Concurrency in Merkle Trees

Gautam Singh
CS21BTECH11018

K Vivek Kumar
CS21BTECH11026

CONTENTS

***Abstract*—Merkle trees are efficient data structures for**

I. INTRODUCTION

Merkle trees are a powerful cryptographic data structure that have found widespread application across a variety of fields due to their efficiency in verifying and updating large sets of data. In scenarios where data needs to be verified over a network, the process can become time-consuming, especially as the volume of data increases. To address this challenge, Merkle trees enable fast and efficient verification by using cryptographic hashes of smaller subsets of data, rather than requiring the entire dataset to be transmitted or checked. This ability to streamline data verification makes Merkle trees indispensable in domains such as version control systems, distributed networks, and blockchain technologies, including cryptocurrencies. Their role in improving both the security and scalability of these systems has led to their adoption in numerous critical applications, from ensuring data integrity to supporting decentralized consensus mechanisms.

However, challenges arise when Merkle trees are used to store copies of data, such as transactions in a blockchain block. Transactions are typically received over a dense, high-frequency network, creating the need for a highly reliable and scalable system. To achieve this, a concurrent tree structure is essential, capable of efficiently handling updates and queries while maintaining data integrity. In the case of blockchain, Merkle roots are shared across nodes to identify the specific leaf nodes of data that require updates or validation. This process is integral to maintaining the blockchain's immutability and security.

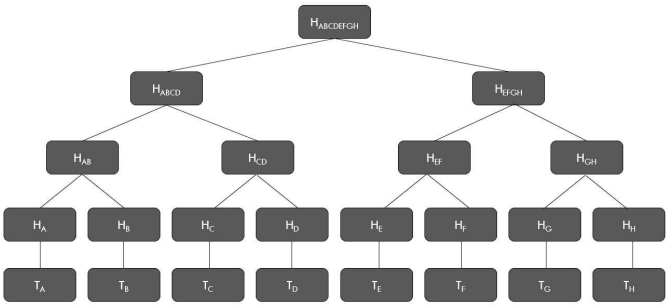

Fig. 1. Example of a merkle tree of depth 3. Here, A, B, etc. denote the data to be hashed.

For these Merkle trees to function efficiently on individual nodes, the system must rely on multiple threads to handle the read and write operations across various parts of the tree. Writer and reader threads are used to concurrently manage data updates and queries, ensuring the tree remains balanced and responsive. However, to maintain consistent and correct behavior, these threads must be programmed carefully to avoid race conditions and ensure linearity in updates and retrievals. Proper synchronization is critical to ensuring that the tree operates efficiently, even under high load, and that data integrity is preserved across concurrent operations.

II. PRELIMINARIES

A. Merkle Trees

A *merkle tree* is a hash tree where the leaves of the tree correspond to hashes of the data to be stored and verified and the inner nodes contain the hash of their children. Usually, merkle trees are complete binary trees. An example of a merkle tree along with the hashes computed is shown in Figure 1. The root of the tree is called the *merkle root*.

Merkle trees use cryptographic hash functions such as SHA-256 or SHA-384 that compress large amounts of data to a fixed-length digest. It is important to choose a cryptographic hash function so that the integrity of the data stored in the merkle tree is maintained. This is extremely useful in peer-to-peer networks like blockchains where transactions must be tamper-proof.

Merkle trees support the following fundamental operations:

1) *Insert/Update Leaf:* A merkle tree on N leaves supports insertion of a new leaf node or an update to an old leaf in $\mathcal{O}(\log_2 N)$ time. Additionally, for updating a leaf, we must also verify the existence of the leaf. The insertion and update methods both return the newly inserted node and a proof of existence of that node on success. The proof consists of nodes that were used in the hashing but were not on the path to the root from the leaf. This set of nodes is also known as the *co-path*. For example, the co-path of T_C is the list $[H_D, H_{AB}, H_{EFGH}]$.
2) *Verification:* To ensure tamper-proofing, merkle trees also support verifying the contents of a leaf. This is also done in $\mathcal{O}(\log_2 N)$ time and space. Similar to insertion, we reconstruct the merkle root given the leaf node contents and the co-path of the leaf in the proof. The leaf is verified to exist if the current merkle root is equal to the computed merkle root. Verification is used to verify the existence of an old node before it is updated.
3) *Get Merkle Root:* For verification, we require the current merkle root of the tree, which can be retrieved in constant time.

