Exploration of Concurrent Merkle Trees using MRLock

Diego Gomez	Tiffany Lin	Wanda Mora	Samuel Tungol
University of Central Florida	University of Central Florida	University of Central Florida	University of Central Florida
Orlando, Florida	Orlando, Florida	Orlando, Florida	Orlando, Florida
dgomez17@knights.ucf.edu	tiffanylin@knights.ucf.edu	wandamora@knights.ucf.edu	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.

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 generic leaf nodes is introduced for the creation/populate function of the tree, where they are converted to Merkle Node pointers with the Leaf Node's value changed through the hash function.

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.

Inside of our insert function for the Merkle Tree, we utilize a FIFO queue to simulate a recursive traverse function by going through each pair and appending if they don't have any children signaling that the nodes are the bottommost level except if the tree's amount of nodes are odd where it will not have a right sibling.

Functions

- `MerkleNode* populate(vector<LeafNode<T> > leaves)`: Taking in a vector of transactions that are hashed, the populate function creates a vector of Merkle nodes that will be built into the tree from bottom up, using a helper function called recursivePopulate().
    - `MerkleNode* recursivePopulate(vector<MerkleNode*> hashedNodes, hash<string> hash)`: Using individual hashed Merkle nodes, the recursivePopulate() function will

pair up neighboring nodes to create parent hashed nodes that will be pushed into a new vector which will represent the next level of nodes above, while attaching the children nodes to the parent. This process will be repeated until you have reached the top level where there is only one node left the Merkle root.

- `MerkleNode* insertLeaf(MerkleNode* root, string passedHash)`: Recursively traverses down the pre-populated merkle tree that was passed in and adds the new leaf node with its hashed value by calling recursivePopulate() on the new list of nodes created from the traversal.

- `bool validate(vector<LeafNode<T> > a1, vector<LeafNode<T> > a2)`: Taking in two lists of transactions (a1 and a2), validate() creates a merkle tree for both of them and returns whether the merkle root of a1 and a2 matches each other.

MRLock

MRLock is built upon an array-based lock-free FIFO queue. Each thread has allocated a cell to write its resource request. The FIFO property implies starvation-freedom as threads will be served according to their arrival. For every resource within the set, each bit in a request maps to one resource. A thread whose bit in a request is a 1 indicates that the thread will use that resource. With this approach of using shared resources, it will allow for the max number of threads without having conflicting requests to continue. Whenever there are conflicts, the threads who are waiting on a shared resource will spin until they can obtain that resource.

We use MRLock within the thread's run function. The thread encapsulates the shared resources it requires and MRLock will determine how those shared resources will be allocated. Once a thread acquires all of its requested resources, it will complete its action and release the resources, allowing another thread to take those resources.

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/.