Lock-Free Resizeable Concurrent Tries

Jennifer Belvin, Faye Strawn, and Thomas Boswell

Abstract— **This paper describes an implementation of a non-blocking concurrent hash trie based on single-word compare-and-swap instructions in a shared memory system. Insert, lookup, and remove operations can be run independently and modify different parts of the hash trie. Remove operations make sure unnecessary memory is freed and that the trie is kept compact. We show this implementation is linearizable and lock-free and present benchmarks that compare concurrent hash trie operations against operations of other concurrent data structures.**

I. Introduction

A. Concurrency

With multiple processors, data needs to be accessed concurrently. Concurrent access to data requires synchronization in order to be correct. One approach to synchronization is to use mutual exclusion locks. However, locks can worsen performance if a thread holding a lock gets delayed. All other threads competing for the lock are prevented from making progress until the lock is released. In other words, mutual exclusion locks are not fault tolerant (a failure may prevent progress indefinitely).A lock-free concurrent object guarantees that if several threads attempt to perform an operation on the object, then at least some thread will complete the operation after a finite number of steps. Lock-free data structures are immune to deadlocks, and unaffected by thread delays and failures.

B. Sequential Hash Array Mapped Tries

The basic structure of a trie is that of a node with several arcs leading to subtries. The arcs each represent "a member of an alphabet of possible alternatives" (Bagwell). The problem of optimizing tries lies in balancing fast traversal against the space occupied by empty arcs. Bagwell proposes a balance between speed and space by limiting nodes to 32 bits each, wasting only 1 bit per empty arc.

An Array Mapped Trie keeps an array of pointers representing arcs to subtries and a bitmap corresponding to those arcs, marking which ones are valid or empty with ones and zeros respectively. Finding a particular node requires computing its place in the bitmap which can be done with Count Population command equivalents commonly available or emulated through shift and add instructions.

When using a hash table, the chief difficulties are in sizing and resizing the root hash table as the number of keys increase so that performance is kept at an optimal level. A Hash Array Mapped Trie assumes an infinite-sized hash table. In order to insert a value into the Hash Array Mapped Trie, values to be entered are given a hash-code through the hashing function of the programmer's choice. Then, proceeding through the root node, a section of the hash code corresponding to the depth of the current node is used to step through the Array Mapped Trie structure until a leaf node is found. If the value at this leaf node is a match, an insert is unnecessary and lookups and removals find the target value. If not, additional subtries are constructed until both values do not collide and can reside in their own leaf nodes.

II. Progress Guarantee

As the entire data structure is lock-free, progress is guaranteed for at least one thread in at least one stage of execution. Node insertions are performed via CAS and are retried upon failure, which in itself implies that another transformation to the trie originating from another thread was successful. The addition of tomb nodes into the structure of a normal trie makes removal of nodes possibly slower than insertion, but given that CAS is also the method used there a failure in one thread implies success in a different thread.

III. Correctness Condition

There are 3 main criteria for correctness: safety- the Ctrie corresponds to some abstract set of keys and all operations change the corresponding abstract set of keys consistently, linearizable- any external observer can only observe the operation as if it took place simultaneously at some point between its invocation and completion, and lock-freedom- if some number of threads execute operations concurrently, then after a finite number of steps some operation must complete (lock-freedom is discussed in Progress Guarantee above).

We assume that the Ctrie has a branching factor of 2^W. Each node in the Ctrie is identified by its type, level in the Ctrie l, and the hashcode prefix p (the sequence of branch indices that need to be followed from the root in order to reach the node). For a C-node $cn_{l,p}$ and a key k with the hashcode $h = r_0 * r_1 * ... * r_n$, we denote $cn.sub(k)$ as the branch with the index r_l or $null$ is the branch does not exist. Modifications to the trie are carried out through CAS instructions, which are the points at which each type of modification can said to have taken place (or, in the case of failure, attempted). All data reads are atomic.

A. Safety

At all times t, a Ctrie is in a valid state S, consistent with some abstract set A. All Ctrie operations are consistent with the semantics of the abstract set A.

