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 a reimplementation of a practical and concurrent binary search tree and an AVL tree that uses the PaVT condition, which permits a lock-free contains operation. We utilize this condition by the use of snapshots, which contains a logical ordering of all items in the tree. We present a concurrent BST and AVL tree that utilizes this condtion. We implemented a GCC software transactional memory version of both the concurrent BST and concurrent AVL tree. This is evaluated against our own concurrent implementations to test the effectiveness and performance of the use of transactional memory as well as the performance of our concurrent operations. Our results show that transactional memory has a lower performance than the concurrent data structure.**

I. Introduction

Over the years the number of cores on the modern processor has increased. Designing data structures for concurrency is the foundation to utilize these cores; otherwise, they become a bottleneck of contention [1]. Search trees preserve an ordering to allow for quick lookups. 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 [2].

Draschler-Cohen et al. of [2] provide a necessary and sufficient condition for path validation in traversals which they call *PaVT* condition. This condition applies to *any* search tree and can validate whether a key is present in the tree or not which can allow for a lock-free membership test. They demonstrate that using the PaVT condition with a Binary Search Tree (BST) and an AVL tree, their implementation outperforms existing solutions using a lock-free contains on search-based data structures.

However, evaluating this condition leveraging software transactional memory (STM) 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 high contention situations exists still requires some research [3]. Transactional memory reduces the difficulties in implementing lock-based data structures, such as deadlock, but balancing the usefulness of the simple resource sharing and performance is a hard task [4].

This paper wishes to show the reimplementation of the author's BSTs (including AVL) which leverages the PaVT condition. Further extending their work, we implement the BSTs using GCC's software transactional memory framework and compare the performance of the STM implementation to the lock-based 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 present in any search tree based on the snapshot of a path to that key. 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 fields, $F = \{L, R\}$. 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 definition 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 between 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 occurred and the traversal would need to restart.

B. Succinct Path Snapshot

Maintaining 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* (SPS) 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, $(<, L), (>, R)$. The author's formal definition is stated in II.1.

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 S_R 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 as the upper and lower bound of the logical ordering of the keys.

C. Correctness

The following section will discuss the overall implementation of the tree. For SPS to be implmented, nodes must have a mark to logical state whether they are in the tree. Also, each node maintains a lock and any operation acquiring locks must acquire all before the operation begins. Correctness of maintaining snapshots follows from the fact that if an update to a path is occurring, 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 insert and remove 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; therefore, 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 rebalance function so it reflects the relaxed behavior stated by the authors. 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, particularly, during removal from their source code. Thus our source code is an amalgam of their pseudo and source code.

Notable changes we made were to only use the traverse algorithm on the node to be inserted and removed. The author's provide in the pseudocode to use traverse to reach the successor for the node being removed. Since snapshots are stored as pointers in nodes, we simply get access to the successor via the pointer. This prevents performing a second optimistic lock on the successor's parent as mentioned in the following sections. In the following subsections, we will discuss the design of our implementation, including the main algorithms.

A. Tree Structure

Our tree structure implements the BST 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 1. 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. If a node is logically in the tree, it is reachable. The keys are immutable, so nodes cannot be recycled. This prevents nodes from needing to be rechecked.

