Concurrent Tree Traversals in Binary Search Trees

Robert Bland
College of Engineering and Computer Science
University of Central Florida
Orlando, FL

Tyler Townsend
College of Engineering and Computer Science
University of Central Florida
Orlando, FL

***Abstract*—We introduce our reimplementation of what is considered a practical and concurrent binary search tree that maintains logical ordering information within the data structure, but our implementation will be transformed and compared against a transactional data structure. A transactional data structure will be implemented using software transactional memory which provides a system for executing atomic sections of code instead of using locks. This will allow for users to monitor memory locations that threads read and write to.**

***Index Terms*—Search Trees, Concurrency, AVL, Data Structures**

I. Introduction

With the increase in the number of cores on the processor, concurrent datastructures become the foundation to which these cores may be utilized otherwise they become a bottleneck of contention [4]. Implementing concurrent search trees becomes challenging when modification to the tree can cause relocation of some of the nodes, especially when the trees are optimized for read operations [**?**].

The Draschler-Cohen et al. of [**?**] provide a necessary and sufficient condition for path validation in traversals which they call *PaVT* condition. This condition applies to *any* search tree can validate whether a key is present in the tree or not. They demonstrate that the using the PaVT condition with a Binary Search Tree (BST) and an AVL tree, the out performs existing solutions using a lock-free contains on search-based data structures.

However, evaluating this condition leveraging software transacitonal memeory to manage concurrent accesses has not yet been shown. Concurrent trees using STM are easy to implement and scale well, but STM introduces overhead and where high contention exists still requires some reasearch [3]. Transactional memory removes the complexity of locks [2]. Balancing the usefulness of the simple resource sharing and performance is a hard task and the transactional memory implementation emulates just how much has to be sacrificed for reduced complexity. Transactional memory can significantly lower the possibility of data races and deadlocks [2].

This paper wishes to show the reimplimentation of the author's BSTs (including AVL) which leverage the PaVT condition. Further extending their work, we implement the BSTs using GCC's software transactional memory (STM) framework and compare the performance of the STM implementation to the BST implementation. The paper will provide a concise background of the PaVT condition presented by the author's work and demonstrate its use in BST's. The implementation details are provided in the following section. Section IV will exam the performance evaluation of all the data structures followed by a brief discussion in section V.

II. Background

A. Path Validation Traversal in Search Trees

Path **V**alidation traversals in search **T**rees is a necessary and sufficient condition for determining whether a key is existent in any tree. It satisfactory to observe only a limited number of nodes in the path leading to this key defined as the succinct path snapshot (SPS). A path is suitable for a key k if the condition c_i associated with every $node - field$ pair in the path. For a BST the conditions are $C = \{<, >\}$ and field $F = \{L, R\}$. Thus a path from the root, n_0, to a an arbitrary node n_k is $P = (n_0, f_0), (n_1, f_1), \cdots, (n_k, f_k)$ where $n_k.f_k = null$ denotes that n_k is the last node in the path as $f_k \notin F$. The author's formal defintion of the PaVT condition stated in Theorem II.1.

Theorem II.1. *(The PaVT Condition) Given a tree T a key k, and a set of node-condition pairs, $S = \{(n_{i_1}, c_{i_1}), \cdots, (n_{i_m}, c_{i_m})\}$, if there exists a moment beteween the traversal's invocation and response where all the following hold:*

1) *For every $(n_i, c_i) \in S, c_i(n_i, k) = 1$*
2) *$n_{i_1}, \cdots, n_{i_m}$ are logically in T*
3) *There is a path in T linking these nodes*
4) *The path is maximal*
5) *For every node n logically in T, either no node in S is reachable from it or there exists a pair $(n_i, c_i) \in S$ where n_i is reachable from n via a field f, and $c_i(n_i, k)$ implies a condition $c(n, k)$ such that $c \implies f$ is the next field.*

Then there is no node logically in T with a key k.

Applying this to a BST, if there is a path $P = (10, L), (8, L), (3, R), (4, R)$, then the suitable keys for this path are $k \in \{5, 6, 7\}$. If this path was taken to search for key $k = 9$, then some mutation must have occured and the traversal would need to restart.