B. Linearizability

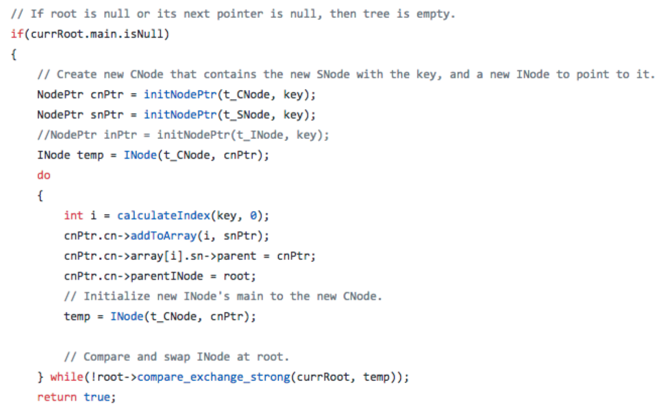
Ctrie operations are linearizable.

IV. Key Techniques

There are two deviations from traditional trie structure that allow this data structure to remain lock-free and concurrent. The first is the addition of indirection nodes, or I-nodes. I-nodes are found between every two nodes that are not I-nodes themselves. These nodes act as a sort of anchor for CAS instructions. Instead of modifying individual fields of nodes (necessary with normal trie structure unless the entire tree is replaced with each modification), a CAS is performed on the I-nodes single field (which may point to a newly constructed node or even several connected nodes if necessary). Conflicting CAS instructions may disallow all but one instruction at a time, but no overwrites are performed and no changes are lost.

The second deviation is the introduction of tomb nodes. A tomb node is any node that is not an I-node or a root node that only contains a single key, and no modifications can be performed directly on an I-node that points to a tomb node (which are referred to as tombed I-nodes). The introduction of a tomb node is to preserve the resizable property of the trie. If an I-node points to a tomb node, restructuring of the trie may be possible and should be attempted. Making tombed I-nodes immutable prevents threads from overwriting changes during restructuring, as any changes to a tombed I-node require that a thread move keys (using concurrent inserts) in such a way that the I-node now points to a node with multiple keys. Either way, no changes will be overwritten by competing threads.

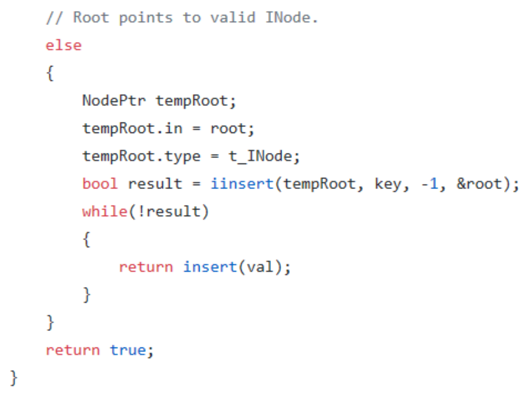
V. Algorithm Overview



```
// If root is null or its next pointer is null, then tree is empty.
if(currRoot.main.isNull)
{
    // Create new CNode that contains the new SNode with the key, and a new INode to point to it.
    NodePtr cnPtr = initNodePtr(t_CNode, key);
    NodePtr snPtr = initNodePtr(t_SNode, key);
    //NodePtr inPtr = initNodePtr(t_INode, key);
    INode temp = INode(t_CNode, cnPtr);
    do
    {
        int i = calculateIndex(key, 0);
        cnPtr.cn->addToArray(i, snPtr);
        cnPtr.cn->array[i].sn->parent = cnPtr;
        cnPtr.cn->parentINode = root;
        // Initialize new INode's main to the new CNode.
        temp = INode(t_CNode, cnPtr);

        // Compare and swap INode at root.
    } while(!root->compare_exchange_strong(currRoot, temp));
    return true;
```

