Final Report: Cache Trie Data Structure

Rebeca Amaya
University of Central Florida
rebeca_amaya@knights.ucf.edu

Melissa Gramajo
University of Central Florida
mgramajo@knights.ucf.edu

Emily Hok
University of Central Florida
ehok@knights.ucf.edu

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 techniques and compare-and-swap operations.

1 INTRODUCTION

Hash tries are data structures that are used to store data by mapping hash values to keys. This data structure is beneficial for achieving efficient lookup, insertion, and/or deletion operations as it can attain nearly hash-table speeds and can grow dynamically without the need for periodic re-sizing [10]. 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. However, a major downfall of hash tries is that they have high space complexity. The memory needs to create enough space for every node and pointer. Nevertheless, the efficiency of the hash trie operations makes it a useful and popular data structure.

Concurrent lock-free hash tries improve the original data structure by allowing multiple operations to be called simultaneously, such as in Ctrie [3]. This data structure was the first non-blocking concurrent hash trie with O(log N) operations. CTrie has better performance than Skip Lists and is only slightly slower than hash tables, making it a powerful concurrent data structure.

The introduction of a cache data structure to CTrie, further improves its space complexity and overall run-time. This modified data structure is called the Cache Trie [9]. Ctries 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.

The original Cache Trie was implemented in Scala, a high-level functional programming language. We introduce a re-implementation of this data structure in C++, which allows for increased granularity in memory control.

2 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 values get inserted by their hash, the array gets populated with pointers to single nodes using CAS operations.

If a different value with the same hash 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 values are occupying the array. The new array will hold the previous value and the second value after a successful CAS at the first level.

Now suppose another thread inserts another value but causes 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 an additional cache data structure 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. This cache is then used to reduce the time complexity of the trie operations by allowing threads to skip O(log N) steps, therefore making the operations O(1).

3 RELATED WORKS

The first trie data structure was introduced in 1959 by Briandais [1], although it had not been officialy named named yet. Its purpose was to aid in the localization of records in a slow memory medium through lookups with keys. This data structure was originally designed to work with strings and 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 by freezing trie nodes, making them un-modifiable. 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 stored 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.

Similar to the PTrie, 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 as a result demonstrate performance improvements.

4 DETAILED IMPLEMENTATION

4.1 Node Types

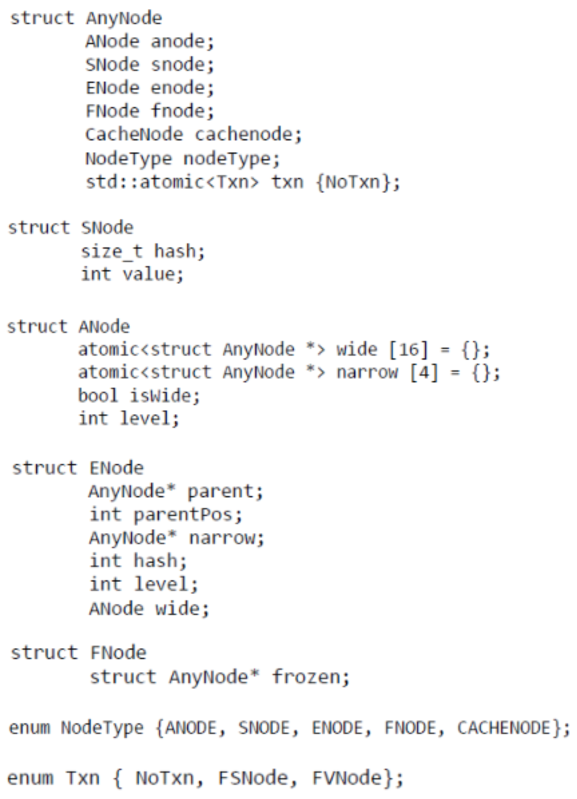
The concurrent Cache Trie implementation, proposed by Prokopec, was originally created in Scala. Moreover, part of the challenge was finding an equivalent representation of certain data types in C++. For example, we had to find an alternative to the Any class, which is a special class that sits on top of the Scala class hierarchy. To replicate the functionality of the Any type described in the paper, we created the struct AnyNode whose sole purpose is to wrap a resource of any type. The purpose for this functionality 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 structs, respectively. We implemented our version of the Cache Trie to store integers values. These values are stored in the SNode struct. The SNode also holds an unsigned integer hash, which tells the Cache Trie where to insert and how to retrieve the leaf node.