B. Succinct Path Snapshot

Maintaing a full path is difficult as part of the path may have an inconsistency from some other mutation. Therefore, the authors formally define a succinct path snapshot which decouples 1) from the PaVT, and allows for concurrent mutations to change nodes that link the snapshots, but do not affect the snapshot. There is a transitivity property for the paths as we traverse them. For example $P = (10, L), (8, L), (3, R), (4, R)$ can be reduced to $P = (8, L), (4, R)$ since $5 > 4 \implies 5 > 3$ and $5 < 8 \implies 5 < 10$. This would be the maximal set of nodes with respect to all the conditions for a BST i.e, $(<, L), (>, R)$. This is known as the *succinct path snapshot* (SPS). The author's formal definition is stated below.

Definition II.1. (Succinct Path Snapshot) Let $P = (n_0, f_0), \cdots, (n_m, f_m)$ be a path and $C' = (n'_0, c_0), \cdots (n'_k, c_k)$ be a node-condition series corresponding to $P.S \subseteq C$ is a succinct path snapshot (SPS) of P if:

1) $n'_0 \cdots, n'_k$ are logically in T
2) There is a path in T linking these nodes which is maximal if $n'_k.f_k = null$
3) For every node n logically in T, either no node in S is reachable from it, or there exists a pair $(n_i, c_i) \in S$ where n_i is reachable from n via some field f such that for every k, $c_i(n_i, k)$ implies $c(n, k)$ that returns f for all valid keys $w.r.t$ S of n.
4) For every $(n_i, c_i), (n_j, c_j) \in S$ such that $j > i$ and for every key k, $c_j(n_j, k)$ does not imply $c_i(n_i, k)$ for some valid key w.r.t. S of n_i.

The validation of an SPS is not only sufficient but necessary for unsuccessful traversals. The author's state that for BST's, the snapshots can be optimized by observing that for any node n, that n's left snapshot S_L, and n' right snapshot R_L consist of its predecessor and successor, respectively, in the logical ordering of the tree [6]. Our implementation uses pointers to maintain a reference to the predecessor and successor for a node. In order to ensure that all node's snapshots are of the same size, sentinel nodes are used to maintain order.

C. Correctness

Correctness of maintaining snapshots follows from the fact that if an update to a path is occuring, all locks must be acquired before the operation. Therefore, the snapshots are guaranteed to reflect those of the path.

D. Linearizability

Snapshots are read and written to atomically which allows them to be linearized. The linearization point for a snapshot is the moment it is an SPS of a path in the tree. These snapshots can be linearized with respect to the operations. If the snapshot were to contain a node that has been removed, it is linearized just prior to the removal. If the snapshot does not contain a node that is now reachable via an insertion, it is linearized just before it was added. The idea is that if the snapshot has been read, then it is from a path that existed in the tree, and the fact that it is read means that the operating thread performing a mutation has not completed it's operation.

III. Data Structure Design Principles

The author's paper presented the topic of *PaVT* and provided little detail on the implementation of the data structure. Available pseudocode was used to construct the methods necessary for the BST data structure. Since only a brief description of the AVL rebalancing property was mentioned, an attempt to construct the AVL function was taken. We contacted the author's about some synchronization issues not explicitly stated in the paper, particularly about updating snapshots. As of Nov. 8, 2018, the source code for the author's implementation is available at https://github.com/logicalordering/trees. We borrowed the synchronization for snapshots, particuarly, for removal from their source code. Thus our source code is an amalgam of their pseudo and source code. In the following subsections, we will discuss the design of our implementation, including the main algorithms.

A. Tree Structure

Our tree structure implements both the Binary Search Tree ordering property, but provides a boolean argument to the constructor to specify the AVL structure property. The AVL operation **rebalance** is performed at the end of the **insert** and **remove** operations of the tree. This allowed the code to follow the *DRY* principle. The remaining functions **traverse** and **contains** are almost identical except that **traverse** will lock the node to be returned while **contains** returns a boolean value.

The tree has four member variables; three of *Node* and the fourth is the *AVL* boolean. The *Node* class can be seen in Figure **??**. The key values are represented as integers for simplifying the data structure requirements. In order for the structure to meet the correctness requirements, the Node class must allow the node to be marked. This will depict a logical removal when $n.mark = true$. This is done right before a node becomes unreachable. The keys are immutable, so nodes cannot be recycled. This prevents nodes from needing to be rechecked. If a node is logically in the tree, it is reachable.

We store the snapshots of the node as atomic pointers. This allows the snapshots to be linearized when they are read. Each node also a parent pointer as well as left and right pointers to its children, which allows traversing up the tree. A height field is provided to store the current height of the tree. A tree with no node has height -1, a single node, 0, and the height of the tree is the maximum of the left, and right subtrees + 1.