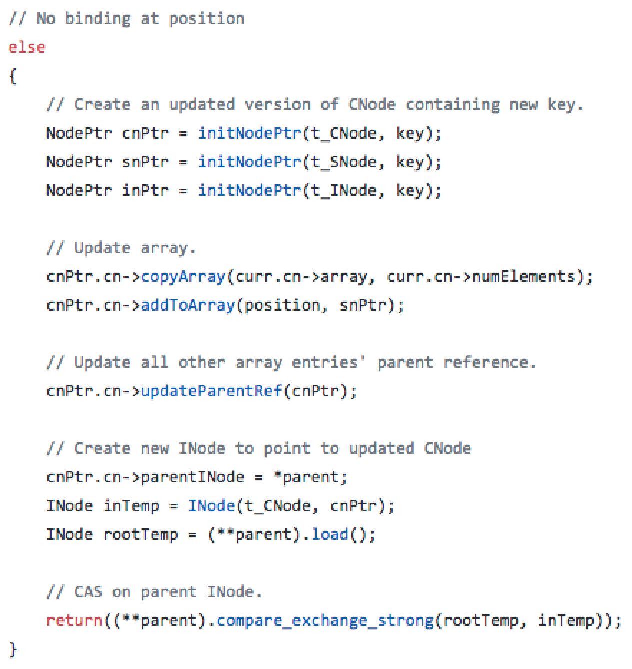
Fig. 1. insert(): Insert on Empty Tree

There are three main operations that may be performed on the Ctrie: insert, lookup, and removal. All operations start by reading the root. If the root is null and the trie is empty, neither removal nor lookup find a key. If the root points to an I-node set to null, the root is set back to null before repeating.



```
        // Root points to valid INode.
        else
        {
            NodePtr tempRoot;
            tempRoot.in = root;
            tempRoot.type = t_INode;
            bool result = iinsert(tempRoot, key, -1, &root);
            while(!result)
            {
                return insert(val);
            }
        }
        return true;
    }
```

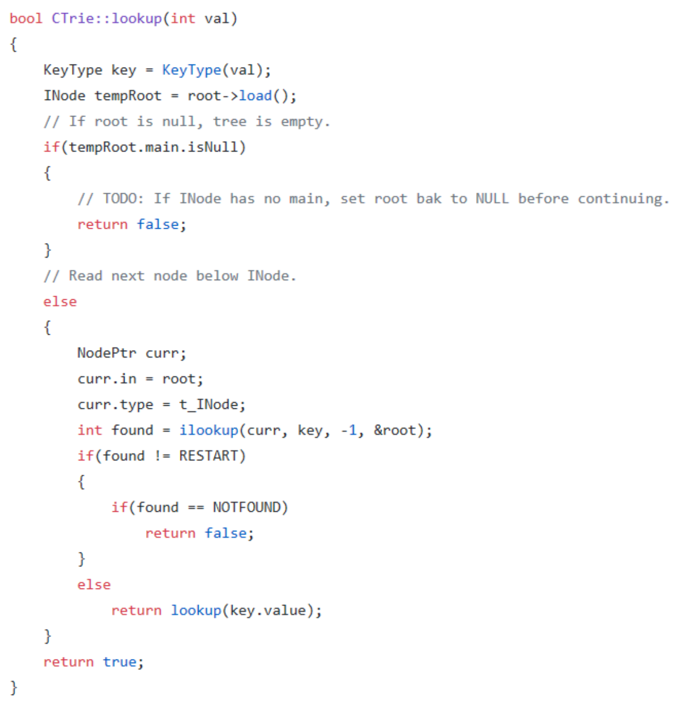
Fig. 2. insert(): Insert Operation on Non-empty Tree



```
// No binding at position
else
{
    // Create an updated version of CNode containing new key.
    NodePtr cnPtr = initNodePtr(t_CNode, key);
    NodePtr snPtr = initNodePtr(t_SNode, key);
    NodePtr inPtr = initNodePtr(t_INode, key);

    // Update array.
    cnPtr.cn->copyArray(curr.cn->array, curr.cn->numElements);
    cnPtr.cn->addToArray(position, snPtr);

    // Update all other array entries' parent reference.
    cnPtr.cn->updateParentRef(cnPtr);

    // Create new INode to point to updated CNode
    cnPtr.cn->parentINode = *parent;
    INode inTemp = INode(t_CNode, cnPtr);
    INode rootTemp = (**parent).load();

    // CAS on parent INode.
    return((**parent).compare_exchange_strong(rootTemp, inTemp));
}
```