B. Concurrency in Merkle Trees

Merkle trees are fast data structures for verifying and updating data. They are widely used in file-storage systems, media distribution systems and in peer-to-peer blockchain networks. Hence, it is very important to parallelize these operations to achieve high performance at scale. The following complexities are to be addressed while trying to parallelize operations on merkle trees.

1) The contents of an internal node depend on the subtree rooted at that node. If any leaf nodes in its subtree changes, the hash of this node must change. This dependency makes it hard to parallelize the tree.
2) Concurrent updates will intersect at their lowest common ancestor in the tree, which we call a *conflict node*. Thus, during inserts and updates, threads must be able to detect whether a node is a conflict node or not and update the node accordingly.
3) During a proof verification, concurrent updates can invalidate all proofs since it changes the merkle root. Thus, we require a linearizable mechanism for validating proofs even though the merkle root may be changed, or to synchronize modifications to the tree to happen after all reads and verifications (quiescent consistency).

III. Related Work

For the breadth of applications of merkle trees, there is comparatively very little literature in the area of concurrent Merkle trees. Below, we discuss a few works that deal with this topic.

In [1], the authors implemented both serial and parallelized versions of a Merkle tree in C++ using OpenMP to achieve parallelism. However, the use of OpenMP directives in `for` loops such as `ordered` indicates a high level of synchronization, thus causing a massive bottleneck in performance.

In [2], the authors propose an approach to lock the ancestors on a path from a leaf to be updated to the root. This is a fine-grained locking approach that avoids deadlocks and is faster than traditional approaches of locking entire levels of trees while performing an update.

Scalable implementations of concurrent merkle trees are also seen in [3] and [4]. The authors of [4] have created the Solana Programming Library and implemented a concurrent merkle tree in Rust to be used in the Solana blockchain. This implementation makes use of changelog buffers to track recent updates. The key idea here is that concurrent validation methods will compute a merkle root that is present in the changelog buffer, thus the proof is considered valid and "fast-forwarded" to reflect the latest state of the merkle tree. This method has additional storage overheads and the failure rate of validations depends on a hyperparameter, namely the size of the buffer. On the other hand, the authors in [3] use Go and Python for implementation. The difference is that they use an orchestrator node to schedule insertion/updates and reads in batches. The orchestrator has the additional overhead of computing conflict nodes (i.e., nodes where two write operations first collide). These are the nodes that require to be serially executed, while all other updates may be executed in parallel. Unlike these methods which use locks, our method will make use of atomic read-modify-write (RMW) instructions to ensure consistency and correctness.

IV. Proposed Idea

In this section, we outline the proposed algorithm, prove its correctness and analyze it.

A. Batch Update Algorithm

We extend the algorithm presented in Angela to propose a batch update algorithm that does not use locks. Instead, we make use of the atomic compare-and-set (CAS) operation. The difference arises in the fact that we leverage a CAS operation to determine whether a thread updates the merkle tree *last* and not first.

Concretely, in our setup, update (or append) operations to the merkle tree are orchestrated into batches before applying them. As in Angela, the orchestrator will compute the indices where conflicts will happen on executing these batch updates. In our proposition, we only mark the nodes corresponding to these indices. During execution of updates, when a thread encounters a marked node, it attempts to atomically unmark the node using a CAS operation. If it succeeds, this means a conflicting thread will later update this node with the correct hash. Thus, the thread exits on successfully unmarking a node, allowing the other thread to continue. Eventually, the updates percolate to the root of the merkle tree correctly.

Our proposed algorithm is depicted in Algorithm 3, with subroutines in Algorithm 1 and Algorithm 2. The LCP function finds the lowest common prefix of two given binary numbers when read from most significant to least significant bit.