The tree has four member variables; three of *Node* and the fourth is the *AVL* boolean. Two of the three pointers are sentinel nodes. The sentinel nodes are represented as the maximum and minimum, signed 32-bit integers in C. This does give some restriction on the number of keys to use, but this was done as the testing environment would only handle a maximum of 5,000,000 nodes. The snapshots of the sentinel nodes contained each other. The minimum sentinel was the parent of the max sentinel and the max sentinel was also the root of the tree, the third pointer.

```
class Node {
  const int data;
  bool mark;
  std::mutex lock;
  std::atomic<Node *> leftSnap;
  std::atomic<Node *> rightSnap;
  Node *left;
  Node *right;
  Node *parent;
  int height;
}
```

Fig. 1: The Node data structure implemented within the BinarySearchTree class

B. Locking Protocol

The locking order of the lock's is such that if n_1 and n_2 are nodes in the tree, then n_1's lock is acquired before n_2's lock if: (1) n_2 is reachable from n_1, (2) n_1 and n_2 are reachable by a field f_i and f_j, respectively from their lowest common ancestor and $i < j$. This is an essential top-down, left-to-right locking order. The implementation of the data structure; however, will lock the node to be removed, n, prior to locking the parent's node, p. In this case, an optimistic attempt will be taken to acquire the lock. If it fails, then both lock's are realeased and the operation is reattempted.

This protocol is livelock and deadlock free since locks are acquired in the same order by all threads. At least one thread can make progress since the lowest node in the tree (level-ordering) is guaranteed to acquire all necessary locks.

C. Traverse/Contains

Algorithm 1 starts from the root of the tree and uses a helper function **nextField** to reach the next node. If the node is the last in the path, `nextField` will be such that if $f = \mathbf{nextField}(n, key)$ then $n.f = null$. If the node with key is found then the function returns true if the node is not marked. If it is the function restarts. For **traverse** the node is locked just before line 8. Since the thread waits for the lock, we check if the node has been marked for deletion. If so, the node is unlocked and the procedure is restarted. If the end of the path is reached, the last node in the path is locked and snapshot for the node and field is read atomically. If the key is not in the node's snapshot, the procedure is restarted otherwise the node is returned. And for **contains**, the function simply returns false.

Being read atomically allows the snapshot to be linearized when either (i) the unmarked node with the key is found or (ii) the linearization point of the snapshot read at line 14. For (i) since the node was found and locked, this node is currently reachable in the tree. The read snapshot from (ii) guarantees that this snapshot captures the suitable path for the key. If it has not, then a mutation has occurred to alter the path such that the current thread would have traversed to the current node which indicates the procedure to restart.

Algorithm 1 Contains (key)

```
1: $n \leftarrow root$
2: $f \leftarrow \mathbf{nextField}(n, key)$
3: $next \leftarrow n.f$
4: while $next \neq null$ do
5:     $n \leftarrow next$
6:     $f \leftarrow \mathbf{nextField}(curr, key)$
7:     if $n.k = key$ then
8:         if $n.mark = true$ then
9:             restart
10:        return true
11:    $next \leftarrow n.field$
12: $S \leftarrow (field = L\ ?\ n.S_L : n.S_R)$
13: if $(f = L$ and $key \notin (S, n))$
14: or $(f = R$ and $key \notin (n, S))$ then
15:     restart
16: return false
```

Algorithm 2 Insert (key)

```
1: $n \leftarrow \mathbf{traverse}(root, key)$
2: if $n.k = key$ or $(key > n.k$ and $n.R \neq null)$ or $(key <$
   $n.k$ and $n.L \neq null)$ then
3:     $unlock(n)$ then restart
4: $newNode \leftarrow \mathbf{new}Node(key)$
5: $newNode.P \leftarrow n$
6: $S \leftarrow (n.k > newNode.k\ ?\ n.S_L : n.S_R$
7: if $n.k > newNode.k$ then
8:     $newNode.S_L \leftarrow S$
9:     $newNode.S_R \leftarrow n$
10:    $S.S_R \leftarrow newNode$
11:    $n.S_L \leftarrow newNode$
12:    $n.L \leftarrow newNode$
13: else
14:    $newNode.S_L \leftarrow n$
15:    $newNode.S_R \leftarrow S$
16:    $S.S_L \leftarrow newNode$
17:    $n.S_R \leftarrow newNode$
18:    $n.R \leftarrow newNode$
19: $unlock(n)$
20: if $AVL$ then $\mathbf{rebalance}(n)$
```

