Re-implementation of Angela

Diego Gomez
University of Central Florida
Orlando, Florida
dgomez17@knights.ucf.edu

Tiffany Lin
University of Central Florida
Orlando, Florida
tiffanylin@knights.ucf.edu

Wanda Mora
University of Central Florida
Orlando, Florida
wandamora@knights.ucf.edu

Samuel Tungol
University of Central Florida
Orlando, Florida
samryo1@knights.ucf.edu

Quick Summary of Merkle Trees

A Merkle tree consists of nodes that store hash values, where the leaf nodes are the individual hash values of each node. Every parent node is the concatenation of the hash values of its children, until there is one node left, which will become the root. All leaf nodes are at the same depth and are as far left as they can be.

A main property in this structure is the use of hash functions to efficiently search through large amounts of data. The key method used on this data structure is constructing the tree with the hash functions and building from the bottom row of leaf nodes and up to the root. Another key method is to validate the content of the data being stored using the hash values.

Sparse Merkle Trees:

In order to achieve the same results as Angela, we needed to make our Merkle tree into a sparse Merkle tree. The only difference between a regular Merkle tree and a sparse Merkle tree is that the sparse one's data is indexed, and each piece of data is placed at the leaf that is holding it. Leaf nodes that are not holding any data simply remain empty but are still included in the tree.

A main property of this kind of tree is that it can prove non-inclusion of a piece of data. For example, if we were to search for a data point that is meant to be at index 2, we would simply just check the leaf node at that index to see if it exists there or not.

Technical Details of the Implementation

Key Algorithms:

As mentioned in the summary of Merkle Trees, Merkle tree nodes focus on hashed values from the resulting leaf nodes. In our implementation, we decided to utilize the STL string and hash functions to process the hash values. A list of empty leaf nodes containing string fields are introduced for the creation/populate function of the tree, where they are converted to Merkle Node pointers. These leaf nodes will contain empty strings and the hash of an empty string, as per the Angela implementation and sparse representation.

Once the bottom level of Merkle Nodes is created, the Merkle Tree is built from the bottom-up since all the above hashes depend on the bottom layer. The second part of our creation/populate requires a pair of hashes from two Merkle Nodes to be concatenated and then hashed again for the parent's hash value. This is done recursively till there are no more bottom nodes to turn into the next level and only the root node is left in the function.

Outline in section Ⅲ, insert_leaf() function is built using a modified Binary Search for index to find the indexed Merkle node of the tree. While we traverse through the tree, we push the parent and sibling into a stack which will be used to modify the tree once the hash is created. Once the Merkle node at that index is reached and at the bottom level, we create a leaf node to replace the previous with the hashed data. Since the parent hashes depend on the bottom, we pop through the stack modifying all hash values with the new hash.

Similarly to insert_leaf(), generate_proof() focuses on the same modified Binary Search to find the indexed Merkle Node. Once the Merkle Node is found, we create a Proof object that contains the value and hash of the Leaf Node at that index. As well as all the siblings that were along the path of getting to the node. This will be used to create a proof that verify_proof() can determine if it exists in the tree by checking the root hash.

In Section Ⅴ.B, the approach that the Angela Merkle Tree uses for concurrency is explained as well as the form of conflict encoding. Each Merkle Node contains an encoding value which represents it's spot in the tree based on height and index. For finding conflicts in the Merkle Tree, the longest common prefix must be taken into account from all the transactions in the vector (Explained in Ⅴ.C.2).

These conflicts are utilized in the batch_update method where the tree conflict field must be modified to represent the current conflicts in the tree and then update the tree for each transaction. As the pseudo code shows, we determine if there are any conflicts on the parent node if not then we can update the tree. If the parent is in conflict, the thread must lock the parent and see if the parent has been visited before and run the insert if it has not.

As for thread creation and thread functionality, we run operations on the tree that each thread grabs a pre-generated random number to determine which method it will run.

Functions

In this section we will be going over the reimplementation of key functions in Angela.

- `bool insert_leaf(int index, string data);`
  The goal of insert_leaf() is to insert a new transaction into the Merkle tree and make updates to the tree accordingly. To make this happen, as insert_leaf() traverses down the tree to the leaf node at the designated index, it keeps track of the parent and sibling nodes along the way in a stack. With the stacks, insert_leaf() traverses back up the tree to modify the hashes of the parent nodes that were affected by the insertion.
- `MerkleNode* get_signed_root();`
  The goal of get_signed_root() is to return the Merkle root. To make this happen we atomically store the Merkle root into the Merkle tree after the tree has been properly populated.
- `Proof* generate_proof(int index);`
  The goal of generate_proof() is to return a Proof object that can be later used to verify whether an item actually exists or not in the Merkle tree. To make this happen generate_proof() keeps track of the siblings of the node at the designated index as it traverses down the tree, and returns the list of siblings in the proof object at the end.
- `bool verify_proof(Proof* proof, string data, MerkleNode* root);`

The goal of verify_proof() is to take the proof object returned by generate_proof() and process whether that data exists in the tree or not. To make this happen verify_proof() takes the hash of the data passed in and calculates the root hash of the tree with its sibling hashes to verify if that calculated hash is equivalent to the current root's hash.

To support concurrent updates for sparse Merkle trees in a multithreaded environment we followed the *"Finer Grain Conflict Locking"* methodology explained in the paper.