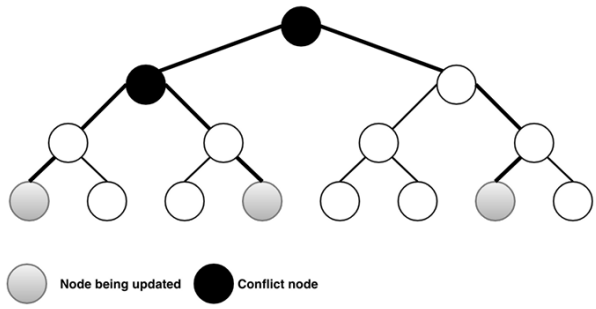

Fig. 2. Illustration of nodes to be updated and identification of conflicting nodes. [3]

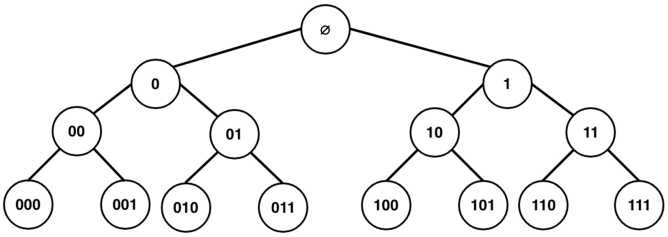

Fig. 3. Visualization of the Lowest Common Prefix (LCP) approach for identifying conflicting nodes within a batch of updates. [3]

Algorithm 1 Algorithm to Mark Conflicting Nodes

```
1: procedure MARKCONFLICTS(Updates U, Tree T)
2:     V ← sort(U)                          ▷ Sort updates by leaf indices
3:     for i ← 1 to V.length − 1 do
4:         j ← LCP(V[i].leaf.id, V[i + 1].leaf.id)
5:         j.marked ← true
6:     end for
7: end procedure
```

Algorithm 2 Algorithm for Single Node Update

```
1: procedure UPDATE(Update u, Tree T)
2:     node ← u.leaf
3:     while node ≠ null do
4:         node ← COMPUTEHASH(node)
5:         if node.parent = null then
6:             return                       ▷ Root hash computed
7:         end if
8:         node ← node.parent
9:         marked ← CASMARK(node, true, false)
10:        if marked then
11:            return                       ▷ Conflict detected
12:        end if
13:    end while
14: end procedure
```

Algorithm 3 Algorithm for Batch Updates

```
1: procedure BATCHUPDATE(Updates U, Tree T)
2:     MARKCONFLICS(U, T)
3:     for all u ∈ U in parallel do
4:         UPDATE(u, T)
5:     end for
6: end procedure
```

B. Correctness

To argue the correctness of the batch update algorithm, we rely on the following important observations. Some of these are based on the algorithm shown in Angela.

1) When the batch of leaf nodes to be updated are sorted by their id, the list of LCPs of adjacent nodes give the full set of conflict nodes. For example, in Figure 2, the leaf nodes in gray labeled ”000”, ”011” and ”101” need to be updated. The list of conflict nodes are ”0” and ””, using the numbering in Figure 3. Denoting the lowest common ancestor of nodes X and Y as $LCA(X, Y)$, we observe that $LCA(A, C)$ is an ancestor of $LCA(A, B)$, where $A < B < C$ are leaves with increasing ids. This is a direct corollary of the fact that the LCP of nodes A and B has to be at least the LCP of nodes A and C, since the leaves are ordered. Hence, it is appropriate to take LCPs of adjacent leaves in their sorted order.
2) In binary merkle trees, the id of a node in binary subtly encodes the path from root to the node. Thus, the LCP of two node ids will encode the path from the root to the deepest node that is a common ancestor to both nodes, which is the conflict node corresponding to these two nodes. This is demonstrated in Figure 3.
3) Each conflict node is visited by exactly two threads. Obviously, conflict nodes are visited by more than one node by definition. Suppose a conflict node N is visited by two update paths from leaves A and B. If it is also visited by another thread with update path from leaf C, then N is an ancestor of C. However, since N has only two children, it follows that the path from C to N must pass through one of these children. Since the paths from A and B also pass through the children nodes of N, it follows that the update path of C will conflict with the update paths of either thread A or thread B before reaching N itself.

The correctness is now evident. Any conflict node is accessed by exactly two update paths, and the atomic CAS instruction will ensure exactly one update path terminates and the other continues. Further, since the thread that arrives at the conflict node later continues, we are also assured that it will compute the correct hash for the conflict node. Applying this argument inductively on the path from leaf to root, we conclude that the root hash is also computed correctly.