D. Insert

Algorithm 2 begins by calling **traverse** starting at the root for key. If the node returned, n, has a key equal to key or the node is no longer the end of the path (i.e., another insertion has occurred on the same node) then the procedure is restarted. Otherwise a new node $newNode$ is allocated with its parent pointer set to n. Snapshots are updated based on the value of the key. The snapshot for n and n's snapshot, S must be updated so that $newNode$ is contained in their snapshots while $newNode's$ snapshots consists of both n and S. This is done prior to making $newNode$ reachable from n which is the moment of linearization.

E. Remove

Algorithm 3 traverses to the node to be removed. If $n.k \neq key$, then by the PaVT condition, the tree does not contain key and it returns. Otherwise, an optimistic attempt to lock the parent is taken. If it fails, the procedure restarts. Once both are locked, the possible scenarios are:

1) n is a leaf
2) n has one child, c
3) n's right child, r, does not have a left subtree
4) r has a left subtree which contains n's successor s

The first case is simple as all locks are acquired. The node is marked and the snapshots are updated, S_{max}, and S_{min}. The corresponding snapshot and p snaps are updated to contain each other as seen starting at line 8. If n has one child, then c takes n's place. Now n's snapshots will be updated to contain each other at line 23. If the node has 2 children, but $r.L = null$, then r is n's successor and r takes n's place. The most difficult remove is derived when s is in r's left subtree. However, since r has a reference to it, locking the node is done directly along with s's parent p_s, its right child, r_s, and s's right snapshot $(S_R)_s$. These are locked using the locking order. It is important to check conditions such that $s.P \neq r$ since r's lock is acquired. If this is carried out then the thread will block itself. After acquiring all the locks, the removal can take place and the snapshots are updated at line 46. Snapshots during a remove require the left and right snapshots of n to update such that $S_L.S_R = S_R.S_L$. This removes n from the logical ordering. AVL rebalancing is performed at the end of the algorithm if necessary. Since the algorithm only proceeds once all locks are acquired, $remove$ is linearized when the node is marked. If **contains** returns true on n, then the node has not been marked. If the node is found marked, then the operation simply restarts.

F. Rebalance

The **rebalance** operation is implemented based on the author's description [**?**]. This algorithm starts at the specified node, n and traversed to the root. It grabs it's parent, p, and lock's both in the described locking order. If now calculates the node's height, h', and check if its member variable height, $n.h$ is different. If so, there has been a mutation, and the new height is updated. The balance factor is calculated bf, which is the difference between the left and right subtrees. If the height has not changed and $|bf| < 2$, then we return which is our implementation of the relaxed rebalancing. Rather than traversing entirely to the root. If $bf \geq 2$, then we must rotate. Helper functions **rotateLeft()** and **rotateRight** are implemented to carry this out. If $bf < -1$ and the left child of n has a positive balance factor, then a double rotation is required, and conversely for $bf > 1$. The nodes necessary for this operation will be the child and its grandchild. Locks are acquired from parent to grandchild. Since the rotations will now reorder the nodes, the locking order changes when they are released. The code is implemented in a straight forward manner to illustrate this. However, rotations for m-arry trees perserve the logical ordering which in turn preserves the snapshots, hence they will not need to be updated. If an insert or remove is interacting with this operation, the linearization point of this occurs once all the locks are released for every traversal up the tree. For a contains, the linearization point occurs during the last rotation. Any incorrect traversal will be reflected by the snapshots.

G. STM Implementation

1) GCC Transactional Memory: GCC's STM framework is used for the STM BST implementation. GCC transactional memory is used to create blocks of code that will be executed atomically. If another transaction block (or thread) causes a conflict on a memory access with the current block then we abort and rollback all modifications and try the transaction. This transactional memory also has specific properties associated with it as well that can be applied to any variable or function. The two properties are defined as transaction safe and transaction unsafe. For the transaction unsafe property, it is assumed if the code has or contains the following [2]:

- it is relaxed transaction statement (we don't use this)
- it contains an initialization of, assignment to, or a read from a volatile object
- it is unsafe asm declaration
- it contains a function call to a transaction-unsafe function, or through a function pointer that is no transaction-safe
- it has a volatile variable (so no atomic or volatile variables)

The transaction safe property, however, will be allowed to execute inside the atomic block of code if it isn't declared transaction unsafe. There are even more rules to what is exactly transaction safe and unsafe, but that is very complicated and some irrelevant, so a reference is provided for knowing more about this [2].