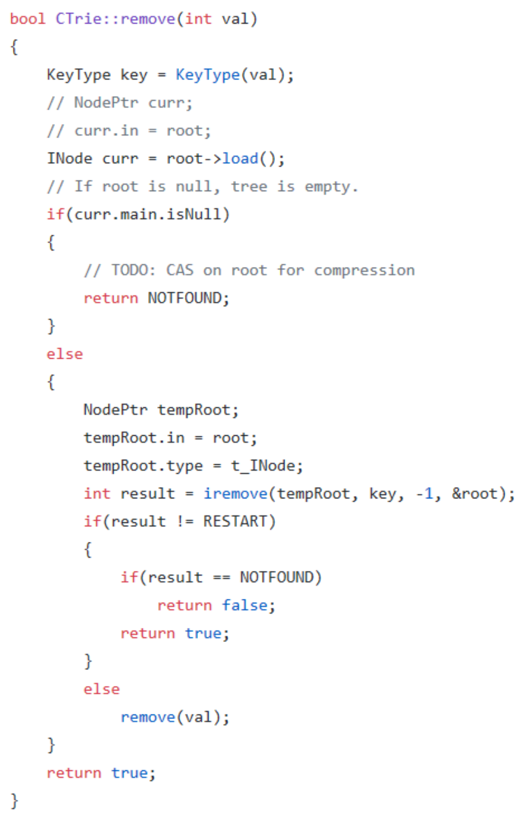
Fig. 3. iinsert(): Physical insert with no collision

If the root is null or the root points to an I-node set to null, insertion will replace the root reference with a new C-node with the appropriate key. If the root is neither null nor a null I-node, then the code reads the next node below the root I-node. If the node pointed to by the I-node is a C-node, the method flagpos computes the values flag and pos from the hashcode hc of the key, the bitmap bmp of the C-node and the current level lev. The relevant flag in the bitmap is defined as $(hc >> (k * lev))\&\&((1 << k) - 1)$, where 2^k is the length of the bitmap. The position pos in the array is given by the expression $\#((flag - 1)\&\&bmp)$, where # is the bitcount. The flag is used to check if the appropriate branch is in the C-node. If it is not, lookup and remove end, since the desired key is not in the Ctrie and insert creates an updated copy of the current C-node with the new key. If an I-node is found, we repeat the operation recursively for all operations. If a key-value binding(S-node) is found, a



```
bool CTrie::lookup(int val)
{
    KeyType key = KeyType(val);
    INode tempRoot = root->load();
    // If root is null, tree is empty.
    if(tempRoot.main.isNull)
    {
        // TODO: If INode has no main, set root bak to NULL before continuing.
        return false;
    }
    // Read next node below INode.
    else
    {
        NodePtr curr;
        curr.in = root;
        curr.type = t_INode;
        int found = ilookup(curr, key, -1, &root);
        if(found != RESTART)
        {
            if(found == NOTFOUND)
                return false;
        }
        else
            return lookup(key.value);
    }
    return true;
}
```

