Final Paper

Rebeca Amaya, Melissa Gramajo, Emily Hok

Department of Computer Science, University of Central Florida, Orlando, Florida, 32816-2450

Abstract — This paper discusses a re-implementation of the cache-trie data structure in C++. This trie allows for constant time operations through a cache structure that reduces the space complexity of the original trie. This data structure is a lock-free parallel solution with major operations that include insert and lookup. Synchronization among threads in these methods is performed through freezing nodes.

Keywords — concurrent data structures, lock-free hash tries, constant-time hash tries, expected constant time

I. Introduction

Hash tries are data structures used to map hash values to keys. They are beneficial when trying to achieve an efficient lookup, insertion, and/or deletion to its trie. The time complexity for these operations is done in linear time, O(N), compared to other data structures, such as binary search trees or AVL trees, that take O(M log N), where m is the length of the string and n is the number of patterns to search for in the string. The problem with hash tries is that it has a high space complexity. The memory needs to create enough space for every node and pointer. The longer the string, the more nodes we store in memory and the bigger the alphabet, the more pointers the memory needs to hold. Concurrent lock-free hash tries improve the original data structure to run on multiple instructions calling the same operation at once, such as in Ctrie [3]. This data structure was the first non-blocking concurrent hash trie with O(log N) operations. One major change was using CAS in its operations to allow for concurrent operations. However, introducing a cache, also known as the cache-trie data structure, can further improve the space complexity and overall run-time of Ctries. This is because Ctries still need to allocate memory for every node inserted, including its pointer, which causes the trie space to grow and not scale down. Cache-tries address the space complexity issue by introducing a new idea of releasing pressure from the memory allocator and utilizing fewer allocations during its operations by creating a cache.

II. Overview

A cache-trie must have a root reference, which has access to an array of pointers called an array node. The array node is empty and is of length 16 at the start of initialization. As keys get inserted by their hash codes, the array gets populated with pointers to single nodes using CAS. If another key with the same hash code gets inserted, the cache-trie will extend to the next level and create another array of length 4, also called a narrow array, to reduce wasted space since only two keys are occupying the array. The new array will hold the previous key and the second key after a successful CAS at the first level. Now suppose another thread inserts a key that has a collision at the first and second level. Instead of extending the cache-trie to yet another level, the narrow array will replace itself with a larger array of size 16. This is a multistep process as a data race can occur. The first step is to freeze the narrow array to prevent further modifications that would otherwise be lost. Once the narrow array is frozen, pointers occupying the narrow array are swapped into the larger array. Lastly, this cache-trie features a cache to improve traversing the inner nodes. The cache is a singly-linked list of arrays containing pointers to nodes. Each array corresponds to one level of the trie and the cache sorts its arrays by starting with the deepest level.

```
class AnyNode


class SNode : public AnyNode
        virtual ~SNode(){}
        unsigned int hash
        int key
        string value
        Txn txn

class ANode : public AnyNode
        virtual ~ANode(){}
        AnyNode * wide[16]
        AnyNode * narrow[4]
        bool isWide

class ENode : public AnyNode
        virtual ~ENode(){}
        ANode parent
        int parentPos
        ANode narrow
        int hash
        int level
        ANode wide
```