2) Implementation Breakdown: then modifying it by surrounding code that will be shared or expected to be modified by other threads in a transaction atomic block. The sequential version of the GCC transactional memory implementation used this style, but in a very broad sense. The sequential implementation involved the removal of all atomic wrappings, so no atomic variables, and the removal of all mutexes, so removing all locks and unlock calls from the code as well. These two actions plus inserting a transactional block of code for all of our function lead to the following sequential implementation for insert:

Implementing the most basic form of transactional memory was simple and straight forward, but increasing the performance was tricky. so that the transactions are smaller. It is implemented such that the transactions are light weight to ensure that when a transaction does fail it won't have to retry more code than it has to. The improvements made to the original implementation was removing transactions from contains and traverse since they do no write operations. Another change made was to decouple the traverse from the insert and remove. This small change made it, so the implementation doesn't have to retraverse the tree. Some checks have been

```
void insert(int const &data){

  while (true) {

    __transaction_atomic{
      n = traverse(root, data)
      /* Do insert op */
      if (isAvl) rebalance(n);
    }
  }
}
```

(a) Initial Implementation. Entire function is encapsulated in atomic block.

```
void insert(int const &data){

  while (true) {
    n = traverse(root, data)
    __transaction_atomic{
      /* Check snapshots */
      /* Do insert op */
    }
    if (isAvl) rebalance(n);
  }
}
```

(b) Second Implementation. Traverse and Rebalance function are decoupled from write operations.

Fig. 2: Implementing the atomic_transaction block in the data structure

inserted as a result to ensure that the current node that is being worked on is still a valid option. Another modification made to increase performance was the decoupling of rebalance from insert and remove to increase performance of AVL tree operations. Before it would

IV. Performance Evaluation

Testing the implementation was done on Intel(R) Core(TM) i5-4460 CPU @ 3.2GHz with: Max Memory Bandwidth 25.6 GB/s, Number of Cores 4, Number of Threads 4, 6 MB SmartCache, 24.0 GB RAM, 64 - bit Windows OS, Run through Ubuntu terminal shell for Windows. It should be noted that the performance is expected to significantly decrease across all platforms when the number of threads increase past four because of hardware limitations. The concurrent data structure is expected to do significantly better than the software and improved software data structure because of how transactional memory operates. The number of benchmarks was three and those three were:

- 9% Insert, 1% Remove, 90% contains
- 20% Insert, 10% Remove, 70% contains
- 50% Insert, 50% Remove, 0% contains

This was carried out by providing 5,000,000 operations for the threads to perform, overall. Each tree was populated to its expected capacity. That is for test 1, the tree would be at 90% expected capacity or 450,000 nodes, 2/3 capacity, and 50% capacity, respectively for the last two.

A. Concurrent Data Structure

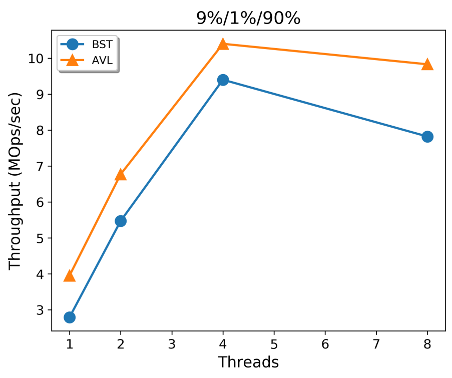
The results for the lock-based concurrent data structure can be seen in Figure **??**. The AVL tree outperforms BST in most scenarios due to the $O(n \lg n)$ traversals from rebalancing. Where the performance decreases is when rebalancing will get called more often and cause threads to lock nodes that other threads require. Another possible reason why the performance starts to equalize around 50% contains and removes is because of how costly it is to remove non-leaf nodes. Any leaf node will require at most two locks, while removals of non-leaf nodes could require up to nine. In BST's, the removal and insertion have a much higher chance of creating leaf nodes. The number of leaf nodes in AVL tree is significantly lower due to rebalancing. In fact the upper bound for a balanced BST is $O(n/2)$ leaf nodes. The performance for 50% inserts and removes demonstrates that with a rebalancing operation performance is not particuarly enhanced. With a 90% contains ratio, the AVL tree can handle up to 10 MOps/sec.