The ANode struct represents the inner node and is not associated with a hash. Instead, each inner node contains two arrays of pointers that can point to nodes of any type. One array is of length 4 and is called the narrow array. The second array is of length 16 and is called the wide array. Moreover, by making both arrays of type AnyNode, these arrays will be able to hold nodes of all types. Additionally, each inner node refers to a specific level of the Cache Trie, which grows in multiples of 4. To better illustrate this, let the root ANode begin 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 struct 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 structs in addition to the SNode and ANode are necessary. These secondary node structs 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 is occurring, however, it also prevents changes to the SNode. Finally, the FVNode struct 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. It is important to note that the FSNode and FVNode are not structs like the other nodes, but instead are denoted as enums which serve as flags to indicate a node state. There is an additional node classification which is the NoTxn that indicates that a node can be modified. These enums are referred to as the node Txn.

The pseudo code for the different node types can be seen in Figure 1.


```
struct AnyNode
        ANode anode;
        SNode snode;
        ENode enode;
        FNode fnode;
        CacheNode cachenode;
        NodeType nodeType;
        std::atomic<Txn> txn {NoTxn};

struct SNode
        size_t hash;
        int value;

struct ANode
        atomic<struct AnyNode *> wide [16] = {};
        atomic<struct AnyNode *> narrow [4] = {};
        bool isWide;
        int level;

struct ENode
        AnyNode* parent;
        int parentPos;
        AnyNode* narrow;
        int hash;
        int level;
        ANode wide;

struct FNode
        struct AnyNode* frozen;

enum NodeType {ANODE, SNODE, ENODE, FNODE, CACHENODE};

enum Txn { NoTxn, FSNode, FVNode};
```

Figure 1: Pseudocode for node types.

4.2 Operation Implementations

Our Cache Trie supports two functions: insert and lookup. While the cache data structure and its repsective fast lookup function have been coded, they have not been thoroughly tested due to insufficient information on how to code other important functions that support the cache such as *sampleSNodesLevels*, *findMostPopulatedLevel*, and *countTrailingZeros*, which are referenced in the original Cache Trie paper [9]. It is for this reason, in addition to the time constraint, that the decision was made to test only the regular insert and lookup operations without the cache. The result of the implementation is a working CTrie that with more time, can be easily converted into a Cache Trie. This section will primarily focus on how insert and lookup were implemented.

4.2.1 Insert Operation. The insert operation takes an integer value to insert into the Cache Trie. Depending on the current node's type, different actions are taken. Firstly, if the current node is empty, the value will be inserted into the Cache Trie using CAS. If the current node is an ANode type, the insert will recursively call itself again while changing two parameters. To begin with, 4 will be added to the current level being passed in the parameter since the value is being inserted into the array at the next level. Secondly, the ANode residing in the entry will be passed as the current array. The parameters for the hash and the previous ANode remain the same.

If the current node is an SNode type, then a data collision has occurred. It will first check the node's Txn value to see whether other threads are modifying it. NoTxn indicates that no thread is modifying this part in memory, thus modifications are allowed. Moreover, there are three scenarios to consider:

(1) The value being inserted already exists in the Cache Trie.
(2) The current array is narrow.
(3) The current array is wide.

The first scenario involves an easy solution. That is, a CAS is done to update the old value with the new value. The thread will reattempt to insert the value if the CAS fails.