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.

```
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

.

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 4,000,000 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.

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 a 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 released 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 only if the node is not marked otherwise 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. If the key is not in the node's snapshot, the procedure is restarted. If key is in the node's snapshot for **contains**, the function simply returns

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: $\quad n \leftarrow next$
6: $\quad f \leftarrow \mathbf{nextField}(curr, key)$
7: $\quad$ **if** $n.k = key$ **then**
8: $\quad\quad$ **if** $n.mark = true$ **then**
9: $\quad\quad\quad$ restart
10: $\quad\quad$ **return** true
11: $\quad next \leftarrow n.field$
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: $\quad$ restart
16: **return** false

false. For traverse, once snapshot/mark conditions are met, the locked node is returned.

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 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.

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. If a thread were to traverse to it, the snapshot would be incorrect if they have not updated, hence the traversal would restart.

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

Algorithm 2 Insert (key)

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

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 22. 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 43. 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.

To reduce the amount of code in Algorithm 3, **rebalance** was omitted. It is called immediately after the $unlockAll()$ after a when the marked node is physically removed. The node to rebalance are the nodes where the end of the path has changed. For case 1) and 2), the parent p, for case 3), the child c, and for case 4), the successor s and successor's parent p_s.

The linearization point occurs once the node is marked. At this point, any traversal will find the node marked. Because locks are acquired before any insertion or other deletion can take place, the waiting thread will observed the marked node and will restart its operation. A contains linearizes from the linearization of the snapshots as mentioned earlier.

Algorithm 3 Remove (key)

1: $n \leftarrow \mathbf{traverse}(root, key)$
2: **if** $n.k \neq key$ **then** $unlock(n)$; **return**
3: $p \leftarrow n.P$; $l \leftarrow n.L$; $r \leftarrow n.R$
4: **if** $!tryLock(p)$ **then** $unlockAll()$ **then** restart
5: **if** n is a leaf **then**
6: $S_{\max} \leftarrow n.S_R$; $S_{\min} \leftarrow n.S_L$
7: $n.mark \leftarrow true$
8: **if** $p.k > n.k$ **then**
9: $p.L \leftarrow null$; $p.S_L \leftarrow S_{\min}$; $S_{\min}.S_R \leftarrow p$
10: **else**
11: $p.R \leftarrow null$; $p.S_R \leftarrow S_{\max}$; $S_{\max}.S_L \leftarrow p$
12: $unlockALl()$; **return** ;
13: **if** n has one child **then**
14: $\alpha \leftarrow l = null$; $c \leftarrow (\alpha \;?\; r : l)$
15: $S_{\max} \leftarrow n.S_R$; $S_{\min} \leftarrow n.S_L$
16: $snap \leftarrow (\alpha \;?\; S_{\max} : S_{\min})$
17: $lock(c, snap)$
18: **if** $(\alpha \,\&\&\, snap.S_L \neq n) \,||\, (!\alpha \,\&\&\, snap.S_R \neq n)$ **then**
19: $unlockAll()$ **then** restart
20: $n.mark \leftarrow true$
21: $(p.L = n \;?\; p.L : p.R) \leftarrow c$
22: $S_{\min}.S_R \leftarrow S_{\max}$; $S_{\max}.S_L \leftarrow S_{\min}$
23: $unlockALl()$; **return**
24: $l \leftarrow n.L$; $r \leftarrow n.R$; $S_{\min} \leftarrow n.S_L$; $S_{\max} \leftarrow n.S_R$
25: $lock(l, r, S_{\min}, S_{\max})$
26: **if** $S_{\min}.S_R \neq l \,||\, S_{\min}.mark$ **then**
27: $unlockAll()$ **then** restart
28: **if** $r.L = null$ **then**
29: $n.mark \leftarrow true$
30: $r.L \leftarrow l$; $l.P \leftarrow r$; $r.P \leftarrow p$
31: $(p.L = n \;?\; p.L : p.R) \leftarrow r$
32: $S_{\min}.S_R \leftarrow S_{\max}$; $S_{\max}.S_L \leftarrow S_{\min}$
33: $unlockALl()$; **return**
34: $s \leftarrow S_{\max}$; $p_s \leftarrow s.P$; $r_s \leftarrow s.R$; $(S_R)_s \leftarrow s.S_R$
35: $lock(s, p_s, r_s, (S_R)_s)$
36: **if** $s.S_L \neq n \,||\, s.mark||(S_R)_s.S_L \neq s||(S_R)_s.mark$ **then**
37: $unlockAll()$ **then** restart
38: $n.mark \leftarrow true$
39: $s.R \leftarrow r$; $r.P \leftarrow s$; $s.L \leftarrow l$; $l.P \leftarrow s$
40: $(p.L = n \;?\; p.L : p.R) \leftarrow s$
41: $p_s.L \leftarrow r_s$
42: **if** $r_s \neq null$ **then** $r_s.P \leftarrow p_s$
43: $S_{\min}.S_R \leftarrow S_{\max}$; $S_{\max}.S_L \leftarrow S_{\min}$
44: $unlockALl()$

F. Rebalance

The **rebalance** operation is implemented based on the author's description [2]. 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 preserve 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 [4]:

- 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 [4].

2) Implementation Breakdown: GCC TM uses `__atomic_transaction` blocks to wrap the code that must be executed atomically. As stated before, in order to convert the data structure to implement STM, the node class removes the atomic wrappers from the snapshots as well as the mutex as they are not transaction safe. The node class is an external class to the BST data structure which required all fields and methods to be marked `transaction_safe`.

The initial implementation required wrapping the entire **insert** and **remove** functions in the atomic block as seen in Figure 2(a). Due to the PaVT condition, the contains method only required reading the snapshots atomically, so line 14 would be wrapped in the atomic block. All helper function would be enclosed in the atomic blocks.

The Improved STM implementation required two important changes to the code. The first would be decouple the **traverse** algorithm from **insert** and **remove**. The second would be to decouple **rebalance** from the aforementioned functions as well. This can be seen in Figure 2(b). The former required checking the snapshots after the node has been locked and returned. This resulted in a synchronization-free traversal. The former required more organization in the removal section. Pointers to the nodes to be rebalanced were initialized at the start of the function. Depending on the removal the pointers were set to those values and if they pointers were not $null$, they would be rebalanced. In order to synchronize **rebalance**, the traversal from the node to the root would need to be decoupled from the atomic reads and write performed on the height and the structural layout of the tree.

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, 4 cores and 4 threads along with a 6 MB SmartCache. 24.0 GB RAM was available, and the evaluation was run through Ubuntu 18.04 shell VM on a 64 - bit Windows OS. It should be noted that the performance was expected to decrease past the 4 thread count due to hardware limitations. Improving the available threads, and extend to hyper-threading could increase the overall performance of the data structure.

The benchmarks used to test follow the standard empirical evaluation which are:

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

This was carried out by providing 2,000,000 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 3. The AVL tree outperforms BST in most scenarios due to the $O(n \lg n)$ traversals from rebalancing. It can be seen that as the number of remove operations