B. Software Transactional Memory Data Structure

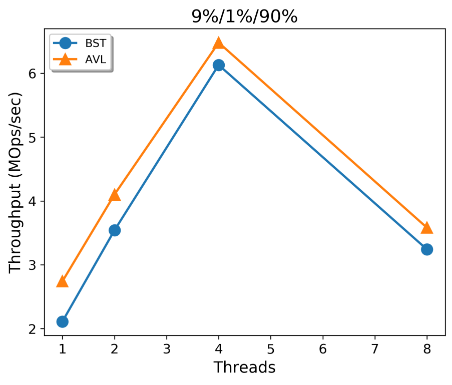
Our software transactional memory data structure performance is shown by Figure **??**. The performance of AVL is higher than BST in all, but the last scenario of 50% removes and inserts. This is due to the costly nature of rebalancing in AVL, which allowed BST to be on equal terms when using 1 or 2 threads. The performance in all three graphs are significantly lowered when moving from 4 to 8 threads. The reason being for this is that the hardware has limited how well the threads can process these atomic transaction. This hardware limitation increases the chance of a transaction not completing because it has to stall for another thread to start its own transaction. This leads into a massive increase in failed atomic transactions and is the reason for this decline in the graph. A reason for AVL outperforming BST is due to the fast read operations since the graph is more balanced since the scenarios where it does outperform BST is where there are a significant amount of read calls.

C. Improved Software Transactional Memory Data Structure

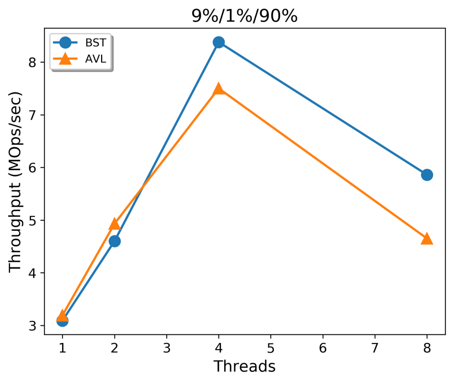
In the results shown for our improved software transactional memory in Figure **??**, BST has overtaken AVL in most scenarios, where the only exception being 1 or 2 threads being used in Figure **??** (c). The reason for this is because of traverse decoupling and how insert and remove transaction do not restart completely or don't restart while traversing. AVL and BST perform worse in both STM implementations because of how changing snapshots will cause them to restart since snapshots are updated in both insert and remove. This is one reason why the performance for 8 threads for BST in Figure **??** (c) can do about 3.5 million more operations per second than in (f).