For the second scenario, the narrow array needs to be replaced by a wide array. To accomplish this, a new AnyNode is allocated. The new node will be of type ENode, which indicates that the array is in the process of being expanded. It will then proceed to call the function *completeExpansion* on the new wide array. The *completeExpansion* function calls another function called *freeze*, which iterates through the entire array and marks each array entry to prevent further modifications from other threads. Specifically, FVNode is inserted for empty entries, FSNode is inserted for entries that contain an SNode, and an FNode is inserted for entries that contain an ANode. If a thread encounters another FNode, it can help complete the freezing. Similarly, a thread can help complete expansion for another thread if it encounters another ENode. After *freeze* completes, *completeExpansion* will call another function called *copyToWide*, which will copy the old SNode value that was in the current array into the new AnyNode's wide array through CAS. The parent node will swap it's empty wide array for the new wide array, which now contains the previous value. Once *completeExpansion* finishes, it will recursively call insert to put the new value into the new wide array by passing the new wide array as the current array.

In the third scenario, the Cache Trie must extend to the next level since the current array is already of length 16. Furthermore, a new ANode will be created. Additionally, two new SNodes will be created. One of the SNodes will contain a copy of the value in the old SNode and will be inserted into the new ANode's narrow array using a CAS. Next, the second SNode will contain the new value and will be inserted into the narrow array. After, the new narrow array will replace the old SNode in the current array using CAS.

Additionally, it is possible for the current node to be an FSNode. In this scenario, the insert function will return false since the current SNode is frozen. It is important to note that hash collisions, which occur when two different values have identical hashes, can happen. While the original implementation of CTrie resolved this issue with a special ListNodes function, we are currently not adding this feature to our implementation due to time constraints and because hash collisions are rare.

4.2.2 Lookup. The purpose of the lookup operation is to return whether a value resides in the Cache Trie. To accomplish this task, the function takes in three parameters: a hash, level integer, and an ANode. First, the Cache Trie would need to determine the specific position in the array to search. This position is calculated by performing the bitwise-and operation using the three parameters. The Cache Trie then looks inside the current array using the calculated position. This node is then copied into a new AnyNode, which will be further processed based on its Txn value and node type. Furthermore, there are five scenarios to consider for the lookup operation:

(1) The node is 0 (which is the equivalent of null) and contains a FVNode for its Txn value.
(2) The node type is ANode.
(3) The node type is SNode.
(4) The node type is ENode.
(5) The node type is FNode.

In the first scenario, if the node is zero and the Txn value is FVNode, the Cache Trie returns zero because FVNode is the equivalent of an empty entry. The only exception is that FVNode is non-modifiable.

To address the second scenario, if an ANode is found, the Cache Trie will recursively call lookup since it will need to search the next level. The parameters passed to the recursive call with be the hash, the level with 4 being to added to it to denote an additional level, and a new node that contains the copied the contents from the ANode at the next level.

On the other hand, if the Cache Trie encounters an SNode as described in the third scenario, the Cache Trie no longer has to keep searching. However, the hash will need to be validated before returning the SNode's value. Otherwise, the hash is different and the Cache Trie will return 0.

Finally, the last two scenarios will have similar solutions. In the event an ENode is found, the lookup function will perform a recursive call and pass the hash, the level with 4 added to it, and a new node, which will have copied the content found in the next level's wide array. The solution is similar if an FNode type is found. The only difference is that instead of passing a new node, the current node's frozen array is passed in its place.

The lookup operation is wait-free as it does not have to wait on pending operations from other threads. If the search encounters a special node (ENode, FNode, FSNode, FVNode), it can extract the necessary information from the special node to complete its operation. This allows guaranteed progress for the lookup operation.

4.3 Progress Guarantee

This program guarantees progress to some threads, which is shown in the following scenario: if one thread fails, then another progresses. For this reason, this program doesn't guarantee fairness as well. Also, this program is lock-free but not starvation-free since it can fail repeatedly when attempting a compare-and-swap. Because we do not contain any locks, it is deadlock-free as well. Additionally, this program has a liveness guarantee as threads help finish other thread's pending operations and the lookup operation is wait free.

4.4 Correctness Condition

As this program is linearizable, we can guarantee correctness of the operations. By using CAS operations, no thread can accidentally overwrite the changes being made by another thread. The freezing techniques used also ensure that the trie can grow dynamically without values being lost.