C. Analysis

The BATCHUPDATE procedure does not use additional memory unlike Angela. Instead, it simply marks nodes of the

merkle tree which threads use as an indication to exit.

Our algorithm does not use locks and is clearly wait-free. However, a glaring pitfall is that faster threads will be the first ones to reach conflict nodes, leaving slower threads to do more work. This would impact performance negatively. An alternative to having one thread per update would be to split the entire merkle tree into various subtrees, assigning one subtree to a thread as in Angela. However, this would make the implementation quite complex, requiring communication between threads such as message passing or work queues.

Another challenge is related to the fault tolerance of merkle trees. If a thread crashes while performing an update, it would leave the merkle tree inconsistent. In our situation, we can provision the orchestrator to take a snapshot of the merkle tree before scheduling the updates. The inherent risk is that if one thread of the update batch crashes, the other threads would have to start again from the snapshot. This could possibly lead to a high failure rate of insert and update operations.

V. Results

We implemented a prototype of the proposed approach using the `thread` library in `C++` and conducted a performance comparison between three implementations: (1) the sequential baseline, (2) Angela et al.'s implementation, and (3) our atomic-update implementation. The evaluation focused on measuring the time taken for updates, considering the number of blocks updated in a batch, and comparing the results across the implementations. In the sequential baseline, updates were applied by locking the entire path from the leaf to the root using the `mutex` library, ensuring exclusive access during each update operation.

Details of all three implementations are available in the following repository: https://github.com/goats-9/cs5300/tree/main/programming/Project/src.

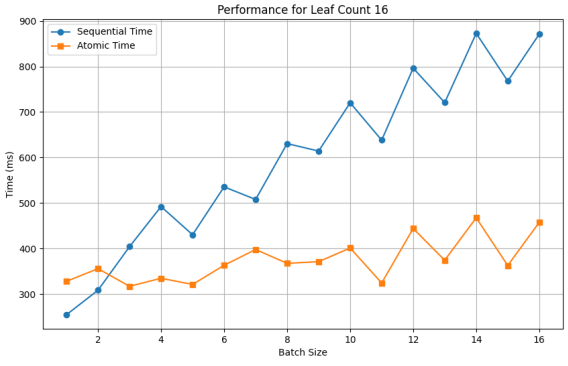

Fig. 4. Performance comparison of sequential and proposed implementations for a Merkle tree with a leaf count of 16, varying the batch size from 1 to 16.

Figure 4, Figure 5, and Figure 6 illustrate the comparative performance of the sequential and proposed implementations

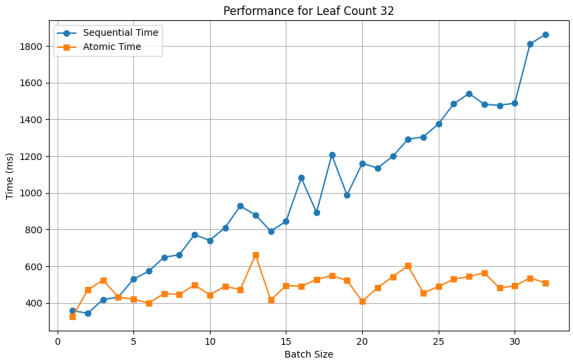

Fig. 5. Performance comparison of sequential and proposed implementations for a Merkle tree with a leaf count of 32, varying the batch size from 1 to 32.

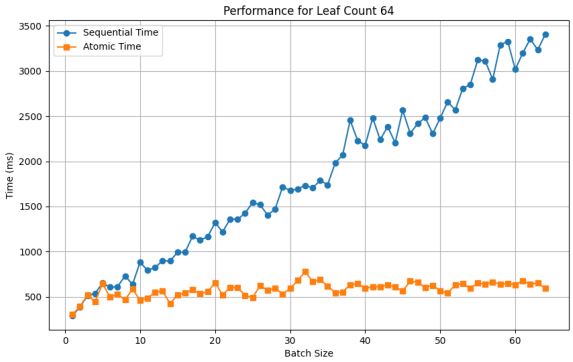

Fig. 6. Performance comparison of sequential and proposed implementations for a Merkle tree with a leaf count of 64, varying the batch size from 1 to 64.

for Merkle trees with leaf counts of 16, 32, and 64, respectively. The results demonstrate that our proposed atomic-update implementation significantly outperforms the sequential baseline. Specifically, the proposed approach achieves approximately six times lower latency across varying batch sizes. This improvement highlights the potential reliability and efficiency of our approach in blockchain applications, where reduced latency is critical.

