A Study on the FEAST Algorithm: Midterm Report

Natalia Colmenares
Department of Computer Science
University of Central Florida
Orlando, Florida USA
natcolchu@knights.ucf.edu

Aedan Deyto
Department of Computer Science
University of Central Florida
Orlando, Florida USA
aedandeyto22@gmail.com

Eduardo Bourget
Department of Computer Science
University of Central Florida
Orlando, Florida USA
eduardobourget@gmail.com

Joel Joy
Department of Computer Science
University of Central Florida
Orlando, Florida USA
joeljoy1912@gmail.com

***Abstract*—Abstract will go here.**

I. Introduction

There have been remarkable developments in the field of multiprocessor programming as well as multi-core systems in recent years. With the rise of such technologies, it is important to properly utilize the power of the multiple processors to operate on overlapping regions of the data structures at the same time. Concurrency has often been achieved by way of locks and lock-based algorithms being implemented in various data structures, such as queues, linked lists, hash tables, skiplists, and tries. When locks are being utilized in these data structures, a certain process can restrict access to a particular portion of the data structure for an extended period of time. Because of this, lock-based algorithms that implement these data structures are prone to deadlocks, and performance issues.

One of the fundamental data structures in computer science are Binary Search Trees (BSTs). They are used to store and organize ordered data and they support the insert, delete, and search operations. As it happens, generalized attempts to derive a concurrent non-blocking BST from its sequential version tends to be quite inefficient, but the FEAST (Fast Concurrent Algorithm for Binary Search Tree) implementation guarantees a lock-free manipulation of a concurrent BST in an asynchronous memory system that supports all of the essential BST functions: insert, search and delete. In stark contrast to most of the existing algorithms that implement a concurrent BST, the FEAST algorithm is edge-based rather than node-based.

The proposed FEAST algorithm by Natarajan et al. explores how to implement the lightweight lock-free binary search tree, but there is not a study on the FEAST algorithm implementation in other languages. In fact, the experimental evaluation for the FEAST algorithm was created with the Intel C++ compiler (version 18.0.2). Thus, the purpose of this paper aims to answer the following questions:

1. What kind of modifications to the program will be necessary to implement this algorithm in different programming languages?
2. Is the efficiency of the program affected when it is implemented in different languages? If so, is there a particular programming language that is substantially more efficient than others when implementing this algorithm?

II. Background and Related Work

A. Motivation

Binary Search Trees are data structures that support dynamic-set operations such as search, insert, delete, minimum, maximum, predecessor and successor, and are represented with key fields and node objects [3]. The node objects contain left and right pointers to their respective node objects. The support of such operations allows for an organized representation of data, where keys, the field in which the organization revolves around, satisfy the binary-search-property: Child nodes that have keys larger than the parent node's are expected to station themselves in the left subtree of the parent node, while nodes with keys smaller than the current parent node's are positioned to be in the right subtree of the parent. The efficiency of the dynamic-set operations, O(log n) time on average, where n is the number of nodes in a binary search tree [3], is one of the main reasons a concurrent representation of the data structure has been studied.

Due to the spread of multiprocessor architectures, studies on popular sequential data structures and how to represent them concurrently have already been implemented in program libraries; thus, research on them continues to progress. For example, the Harris-Michael algorithm as a representation of a concurrent linked-list is one used in the Java Concurrency Package [6]. There are various methods to implement a concurrent data structure by using a blocking or a non-blocking strategy. BSTs that have used locks have been tested and proven correct [5], but due to the use of locks, other operations, such as search, can be blocked. Blocking can prevent other threads from making progress if there is an unexpected delay in one thread [6]. Although a blocking implementation may

prove easier to implement and test, a nonblocking property is attractive since a delay in one thread may not result in preventing other threads from making progress. The wait-free property and the lock-free property are examples of nonblocking progress conditions. In a wait-free property, every thread that takes steps is guaranteed to make progress. In contrast, a lock-free property states infinitely often, at least one of the operations eventually completes. Although a wait-free property sounds generous, wait-free algorithms may prove to be slower than lock-free algorithms. Thus, "A fast lock-free algorithm may be more attractive than a slower wait-free algorithm" [6]. Current research on blocking BSTs include CASTLE [9], BST-TK [4], and CITRUS [1]. In addition, Non-blocking BSTs include RM-BST[8], Non-blocking Binary Search Trees by Ellen et al. [5], and FEAST [7].

FEAST is described as a lightweight lock-free concurrent binary search tree. Due to the non-blocking progress condition with a lock-free property, this algorithm was of the options studied to implement a concurrent BST. The work on FEAST presents an experimental evaluation with a few of the mentioned concurrent BST implementations above: CASTLE, BST-TK, CITRUS, and RM-BST. Using system throughput as an evaluation metric, the performances of the mentioned algorithms were compared with FEAST on a multi-core machine. Results of the experiment described FEAST as having the best performance out of the five BST algorithms with smaller key spaces when contention was high [7]. Thus, due to the superior performance of FEAST for small and medium key spaces, we decided to study the FEAST algorithm as a concurrent BST.