4.5 Synchronization Techniques

The major synchronization techniques used in this data structure are the freezing of nodes and the use of CAS instructions when modifying the trie.

4.6 GCC Transactional Memory

In addition to the concurrent lock-free Cache Trie implementation, we also implemented a Software Transaction Memory (STM) version of this data structure using GCC. While the code remained largely the same, atomic variables and all instances of CAS were removed. The critical sections once protected by the atomic library were instead wrapped in transaction blocks.

5 ANALYSIS

The number of values, for a single thread, reached our desired fixed input of being able to insert 500,000+ nodes. However, when our program runs with multiple threads, we observed that the input size reaches a maximum transaction size of 20,000, run with a debugger, before a race condition is met indefinitely.. When each thread tries to push more than 20,000 nodes into the trie, a segmentation fault occurs that we believe is caused by a race condition in the code. Unfortunately, this was not removed nor resolved before the deadline for this project. However, we were able to see correct and successful use of the data structure when run with the GDB debugger [8].

We ran this program in both Window and Linux environments, with and without a debugger, and observed several results: In both the GCC TM version and the normal concurrency version, the program experiences a high level of segmentation faults when run with more than one thread. When run with the GDB debugger, the number of times it segfaults decreases significantly because with a debugger, time works differently, making it slower, thus not running into the race condition and segfaulting as often. It is also possible that the memory used when running with the debugger is different, so the program does not run into the memory issues.

Since we did not have a properly working trie before the deadline, due to the race condition found, we could not fully implement the cache. However, the foundation code for the cache was implemented, as well as the modified lookup function that uses it. It is not connected to the trie, though, and requires further testing. We also found it difficult to interpret the entire cache data structure with the information given in the original Cache Trie paper as they did not provide clear instruction for its implementation.

5.1 Changes from original design

In the original data structure, there were several types of nodes that were set to each other. We created a wrapper struct that mirrored the 'Any Type' Scala variable to C++. This caused a challenge, due to the fact that there was a circular dependency. To solve this issue, a CMake file was created that ran all the dependency files before execution.

5.2 Obstacles

Since this program is lock-free, the data structure relies on garbage collection to function properly [11]. The original data structure was created in Scala, which has implicit garbage collection. When re-implementing the Cache Trie in C++, our team encountered several obstacles in completing the trie and cache data structures. Because the Cache Trie is based off of the CTrie, and the CTrie was originally developed for JVM, we were were unable to achieve a correctly functioning cache. With the time given and due to our learning curve, our team could not get this part of the data structure to work in its entirety, though minor parts of it were implemented.

Another major obstacle encountered was a race condition in our trie. This blocked us from advancing in running multiple threads with a larger fixed input size. A way around it was running our program with the GDB debugger, as it hides the race condition and allows the program to complete successfully.

5.3 Ways to Improve our Design

As the data structure relies on garbage collection, it may be better to implement this data structure in a high level programming language like Java. Both Scala and Java are high level programming languages with implicit garbage collection, while C++ is a lower level programming language that allows a higher level of memory manipulation. Another way to improve would be to add explicit garbage collection to the C++ implementation to alleviate any issues caused by faulty memory management.

6 PERFORMANCE EVALUATION

6.1 Testing Environment

Testing was conducted for both the Concurrent Cache Trie and the GCC TM Cache Trie on a 2.60 GHz Intel i7-6700HQ four-core processor using the Ubuntu 16.04.3 release Linux subsystem on a Windows 10 machine. Due to a race condition in the code, the testing was run using the GNU Debugger, GDB, to avoid segmentation faults. This is because the debugger slows down the execution time, and the time-sensitive race condition is not triggered.

6.2 Results & Discussion

Nine test scenarios were created using three operation distributions (50/50, 75/25, 25/75) and three transaction sizes (5000, 10000, 20000) with 1, 2, 4, and 8 threads. The threads performed insert and lookup operations on a shared Cache Trie data structure that held integer values. These values were inserted in ascending order, with each thread inserting a different set of values. The insert operations were performed first on an initially empty Cache Trie and the lookup operations were performed second. Each test scenario was run ten times and the average execution time was recorded. The results of the testing can be seen in Figures 2 – 10.