Figure 1. Pseudocode of Cache-Trie Data Types

III. Related Works

The first trie data structure was introduced in 1959 by Briandais [1], although it had not been named yet. Its purpose was to aid in the localization of records in a slow memory medium through lookups with keys. This original data structure worked with strings and was designed for sequential operations. In 1960, Fredkin [2] further formalized and named the data structure. Some of the key advantages of the trie data structure are faster access time, flexible argument length handling, and the re-use of redundant data. Ctrie was the first proposed non-blocking concurrent hash trie [3]. Through atomic compare-and-swap instructions, Ctrie is able to perform insert, lookup, and remove operations independently, allowing for increased concurrency and dynamic trie growth. Synchronization among these operations is handled through freezing trie nodes, making them unmodifiable. These operations run in O(log N) time complexity. Ctrie was later expanded to allow O(1) snapshot operations which are used for linearizable iterators and size retrieval [5]. Patricia Tries store keys represented as strings where the path for a key is at most the length of the key [4]. Lookups are wait-free operations. This data structure also supports insert, delete, and replace operations in a linearizable and lock-free manner. The main feature of this trie is its replace operation, which allows for the modification of two different trie locations atomically. One key is deleted while another is inserted through a single word CAS instruction. However, the algorithms for the Patricia Tries use a large set of flags to avoid locking and require efficient memory management. PTries store binary strings and perform fast lookups and insertions like its other hash-trie counterparts [6]. It differs from other trie data structures because it compresses the data to reduce the amount of memory resources needed. The compression is performed through prefix sharing of the binary strings. The goal of the PTrie was to create an optimal balance between speed and memory utilization. While it accomplishes these goals, this implementation of PTries does not support parallelization yet. The CHAMP (Compressed Hash Array Mapped Prefix-tree) data structure works to minimize memory footprints through compression and improved cache locality [7]. However, it continues to be a sequential design with methods that run in

O(log N) time. Multiple threads can be supported though through Software Transactional Memory. Overall, it can be seen that while significant improvements have been made to trie data structures, they suffer from issues with memory management, lack a parallel implementation, and/or continue to be bounded by O(log n) time complexity. What makes the cache-trie novel is that it differentiates itself from other tries by introducing algorithms that can run in constant time, support lock-free concurrency, and demonstrate performance improvements. IV. Detailed Implementation *Make the pseudocode classes a picture and put at the end of the node class explanations. class AnyNode

*Need to add FSNode, FVNode, FNode, and CacheTrie Node Types The concurrent cach-trie implementation proposed by Prokopec was originally done in Scala. Moreover, part of the challenge was finding an equivalent representation of certain data types in C++. For example, we had to find an alternative to the Any class, which is a special class that sits on top of the Scala class hierarchy. To replicate the functionality of the Any type described in the paper, we created the class AnyNode whose sole purpose is to wrap a resource of any type. This will be described further in this section. Like the conventional trie, the cache-trie is composed of leaf nodes and inner nodes– represented by the SNode and ANode class respectively. We implemented the cache-trie to store words. Furthermore, SNode holds a string value. The SNode also holds an int type key and an unsigned int hash code. Together, the pair of values consistently and accurately tells the cache-trie where to insert and how to retrieve the leaf node. Lastly, it holds a txn field, which supports synchronization in the insert operation. The ANode class represents the inner node and is not associated with a key. Instead, each inner node contains an array of up to 16 pointers that can point to nodes of any type. This is accomplished by making the array of type AnyNode, which all node classes inherit. Additionally, each inner node refers to a specific level of the cache-trie, which grows in multiples of 4. To better illustrate this, the root ANode begins at level 0. The pointers held in its array point to nodes at level 4. The pattern repeats until we have a fully populated cache-trie. It is important to note that while the ANode class holds two arrays, only one of the arrays is being used during any quiescent period. In general, a newly created ANode will initialize and use the narrow array. Collisions will then prompt the cache-trie to expand the array, hence, replace the narrow array with the wide array. The root ANode is the only exception and will initialize and use the wide array upon the creation of the cache-trie. In order to allow multiple threads to access and modify contents concurrently, other node classes in addition to the SNode and ANode are necessary. These secondary node classes include the ENode, FNode, FSNode, and FVNode. They are especially important for the lookup function as they provide insight as to what state the cache-trie is in. For example, the ENode signals that an ANode needs to be expanded from its narrow array to its wide array. Finding an FNode indicates that an ANode is frozen and prevents further modifications to that inner node. The FSNode is like the FNode as it signifies a concurrent expansion has occurred, however, it also prevents changes to the SNode. Finally, the FVNode class is important for the freeze function. Specifically, the FVNode is used like a marker across all entries in a frozen ANode to prevent updates from other threads.