B. Overview of the FEAST Algorithm

The FEAST algorithm is a lock-free algorithm used for concurrent manipulation of a binary search tree in an asynchronous shared memory system that supports search, insert, and delete operations. In contrast to other concurrent algorithms, this algorithm is edge-based rather than node-based. This means that when compared to other algorithms for a binary search tree, the insert and delete operations will work on a smaller tree section, execute fewer instructions and/or use fewer dynamically allocated objects.

This algorithm uses an external representation instead of an internal representation. The main comparison between these two is that in the internal representation, the data is stored in every node: both internal and leaf nodes, while In an external representation, the data is stored only in leaves [2, 7]. The internal nodes are used for routing purposes only. The main advantage of using the external representation over the other is that it dramatically simplifies the tree's search, insert, and delete operations in a concurrent environment [7]. Both the insert and delete operations use a constant-size window and do not cause keys to move from one location in the tree to another. As a result, when looking for a key, the traversal does not need to start if the key is not found. It is better to choose the internal representation when looking for a smaller memory footprint and a better cache performance since it contains about half the number of nodes than the external representation.

Every operation of the algorithm will start with a seek phase, in which the operation traverses the search tree starting from the root node until it reaches a leaf node. The path traversed by the operation in this phase as the access-path and the last three nodes on that access-path as the grandparent, parent and terminal nodes, respectively. The operation then compares the key stored in the terminal node, on the access-path, and according to that result of the comparison and the type of operation, it will either terminate or move to the next phase.

The modifying operations like insert and delete obtain ownership of the edges it needs to work on. The delete operation will obtain ownership of an edge by marking it. By marking, it indicated that either both its tail and head nodes, or only its tail node would be removed from the tree. The first case is referred to as flagging and tagging as the second type. To enable this tagging or flagging of an edge, two bits from each child field stored at a node are stolen, denoting the two bits by flag and tag. A value of 1 in the respective field means that the corresponding edge has been flagged (or tagged). Contrastingly, a value of 0 exhibits a case where the edge is unflagged (or untagged).

To demonstrate the correctness of the algorithm, FEAST uses linearizability as the safety property and lock freedom as the liveness property. This means that the linearizability property requires that an operation should appear to take effect instantaneously at some point during its execution [6]. Having lock-freedom requires that if a process takes an infinite number of steps, then at least one operation eventually completes.

C. Details of the Algorithm

Seek **:** Most of the dynamic-set operations make use of the seek function to return the access-path from a given key. The access-path is described as a simple path from the root node to the leaf node. The ancestor, successor, and parent variables in the function are first initialized with the sentinel nodes in the tree for traversal. Traversal begins when the address of the next node (the current node's address field) on the access-path exists. With every iteration of the while loop, we attempt to extend the access-path by examining the state of the nodes and continuously updating the ancestor, successor, parent, and current nodes. Once the terminal node points to a leaf node, the loop breaks and the final values stored in the local ancestor, successor, parent, and terminal variables are returned in the form of a SeekRecord object. The SeekRecord class holds the ancestor node (the grand-parent of the terminal node), successor, parent, current, and terminal nodes. Functions make use of this class to be able to traverse and modify the tree effectively.

Insert **:** The insert operation runs inside a while loop and only returns when a successful insert occurs or through the identification of an existing object with the same key value. The function starts execution by calling the seek function to find parent and terminal nodes in the tree. Once the address of the child field is obtained through the parent node, two new

nodes are created: an internal node (newInternal) and a leaf node. The newIntermal node will contain the larger variable between the key to insert and the key of the terminal node, while the new leaf node will hold the key to insert. A subtree is then created to insert into the tree, where newInternal acts as a root and newLeaf and the terminal nodes as the child nodes. A CAS operation is then performed to attempt to insert the subtree into the tree. The CAS operation is only successful if the references did not change in the time between initializing new nodes and getting the address of the next child field, and if the stamp value is not tagged nor flagged. In other cases, if insertion fails due to changing states of a node being flagged or tagged, cleanup is called to delete the flagged nodes before attempting to insert again.

Delete **:** The delete function will mark the node that needs to be deleted and try to delete it. We do so by first setting the mode to injection: Mark the leaf node. Within the injection mode, a search for the key in the tree occurs by using the SeekRecord object called by the delete function. If the key is not present in the tree, the function immediately returns. Otherwise, we attempt to flag the edge of the leaf node and then try to physically remove the leaf node from the tree by calling the cleanup function. At this point, the mode is transferred to cleanup. The delete operation is persistent with physically deleting the flagged node during cleanup mode and will not return until successful.