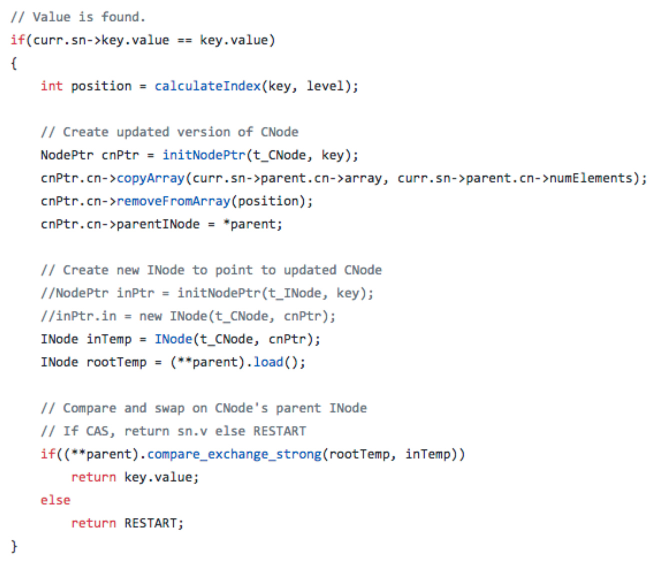
Fig. 4. lookup(): Lookup Operation



```
bool CTrie::remove(int val)
{
    KeyType key = KeyType(val);
    // NodePtr curr;
    // curr.in = root;
    INode curr = root->load();
    // If root is null, tree is empty.
    if(curr.main.isNull)
    {
        // TODO: CAS on root for compression
        return NOTFOUND;
    }
    else
    {
        NodePtr tempRoot;
        tempRoot.in = root;
        tempRoot.type = t_INode;
        int result = iremove(tempRoot, key, -1, &root);
        if(result != RESTART)
        {
            if(result == NOTFOUND)
                return false;
            return true;
        }
        else
            remove(val);
    }
    return true;
}
```

Fig. 5. remove(): Remove Operation

lookup compares the keys and returns the binding if they are the same. An insert operation will replace the old binding if the keys are the same, or extend the trie below the C-node if



```
// Value is found.
if(curr.sn->key.value == key.value)
{
    int position = calculateIndex(key, level);

    // Create updated version of CNode
    NodePtr cnPtr = initNodePtr(t_CNode, key);
    cnPtr.cn->copyArray(curr.sn->parent.cn->array, curr.sn->parent.cn->numElements);
    cnPtr.cn->removeFromArray(position);
    cnPtr.cn->parentINode = *parent;

    // Create new INode to point to updated CNode
    //NodePtr inPtr = initNodePtr(t_INode, key);
    //inPtr.in = new INode(t_CNode, cnPtr);
    INode inTemp = INode(t_CNode, cnPtr);
    INode rootTemp = (**parent).load();

    // Compare and swap on CNode's parent INode
    // If CAS, return sn.v else RESTART
    if((**parent).compare_exchange_strong(rootTemp, inTemp))
        return key.value;
    else
        return RESTART;
}
```

Fig. 6. iremove(): Physical Removal

they are not. A remove compares the keys and if they are the same, it replaces the C-node with its updated version without the key.

After a key is removed, the trie is contracted. Remove reads the node below the current I-node to check if it is still a C-node in order to create a tomb from the current C-node. Then it calls toWeakTombed that creates a weak tomb to check is it is still a C-node. If the number of nodes below the C-node that are not null I-nodes is greater than 1, then it is the C-node itself and there is nothing to entomb. If the number is 0, then the weak tomb is null. Otherwise, if the single branch below the C-node is a key-value binding or a tomb I-node, the weak tomb is the tomb node with that binding. If the single branch is another C-node, a weak tomb is a copy of the current C-node without the null I-nodes. The procedure tombCompress keeps trying to entomb the current C-node until it finds out there is nothing to entomb or it succeeds. If it succeeds it will return true, meaning a parent node should be contracted. The contraction is done in contractParent that checks if the I-node is still reachable from its parent. To check, it removes the null I-node or resurrects a tomb I-node into an S-node.

If any operation encounters a null or tomb node, it attempts to fix the Ctrie before proceeding. A tomb node may have originated from a remove operation that will attempt to contract the tomb node at some time in the future. Rather than waiting for that remove to do its work, the current operation contracts the tomb itself. It invokes the clean operation on the parent I-node, which will attempt to exchange the C-node below the I-node with its compression. If the C-node has a single tomb node directly beneath, then it is that tomb node. Otherwise, the compression is the copy of the C-node without the null I-nodes (the filtered call in the toCompressed procedure) and with all the tomb I-nodes resurrected to regular key nodes (this is what the map and resurrect calls do). For example, if some other thread were to attempt to write to I2, it would first do a clean operation on the parent I1 of I2 and it would contract the trie in the same way as the remove would have.