- `void batch_update(vector<Transaction> trans, int ind);`
  The goal of batch_update() is to insert a list of transactions into the tree. To make this happen, batch_update() needs to create a list of conflicts that the transactions will have when modifying the tree. After the conflicts have been determined, batch_update() is safe to call update on all the transactions.
  - `vector<string> find_conflicts(vector<Transaction> trans);`
    The goal of find_conflicts() is to assist batch_update() in finding all of the conflicts between the transactions. To make this happen find_conflicts generates all the encodings for the nodes in the tree based on Huffman encoding, then it makes comparisons between the transactions encodings to find common prefixes. The node's with common prefixes as their encoding will be a conflict node.
  - `void update(Transaction trans);`
    The goal of update() is to reduce locking contention between the nodes. To make this happen update() has to check whether the parent to that transaction is a conflict node. If the parent node is a conflict node then we must lock it to ensure that if it has not been visited, we should mark it as visited so that another thread can complete the update. Then the current thread should release the lock on the parent node and terminate itself to relieve any contention between itself and the other thread who is trying to make changes to the same sub-tree. However, if the parent node has been visited or is not a conflict node, then the current thread can insert the new transactions into the tree with ease.

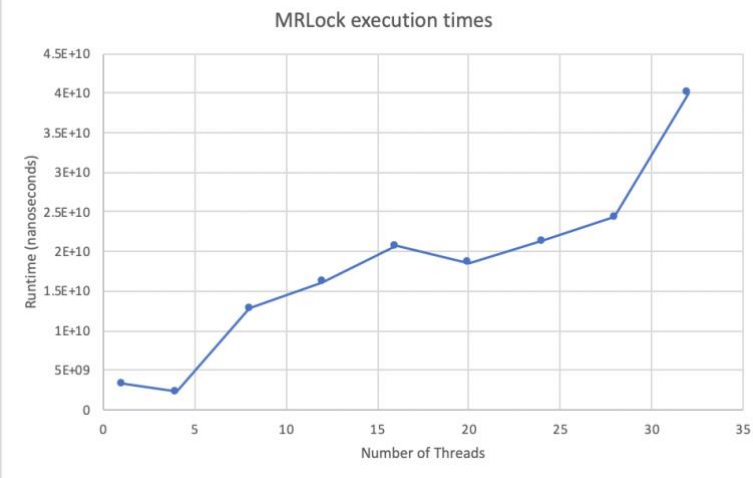
Performance Analysis

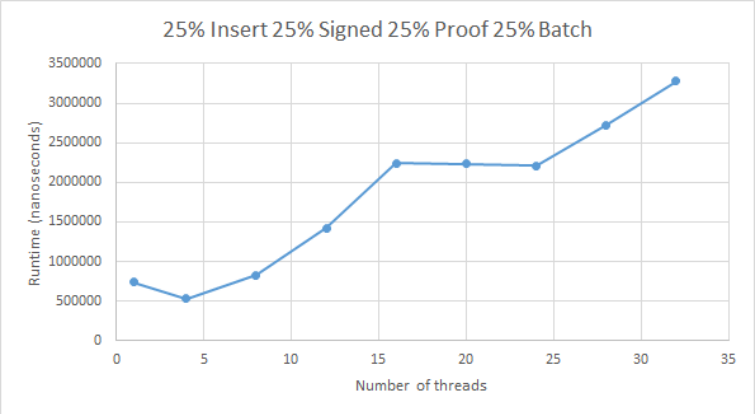
We measured performance by comparing how the number of threads affect the runtime. We only start the timer after pre-populating the Angela tree with some nodes before running the threads. We do the same for the MRLock concurrent tree.

Each graph has different ratios of the insert_leaf, get_signed_root, generate_proof with verify_proof, and batch_update. As seen with these following graphs, a general trend for all the graphs is that the runtime increases as the number of threads increases. Based on the graphs, having insert_leaf run at a greater ratio resulted in longer runtimes. This makes sense as batch_update and insert_leaf could have conflicting nodes that would result in extra runtime as threads have to wait for a thread to unlock a node.

When comparing the two graphs, we can see that the Merkle tree using MRLock is much slower than the Angela tree. With the way we use MRLock in our concurrent tree, we lock right before a thread calls a function. This can cause a lot of contention when multiple threads are requesting the same resource. If multiple threads are trying to access the same resource, MRLock uses a FIFO queue to handle which thread gets the resource. While a thread waits for

a resource, they will spin - this will greatly increase runtime, especially when multiple threads are waiting on the same resource.





References

MRLock:

- Zhang D., Lynch B., Dechev D. (2013) Fast and Scalable Queue-Based Resource Allocation Lock on Shared-Memory Multiprocessors. In: Baldoni R., Nisse N., van Steen M. (eds) Principles of Distributed Systems. OPODIS 2013. Lecture Notes in Computer Science, vol 8304. Springer, Cham

Main Topic Paper:

- Kalidhindi, J., Kazorian, A., Khera, A., & Pari, C. (2018). Angela : A Sparse , Distributed , and Highly Concurrent Merkle Tree.

Other:

- Chasinga, Joe. “Implementing a Bitcoin Merkle Tree.” *Medium*, Medium, 7 Mar. 2018, medium.com/@jochasinga/implementing-a-bitcoin-merkle-tree-cb0af3d53ec9.
- Ray, Shaan. “Merkle Trees.” *Hackernoon*, 14 Dec. 2017, hackernoon.com/merkle-trees-181cb4bc30b4.
- Lakshita. “Introduction to Merkle Tree.” *GeeksforGeeks*, 6 Dec. 2018, www.geeksforgeeks.org/introduction-to-merkle-tree/.
- Fichter, Kelvin. “What's a Sparse Merkle Tree?” *Medium*, Medium, 17 Oct. 2018, medium.com/@kelvinfichter/whats-a-sparse-merkle-tree-acda70aeb837.