The GCC TM Cache Trie and Concurrent Cache Trie performed nearly identically, with the Concurrent Cache Trie performing slightly better, for the lowest transaction size (5,000) with a 50/50 operation distribution, as seen in Figure 2. Similar results can be seen for Figure 5, with a 10,000 transaction size and 50/50 operation distribution. However, for the remaining test scenarios, a more distinct performance difference can be seen between the two data structure versions. The GCC TM version performs slower than the Concurrent version, which is to be expected as STM can cause unnecessary transaction abortions when it detects possible memory conflicts.

Overall, the performance of the two data structure versions is relatively similar across transaction sizes and operation distributions. The Concurrent Cache Trie performed better with 8 threads at higher transaction levels than at lower ones. The performance difference between the two versions becomes more apparent as the transaction size increases, with the GCC TM Cache Trie performing slower.

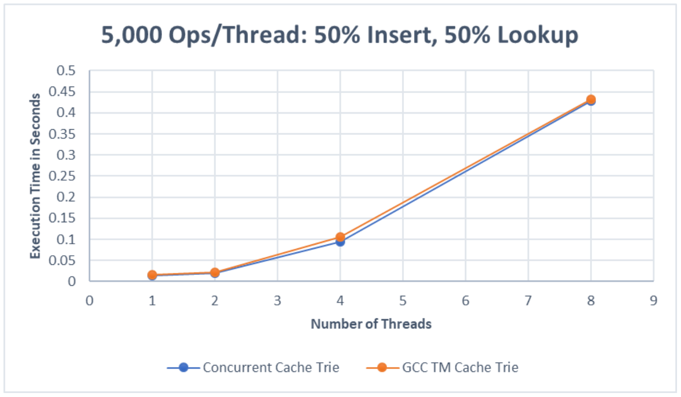


Figure 2: Execution time by number of threads for transaction size of 5,000 and 50/50 operation distribution.

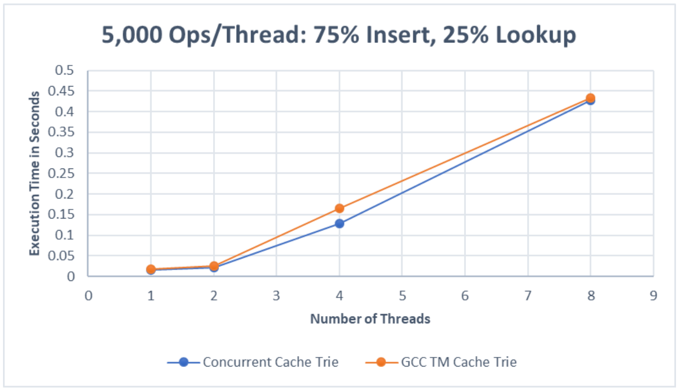


Figure 3: Execution time by number of threads for transaction size of 5,000 and 75/25 operation distribution.

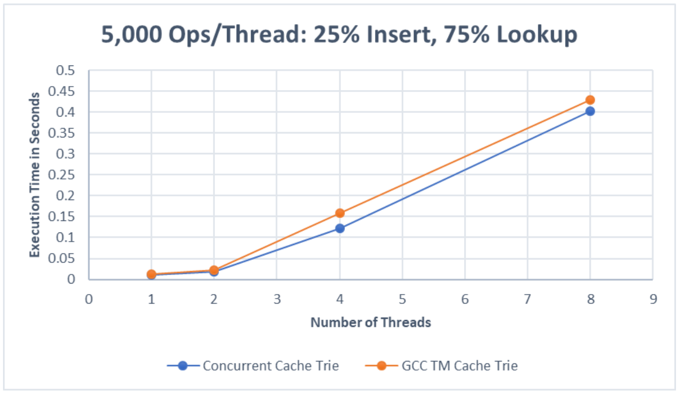


Figure 4: Execution time by number of threads for transaction size of 5,000 and 25/75 operation distribution.