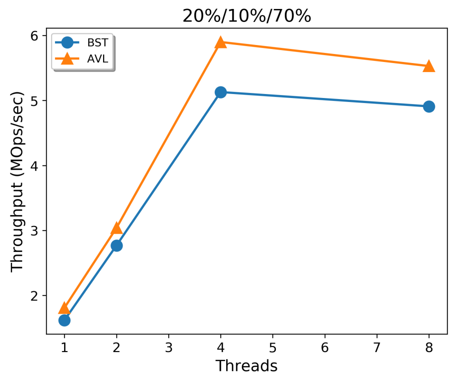
(a) Locking



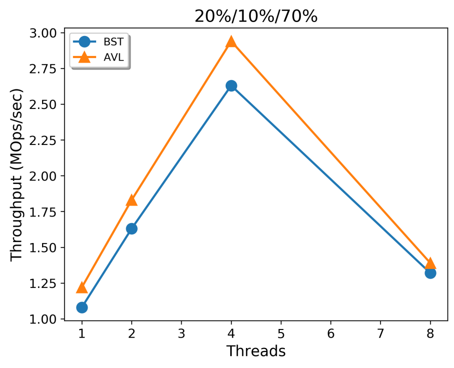
(b) STM



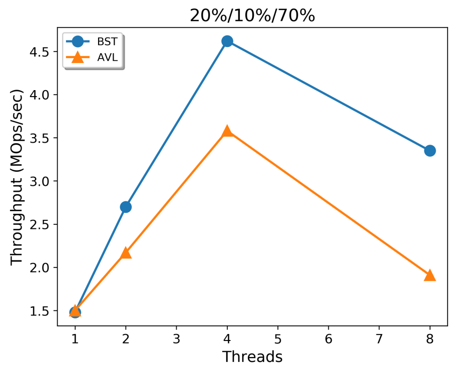
(c) Modified STM



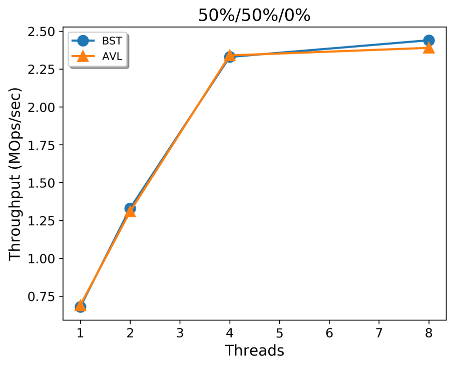
(d) Locking



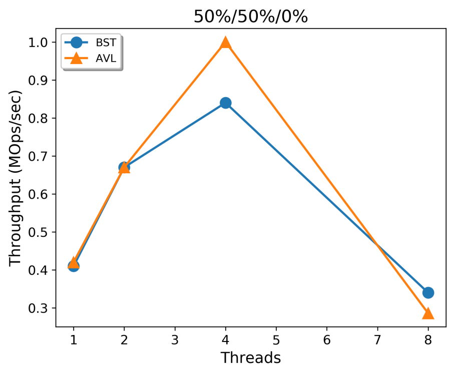
(e) STM



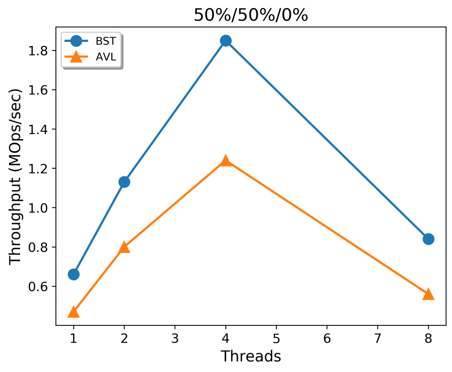
(f) Modified STM



(g) Locking



(h) STM



(i) Modified STM

Fig. 3: The performance results of all tests with each of the three data structure implementations.

V. Discussion

A. Analysis Summary

The locking PaVT BST and AVL trees should to be highly scalable for the architecture that they were run on. Running the data structures on a larger number of cores could depict the true limitation for the data structure. The performance limitations became very obvious in the STM data structures. The performance would result in a sequential if not worse performance. However, the STM framework provided a relaxed programming effort for lock management as opposed to the locking requirements of the original implementation. The PaVT condition proved to provide a large amount of concurrency as seen in the $9\%/1\%/90$ tests. Due to the lock-free nature of the **contains** method, only a small atomic transaction for reading the snapshots was necessary which allowed for a high concurrent environment before improving the STM data structure.

B. Development Issues Encountered

Providing synchronization in updating the snapshots proved to be a challenging aspect due to the few details provided by the author's. Their previous work with the logical ordering provided an initial starting point on how to maintain the snapshots. After contacting the authors, the algorithm became very simple to implement. The following difficulty that arose from synchronization came from the lock management of the removal method. There were two defining challenges with the locking protocol in **remove()**: (1) Managing locks that have already have been acquired. Although previously mentions in the details of **remove()**, there are some instances where two locks are needed that pertain to the same node (i.e., locking the successor's parent p_s and n's right child, r, which are the same node). The second difficulty came from preserving locking order upon releasing locks. This was addressed by providing Boolean to indicate if the lock needed to be locked (from a case of the first challange). This created code bloat which was solved by the STM implementations. The locking order became a problem during the rebalancing operations as the locking order changed with rotations. The design of the **rebalance** function was designed to address this problem to easily determine which locks would need to be unlocked first.

References

[1] D. Drachsler-Cohen, M. Vechev, and E. Yahav. 2018. Practical concurrent traversals in search trees. In Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18). ACM, New York, NY, USA, 207-218. DOI: https://doi.org/10.1145/3178487.3178503

[2] V. Luchangco, J. Maurer, M. Moir, H. Boehm, J. Gottschlich, M. Michael, T. Riegel, M. Scott, T. Shpeisman, M. Spear, M. Wong, "Transactional Memory Support for C++",[online document], 2013. Available: JtC1/SC22 subcommittee, http://www.open-std.org [Accessed: November 29, 2018]

[3] Bronson, N. G., Casper, J., Chafi, H., Olukotu, K. (2014). A practical concurrent binary search tree. In PPoPP'14.