VI. Experiments

Some alternatives to Ctries are ConcurrentHashMaps and ConcurrentSkipListMaps. If we insert N elements, the insertion is divided among P threads, where P ranges from 1 to 32. Ctries outperform concurrent skip lists for P = 1 because a concurrent skip list corresponds to a balanced binary tree with a branching factor of 2, while Ctries normally have branching depth 32. The lower depth of Ctries means less indirections and fewer cache misses when searching the Ctrie. A Ctrie can also outperform a concurrent hash table at P = 1. This is because the hash table has a fixed size and is resized once the load factor is reached. In other words, a new table has to be allocated and all the elements from the previous hash table have to be copied into the new hash table. In order to do this, parts of the hash table must be locked so other threads adding elements into the table have to wait until the resize completes.

If we remove N elements, Ctries are outperformed by the other data structures. However, a concurrent hash table does not shrink once the number of keys becomes much lower than the table size, so the Ctrie is more space-efficient than the hash table. The Ctrie remove operation when P = 32 outperforms both the skip list and hash table. If we populate all the data structures with N elements and have a lookup operation for every element once, a concurrent skip list has more efficient lookups than other data structures. A Ctrie lookup outperforms a concurrent skip list when P = 32 but it still must traverse more indirections than a hash table.

Finally, if we do both lookups and insertions, the concurrent hash table performance equals that of concurrent tries. As the number of threads increases, opportunity for parallelism is lost during the resize phase in concurrent hash tables. If we preallocate the array for the concurrent hash table to avoid resize phases, it outperforms the concurrent trie. The downside of the hash table in this case is a large amount of memory must be used and the size needs to be known in advance.

VII. Implementation

A. Design Changes

Several changes were made to the original design laid out in the research paper. First, we had to eliminate the use of a bitmap to handle elements of the array. One advantage of using the bitmap is that it provides O(1) lookup or insertion into the member array of an arbitrary C-node. Another advantage is that it preserves the space efficiency of a hash array mapped trie, since the array is only as long as the numbers of bits that are set to 1 in the bitmap. The disadvantage of this is that it involves memory management because of the dynamic nature of the array - something that is very difficult to implement while preserving thread-safety. We thought about implementing a concurrent vector, but decided the overhead wasn't worth the space efficiency for the scope of this project. Instead, our implementation gives each C-node a static array of size 2^W, where W is typically 5. Though the array of size 32 was beneficial most specifically for use of the bitmap, it remains a reasonable arbitrary size for our purposes.

The next change we had to make was to implement generics for the node types. I-nodes point to at most one node, which can be either a C-node or an S-node. C-nodes contain arrays that can hold both S-nodes and I-nodes. While traversing the trie, a node of any type can be passed in to a recursive function. For these three reasons, it became necessary to create a generic type to wrap any arbitrary node. We first tried to implement this using C++ templates. Because templates are difficult to learn and debug, it became preferable to create a simpler, if more indirect, solution to this problem - a NodePtr object to hold pointers to each type of node.

The trouble with C++ templates also stretched to value types that the Ctrie can hold. Because of our trouble with generics, we pared down the structure so that it currently holds exclusively integers. Because of this, the function to calculate a hash code for a given key was modified to work better with integers.

B. Proposed Improvements

While implementing the structure, we hypothesized that thread-safety could be preserved if the only atomic objects are I-nodes. In almost every case, a node's parent I-node is the object that is modified during an operation. For each operation, when a node is to be modified through insertion, removal, or compression, a copy of the node is made with the desired modifications, its parent is declared to be the parent I-node of node being modified, then a compare and swap operation is performed on that parent I-node so that it now holds the address of the modified node. There is one edge case where it is an S-node that is modified- when the value trying to be inserted is already present in the trie.