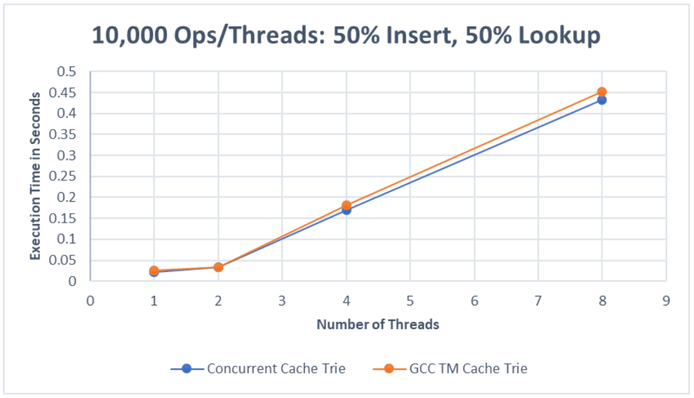


Figure 5: Execution time by number of threads for transaction size of 10,000 and 50/50 operation distribution.

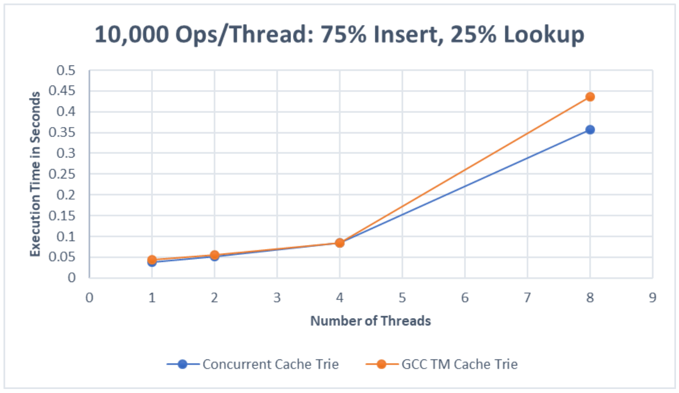


Figure 6: Execution time by number of threads for transaction size of 10,000 and 75/25 operation distribution.

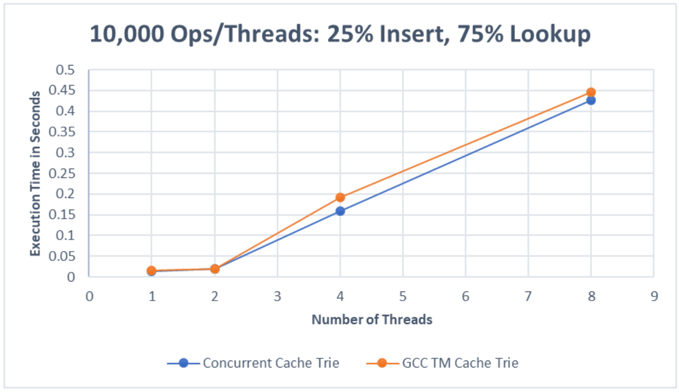


Figure 7: Execution time by number of threads for transaction size of 10,000 and 25/75 operation distribution.

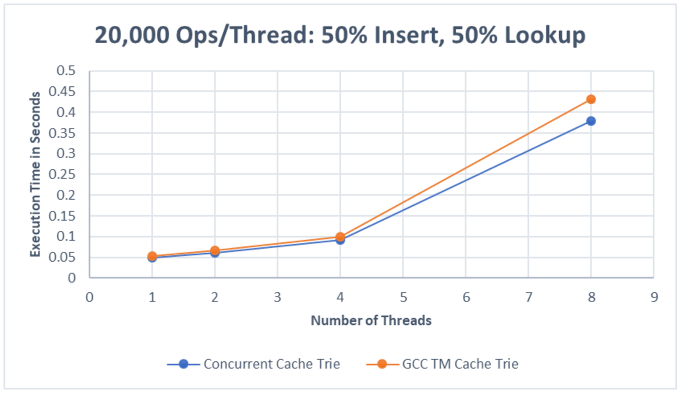


Figure 8: Execution time by number of threads for transaction size of 20,000 and 50/50 operation distribution.