Figure 7 and Figure 8 compare the execution times of the sequential baseline, Angela et al.'s implementation, and our proposed implementation for Merkle trees with leaf counts of 16 and 32. While the performance of Angela et al.'s implementation closely aligns with that of our proposed approach, this similarity is expected given that both build upon similar initial designs. However, our approach's advantages may become more pronounced in a distributed system context, as this prototype was evaluated on a single-node system.

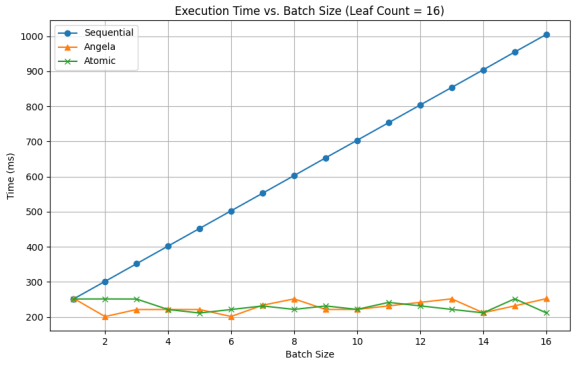

Fig. 7. Execution time comparison among sequential, Angela et al.'s, and proposed implementations for a Merkle tree with a leaf count of 16.

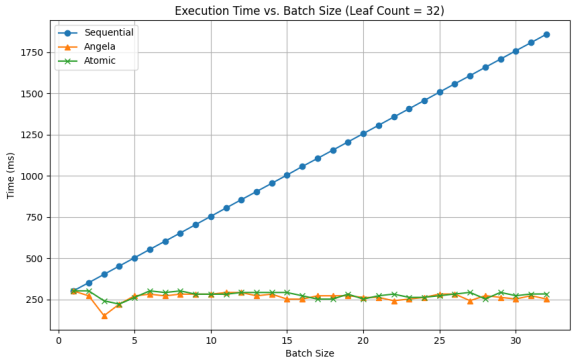

Fig. 8. Execution time comparison among sequential, Angela et al.'s, and proposed implementations for a Merkle tree with a leaf count of 32.

VI. Conclusion

In this report, we have reviewed the literature around concurrent merkle trees. After enumerating the shortcomings of various approaches, we have proposed a method that leverages atomic RMW instructions in batch updates and analyzed this method. The prototype of the proposal has been implemented using C++ and is compared with other implementations including Angela et al. and Sequential Locking. In the future, we would like to propose algorithms to make concurrent update and verify operations linearizable.

References

[1] A. Flangas, A. Cuellar, M. Reyes, and F. C. Harris, "Parallelized C Implementation of a Merkle Tree," in *ITNG 2021 18th International Conference on Information Technology-New Generations*, S. Latifi, Ed. Springer International Publishing, 2021, vol. 1346, pp. 107–114. [Online]. Available: https://link.springer.com/10.1007/978-3-030-70416-2_13

[2] M. El-Hindi, T. Ziegler, and C. Binnig, "Towards Merkle Trees for High-Performance Data Systems," in *Proceedings of the 1st Workshop on Verifiable Database Systems*, ser. VDBS '23. Association for Computing Machinery, pp. 28–33. [Online]. Available: https://dl.acm.org/doi/10.1145/3595647.3595651

[3] J. Kalidhindi, A. Kazorian, A. Khera, and C. Pari, "Angela: A Sparse, Distributed, and Highly Concurrent Merkle Tree." [Online]. Available: https://people.eecs.berkeley.edu/~kubitron/courses/cs262a-F18/projects/reports/project1_report_ver3.pdf

[4] Solana Concurrent Merkle Tree Whitepaper. Google Docs. [Online]. Available: https://drive.google.com/file/d/1BOpa5OFmara50fTvL0VIVYjtg-qzHCVc/view?usp=embed_facebook