Operation Implementations

The first step in re-implementing the cache-trie was to create a sequential hash-trie with insertion and lookup operations. The insertion operation takes a key and a value to insert into the hash-trie. Before this operation is done, a check is made to verify that an insert is possible based on the following parameters: hash, value, current node, previous node, and its level. Depending on the current node's type, different actions are taken.

If the key already exists in the trie then the value is replaced with the new value. This is done by finding an ANode where we can insert or replace an SNode representing the new key-value pair. The hash of the key-value pair is calculated and then used to traverse the hash-trie and find the corresponding ANode and location where we will perform insertion. There are two different scenarios that can occur during this operation:

1. We find an open location in the corresponding ANode. Then we can just insert the SNode.

2. There is a collision in the location of the corresponding ANode. This occurs when we find the location to insert the key-value pair but an SNode is already occupying the location.

a. If the key of the existing SNode matches with the new key, then we simply replace the old value with the new one.

b. If the keys do not match and the ANode is narrow (only 4 entries), then the ANode is expanded to 16 entries. To do this, the ANode is frozen so that no other threads can modify it, and a new wide ANode is created and the narrow ANode values are copied over. The new key-value pair can then be inserted.

c. If the keys do not match and the ANode is wide, then a new level is created at the corresponding location by adding a new ANode. This expands the trie and the key-value pair can then be inserted at this new level. If two different keys have identical hashes, also known as a hash collision, then special List Nodes like those in CTrie can be used to resolve this. We are currently not adding this to our implementation due to time constraints.

The lookup and insertion operations are very similar. In the lookup operation we must traverse the trie to find an SNode that matches a given key and return the corresponding value. The search continues until an empty entry is reached or the SNode is found. Lookup operations do not have to wait on pending operations from other threads. If the search encounters a special node (ENode, FNode, FSNode, FVNode) then it can extract the necessary information from the special node to complete its operation. This makes the lookup operation wait-free. These are the operations that we are currently working on. Future operations/features that we plan to add are remove and List Nodes for hash collisions.

V.Analysis

//TODO

VI.Conclusion

//TODO

References

[1] Rene De La Briandais. 1959. File Searching Using Variable Length Keys. In Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference (IRE-AIEE-ACM '59 (Western)). ACM, New York, NY, USA, 295–298. https://doi.org/10.1145/1457838.1457895

[2] Edward Fredkin. 1960. Trie Memory. Commun. ACM 3, 9 (Sept. 1960), 490–499. https://doi.org/10.1145/367390.367400

[3] Aleksandar Prokopec, Phil Bagwell, and Martin Odersky. 2011. LockFree Resizeable Concurrent Tries. Springer Berlin Heidelberg, Berlin, Heidelberg, 156–170. https://doi.org/10.1007/978-3-642-36036-7—11

[4] N. Shafiei. 2013. Non-blocking Patricia Tries with Replace Operations. In 2013 IEEE 33rd International Conference on Distributed Computing Systems. 216–225. https://doi.org/10.1109/ICDCS.2013.43

[5] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. 2012. Concurrent Tries with Efficient Non-blocking Snapshots. (2012), 151–160. https://doi.org/10.1145/2145816.2145836

[6] P.G. Jensen, K.G. Larsen, and J. Srba.

2017. PTrie: Data Structure for Compressing and Storing Sets via Prefix Sharing. In Proceedings of the 14th International Colloquium on Theoretical Aspects of Computing (ICTAC'17) (LNCS). Springer, 1–18.

[7] Michael J. Steindorfer and Jurgen J. Vinju. 2015. Optimizing Hasharray Mapped Tries for Fast and Lean Immutable JVM Collections. SIGPLAN Not. 50, 10 (Oct. 2015), 783–800. https://doi.org/10.1145/ 2858965.2814312