Cleanup **:** Cleanup will physically remove leaf nodes and its parent from the tree when marked for deletion. Compared to all the other main functions in the algorithm it will need to be sent a SeekRecord object instead of a key. Using the SeekRecord object, it will fill in its local nodes of ancestor, successor, parent, and terminal nodes. Then it will create the variables addressOfSuccessorTheField, addressOfTheChildField, and addressOfTheSiblingField to hold an edge. The addressOfSuccessorTheField will contain the edge between the ancestor and successor. The addressOfTheChildField will get the edge between parent and terminal, and the addressOfTheSiblingField will get the other edge that parent holds. After, we get the stamp of addressOfTheChildField, if the stamp is not flagged, the sibling field needs to be deleted, so set the addressOfSiblingField to addressOfTheChildField. Then, we check if the stamp of addressOfTheChildField is tagged. If tagged, we try to tag the addressOfSiblingField. Subsequently, we attempt to make the sibling node a direct child of the ancestor node with a CAS operation. The result of the CAS operation is returned as the return value of cleanup.

III. Methodology

Describe testing plan and evaluation metric.

A. Java Implementation

The supported dynamic-set operations in our lock free BST algorithm are seek, input, and delete. The components of the tree are a node which contains the key, and Java's AtomicStampedReference class for the left child and right child of the node. The stamped value will represent if the edge between the node and its individual child have been tagged or flagged. In our code, we agreed to use a two digit system based on 1's and 0's. The first digit describes the flagged state and the second digit represents the tagged state. The number 00 represents a stamp that is neither tagged nor flagged. In contrast, 10 is used to mark that it is flagged, but not tagged. The number 01 illustrates that it is tagged and not flagged, and 11 is used to represent a stamp that is both tagged and flagged. The decision to use a digit representation for the states was chosen because AtomicMarkableReference allowed only one markable bit. Additionally, the algorithm always compares a clean state (00) in a CAS operation; thus, using a digit representation does not provide any setbacks. In addition, three sentinel keys are present and never removed from the tree. The sentinel keys must be greater than any other keys.

B. C++ Implementation

Define C++ Implementation.

C. Rust Implementation

Define Rust Implementation.

IV. Experimental Expectations and Future Work

A. Experimental Expectations

With the completion of the algorithm in Java, we plan on testing the algorithm for correctness using one thread to determine if the algorithm was implemented correctly, and then multiple threads following the correctness condition of linearizability. The algorithm will then be tested with the unit of Time to determine how efficient it is in different sized sets of data. We expect the algorithm to perform better in medium sized sets of data than those of a larger size.

B. Future Work

Our plan for the rest of the project deadline is to test the algorithm in two other languages. Languages that are being considered include Rust, and C++. Choosing to study the algorithm in other languages can help determine when and where it is useful to implement the FEAST algorithm. In fact, we expect the algorithm to perform better in C++ since it is a lower level language than Java. Rust being a newer language built to solve problems encountered in C/C++ may have the same performance as C++. The implementation in Java alone had differences in the implementation from the one proposed by Natarajan et. al, therefore we also expect performance differences. We also seek to answer the following: Is the FEAST algorithm able to be translated in other languages using minimal structural modifications?

V. Evaluation and Testing

Evaluation and testing will be provided here.

VI. Results

Results will go here

Use tables to show evaluation. Table above is provided by the OverLeaf IEEE template.

TABLE I
TABLE TYPE STYLES

Table Head	Table Column Head		
	Table column subhead	*Subhead*	*Subhead*
copy	More table copy[a]		

[a]Sample of a Table footnote.

VII. CONCLUSION

Conclusion will go here after research has been completed.

REFERENCES

[1] Arbel, M., AND Attiya, H. 2014. Concurrent updates with RCU: Search tree as an example. In Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing (PODC'14). ACM, New York, NY, 196–205.

[2] Arbel-Raviv, M., Brown, T., AND Morrison, A. 2018. Getting to the Root of Concurrent Binary Search Tree Performance. Proceedings of the 2018 USENIX Annual Technical Conference.

[3] Cormen, T. H., Leiserson, C. E., Rivest, R. L.,, AND Stein, C. (2001). Introduction to Algorithms. The MIT Press. ISBN: 0262032937

[4] David, T., Guerraoui, R., AND Trigonakis, V. 2015. Asynchronized concurrency: The secret to scaling concurrent search data structures. In Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15). ACM, New York, NY, 631–644.

[5] Ellen, Fatourou, P., Ruppert, E., AND van Breugel, F. (2010). Non-blocking binary search trees. Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, 131–140. https://doi.org/10.1145/1835698.1835736

[6] Herlihy, M., AND Shavit, N. 2008. The Art of Multiprocessor Programming. Morgan Kaufmann. ISBN: 0123705916

[7] Natarajan, A., Ramachandran, A., AND Mittal, N. 2020. FEAST: A Lightweight Lock-free Concurrent Binary Search Tree. ACM Trans. Parallel Comput. 7, 2, Article 10 (June 2020), 64 pages. DOI:https://doi.org/10.1145/3391438

[8] Ramachandran, A., AND Mittal, N. 2015. A fast lock-free internal binary search tree. In Proceedings of the 16th International Conference on Distributed Computing and Networking (ICDCN'15). ACM, New York, NY, Article 37, 10 pages.

[9] Ramachandran, A., AND Mittal, N. 2015. CASTLE: Fast concurrent internal binary search tree using edge-based locking. In Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'15). ACM, New York, NY, 281–282.