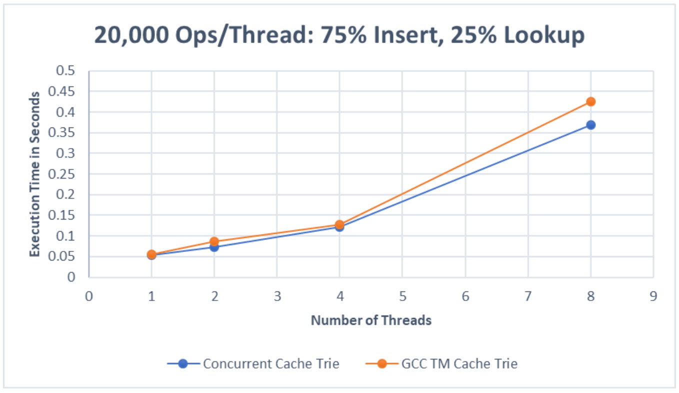


Figure 9: Execution time by number of threads for transaction size of 20,000 and 75/25 operation distribution.

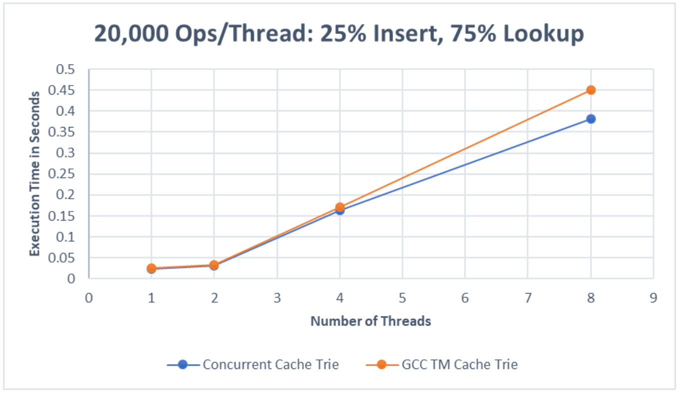


Figure 10: Execution time by number of threads for transaction size of 20,000 and 25/75 operation distribution.

7 CONCLUSION

In this paper, we presented a C++ re-implementation of the Cache Trie. Due to time constraints and learning curves, the cache data structure was not completed. The end result is a CTrie data structure with O(log N) operations with the foundations for being converted into a functioning Cache Trie with constant time operations.

It is important to note that this data structure would be better suited for a programming language with implicit garbage collection, such as Java or Scala. While C++ provides fast and efficient memory control, lock-free data structures tend to rely heavily on the automatic memory management provided by garbage collection. Moreover, the original CTrie and Cache Trie data structures were written for JVM. Translating the pseudocode from a high level programming language to a low level one resulted in unexpected issues.

Future work would be focused on finding and fixing the race condition in the current implementation as well as adding the cache. More research would also need to be done on applying explicit garbage collection to possibly alleviate the segmentation faults that afflict this re-implementation.

Our version of a lock-free concurrent Cache Trie in C++ helped us achieve a better understanding of the following: learning how threads and pointers operate in C++; how to handle circular dependencies in this programming language, C++ syntax, and other main topics encountered to make this program functional for this programming language. After testing our concurrent lock-free trie, we achieved expected results despite not including a cache. However, this program would be well suited for higher programming languages with automatic memory management, thus improving the overall performance.

8 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

[8] GDB: The GNU Project Debugger. Copyright Free Software Foundation, Inc. 19 November 2018.

[9] Aleksander Prokopec. 2018. Cache-Tries: concurrent lock-free hash tries with constant-time operations. *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, 2018), 137–151.

[10] Hash array mapped trie: 2018.
https://en.wikipedia.org/wiki/Hash_array_mapped_trie. Accessed: 2018- 11- 29.

[11] Ctrie: 2018. https://en.wikipedia.org/wiki/Ctrie. Accessed 2018-11- 29.