To preserve correctness, the original algorithm performs a compare and swap on the S-node to place the new occurrence of the value into the same location. One solution to this edge case would be for every index of a C-node's array to hold an I-node, then the I-node will point to the S-node and therefore be the object that is modified atomically. The disadvantage of this method is that it forces a greater depth of the trie, which would affect performance over time. Therefore, our hypothesis was disproven. The solution was to make both I-nodes and S-nodes atomic objects, and leave only C-nodes as volatile.

C. Obstacles

We encountered many obstacles during this project, most of which were frustrating but ultimately widened our base of knowledge. The first obstacle was language. The source code supplied with the paper was written in Scala, a programming language that built on the paradigms of functional and object-oriented programming, and runs on the JVM. Though it claims to be built for conciseness, the logic of the source code written in Scala remains unattainable to the untrained eye. Our team was also unfamiliar with C++ syntax before working on this project, but with the help of VS Code's powerful debugger, this obstacle was mostly overcome.

A major source of the difficulties encountered translating this algorithm into C++ was the way that C++ passes values to functions. In order to successfully alter multiple portions of the trie concurrently, memory had to be altered atomically. Being unfamiliar with C++s syntax extended into being unpracticed combining C++s memory reference architecture (namely, pointers) with multiple threads enacting changes atomically. In addition, the polymorphic aspects of the trie presented difficulties in translation.

Not only is the trie algorithm meant to accommodate the hashing and placement of any data type, but the nodes described were able to reference multiple possible node types with a single reference, something that initially was a source of much difficulty in our implementation. Furthermore, there were obstacles with implementing functions such as removal or compression for various reasons including difficulty translating the source code and the complexity of translating the described methods into C++. Compression and cleaning in particular were given little comparative focus in the algorithm discussion but factored semi-frequently into the actual implementation.

Another obstacle we faced was the conflicting logic between the source code and the paper's design description. What we could decipher from the Scala code sometimes differed wildly from what was described in the original paper. In the end, we decided to ignore the source code and focus on the algorithm's design as described in the paper, and interpret it as we saw fit. The last obstacle is a blunt one- this algorithm is hard. The logic necessary to preserve linearizability in such a complex data structure is very convoluted, and took long hours of reading, attempting to recreate, re-reading, repeat. As we began to understand the design, we saw which bits were extremely clever and which bits were unnecessary, and had to redesign our entire implementation several times.

Despite all of the obstacles encountered, we implemented a functional concurrent insert function as well as remove and lookup functions, but the trie in its current state does not allow for resizing or compression.

VIII. Conclusion

We described a lock-free concurrent hash trie data structure implementation. Our implementation supports insertion, removal, and lookup operations. It is space-efficient because it keeps a minimal amount of information in the internal nodes, and it is compact because after all removal operations complete, all paths from the root to a leaf containing a key are as short as possible. Operations are worst-case logarithmic with a constant factor ($O(log_{32}n)$). Ctries grow dynamically because no locks are used and there is no resizing phase. We proved it is linearizable and lock-free.

References

[1] A. Prokovec, P. Bagwell and M. Odersky, *Cache-Aware Lock-Free Concurrent Hash Tries*. Lausanne, Switzerland: École Polytechnique Fédérale de Lausanne, 2011.

[2] A. Prokopec, "axel22/Ctries", *GitHub*, 2011. [Online]. Available: https://github.com/axel22/Ctries/blob/master/src/main/scala/ctries/ConcurrentTrie.scala.

[3] "Hash array mapped trie", *En.wikipedia.org*, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Hash_array_mapped_trie.

[4] P. Nash, "philsquared/hash_trie", *GitHub*, 2017. [Online]. Available: https://github.com/philsquared/hash_trie/blob/master/hash_trie.hpp.

[5] Marek, Marek's totally not insane idea of the day, *RSS*, 25-Jul-2012. [Online]. Available: https://idea.popcount.org/2012-07-25-introduction-to-hamt/.

[6] Bagwell, P. (2018). *Ideal Hash Trees*. [ebook] Lausanne, Switzerland. Available at: http://lampwww.epfl.ch/papers/idealhashtrees.pdf.