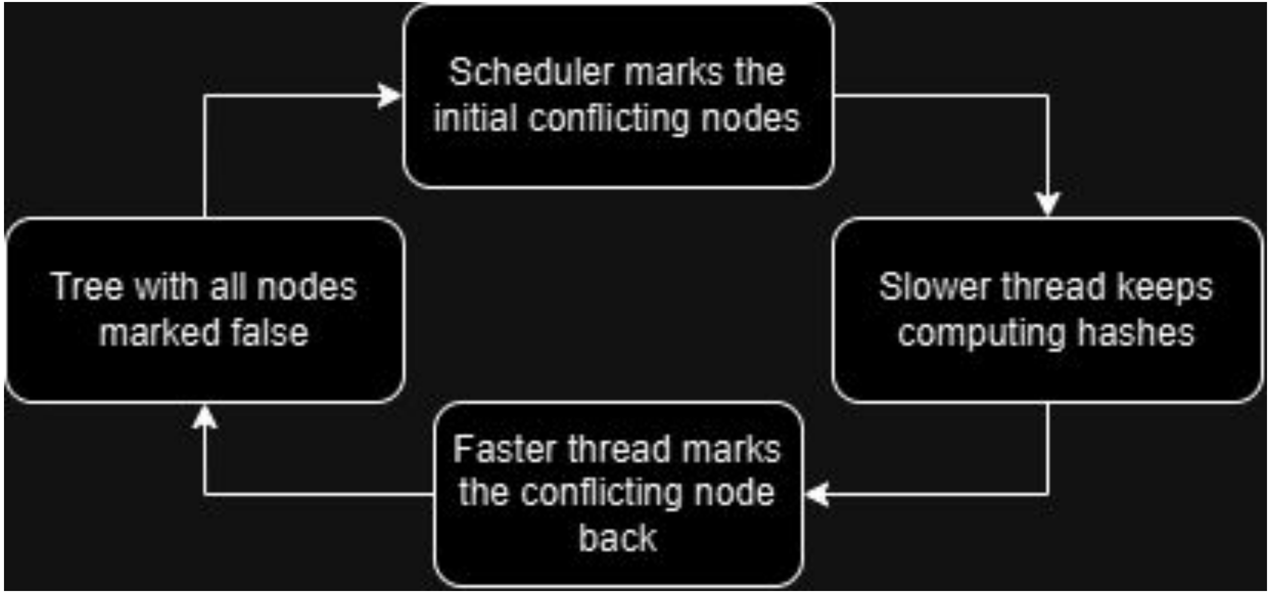
Our Proposition

- Setting: Scheduler decides operations in “read” and “write” phases.
  - Similar to Angela
  - Reads: Verifications
  - Writes: Updates
- Use barriers at conflict nodes.
  - Unlike traditional barriers, threads don’t wait but instead simply return.
  - Avoid lock acquire/release overheads as in Angela.
- For binary merkle trees, the barrier is effectively CAS.
  - Scheduler marks conflict nodes.
  - Threads atomically unmark them.

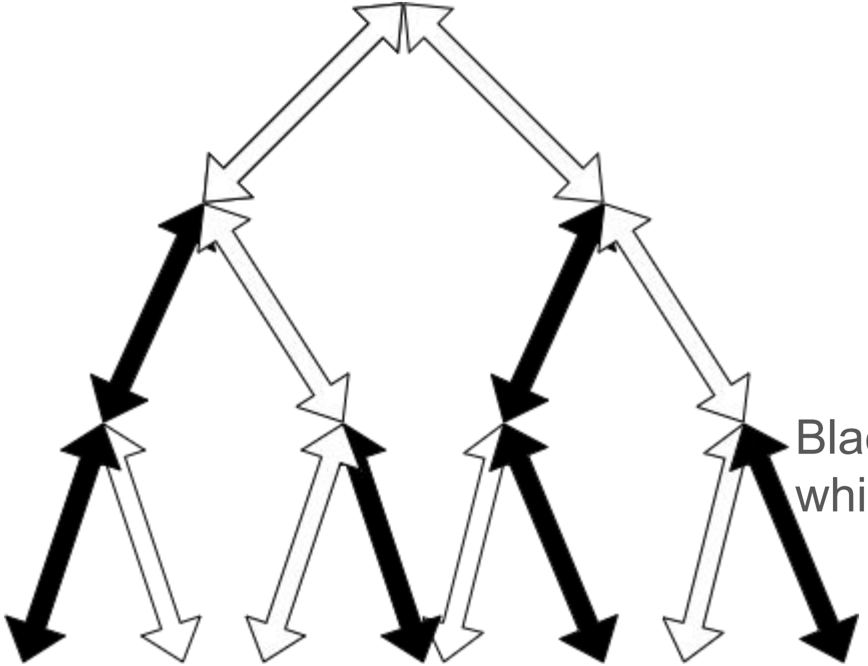




Balancing the nodes markings



Thread life visualization



White ends and black continues

Black ends and white continues

Algorithm

Algorithm 2 Algorithm for Single Node Update

```
procedure UPDATE(Update u, Tree T)
    node ← u.leaf
    while node ≠ null do
        node ← COMPUTEHASH(node)
        if node.parent = null then
            return                                  ▷ Root hash computed
        end if
        node ← node.parent
        marked ← CASMARK(node, true, false)
        if marked then
            return                                  ▷ Conflict detected
        end if
    end while
end procedure
```

Observations/Learnings

- Concurrency in Merkle Trees is not easy
  - Inherent dependencies
  - Need to process updates from bottom up
- Deficiencies of read-write scheduling
  - Overhead on scheduler
  - No concurrency between update and verify operations.
- What next?
  - Benchmark our proposition (implement one subtree per thread instead of one operation per thread).
  - Can we have a concurrent update and verify without significant overheads for lightweight blockchain clients (such changelog buffers in case of Solana)?
  - Better work splitting? One thread per subtree?