[4] Crain, T., Gramoli, V., Raynal, MA Contention-Friendly Binary Search Tree. In Euro-Par '13.

[5] Crain, T., Gramoli, V., Raynal, M. (2011) A transaction-friendly binary search tree. [Research Report] PI-1984, pp.21. ¡inria-00618995v1¿

[6] Draschler, D., Vechev, M., Yahav, E. (2014) Practical Concurrent Binary Search Trees via Logical Ordering. In PPoPP'14.

Algorithm 3 Remove (key)

```
$n \leftarrow$ **traverse**$(root, key)$
**if** $n.k \neq key$ **then** $unlock(n)$; **return**
$p \leftarrow n.P$; $l \leftarrow n.L$; $r \leftarrow n.R$
**if** $!tryLock(p)$ **then** $unlockAll()$ **then** restart
**if** $n$ is a leaf **then**
    $S_{\max} \leftarrow n.S_R$; $S_{\min} \leftarrow n.S_L$
    $n.mark \leftarrow true$
    **if** $p.k > n.k$ **then**
        $p.L \leftarrow null$; $p.S_L \leftarrow S_{\min}$; $S_{\min}.S_R \leftarrow p$
    **else**
        $p.R \leftarrow null$; $p.S_R \leftarrow S_{\max}$; $S_{\max}.S_L \leftarrow p$
    $unlockALl()$
    **if** $AVL$ **then** **rebalance**$(p)$; **return**
**if** $n$ has one child **then**
    $\alpha \leftarrow l = null$; $c \leftarrow (\alpha\ ?\ r : l)$
    $S_{\max} \leftarrow n.S_R$; $S_{\min} \leftarrow n.S_L$
    $snap \leftarrow (\alpha\ ?\ S_{\max} : S_{\min})$
    $lock(c, snap)$
    **if** $(\alpha\ \&\&\ snap.S_L \neq n)\ ||\ (!\alpha\ \&\&\ snap.S_R \neq n)$ **then**
        $unlockAll()$ **then** restart
    $n.mark \leftarrow true$
    $(p.L = n\ ?\ p.L : p.R) \leftarrow c$
    $S_{\min}.S_R \leftarrow S_{\max}$; $S_{\max}.S_L \leftarrow S_{\min}$
    $unlockALl()$
    **if** $AVL$ **then** **rebalance**$(p)$
    **return**
$l \leftarrow n.L$; $r \leftarrow n.R$; $S_{\min} \leftarrow n.S_L$; $S_{\max} \leftarrow n.S_R$
$lock(l, r, S_{\min}, S_{\max})$
**if** $S_{\min}.S_R \neq l\ ||\ S_{\min}.mark$ **then**
    $unlockAll()$ **then** restart
**if** $r.L = null$ **then**
    $n.mark \leftarrow true$
    $r.L \leftarrow l$; $l.P \leftarrow r$; $r.P \leftarrow p$
    $(p.L = n\ ?\ p.L : p.R) \leftarrow r$
    $S_{\min}.S_R \leftarrow S_{\max}$; $S_{\max}.S_L \leftarrow S_{\min}$
    $unlockALl()$
    **if** $AVL$ **then** **rebalance**$(r)$
    **return**
$s \leftarrow S_{\max}$; $p_s \leftarrow s.P$; $r_s \leftarrow s.R$; $(S_R)_s \leftarrow s.S_R$
$lock(s, p_s, r_s, (S_R)_s)$
**if** $s.S_L \neq n\ ||\ s.mark || (S_R)_s.S_L \neq s || (S_R)_s.mark$ **then**
    $unlockAll()$ **then** restart
$n.mark \leftarrow true$
$s.R \leftarrow r$; $r.P \leftarrow s$; $s.L \leftarrow l$; $l.P \leftarrow s$
$(p.L = n\ ?\ p.L : p.R) \leftarrow s$
$p_s.L \leftarrow r_s$
**if** $r_s \neq null$ **then** $r_s \leftarrow r_s$
$S_{\min}.S_R \leftarrow S_{\max}$; $S_{\max}.S_L \leftarrow S_{\min}$
$unlockALl()$
**if** $AVL$ **then**
    **rebalance**$(s)$; **rebalance**$(p_s)$
```