```
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

increases, 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 particularly 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 3. The performance of AVL is higher than BST in all scenarios with the exception of the 50% removes and inserts scenario. 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 a single processor starting two different transactions will cause one of them to stall. Furthermore, if both transactions have any conflicts and one finishes then it will cause the other to fail and restart. 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 in most scenarios is due to the fast traverse and contains operations since the graph is more balanced as shown by (b) and (e) in Figure 3. Traverse and contains are both faster in AVL trees because having fewer leaf nodes and a more compact tree, which is due to rebalancing, means threads won't have to go far to find their target.

C. Improved Software Transactional Memory Data Structure

In the results shown for our improved software transactional memory in Figure 3, BST has overtaken AVL in most scenarios, where the only exception being 1 or 2 threads being used in Figure 3 (c). This may be due to decoupling traverse from the atomic transaction block. This allowed for insert and remove transactions to not restart completely or don't restart while traversing. AVL and BST performs worse in both STM and Modified STM implementations because changing snapshots can cause transactions to restart since snapshots are updated in both insert and remove. This makes Decoupling traverse is one reason why the performance for 8 threads for BST in Figure 3 (c) can do about 3.5 million more operations per second than in (f).

Decoupling rebalance was another improvement, but its impact is far less when compared to the traverse improvement. This is exemplified by how much of an improvement BST gains, so AVL should gain a similar improvement from this change, however there isn't as noticeable of a change in AVL even though two improvements have been made. Part of this reason is that the **rebalance** function is a bottleneck of contention [3]. The relaxed rebalancing performed in [3] used only right rotations which indicates that our implementation may be sub-optimal for STM implementations.

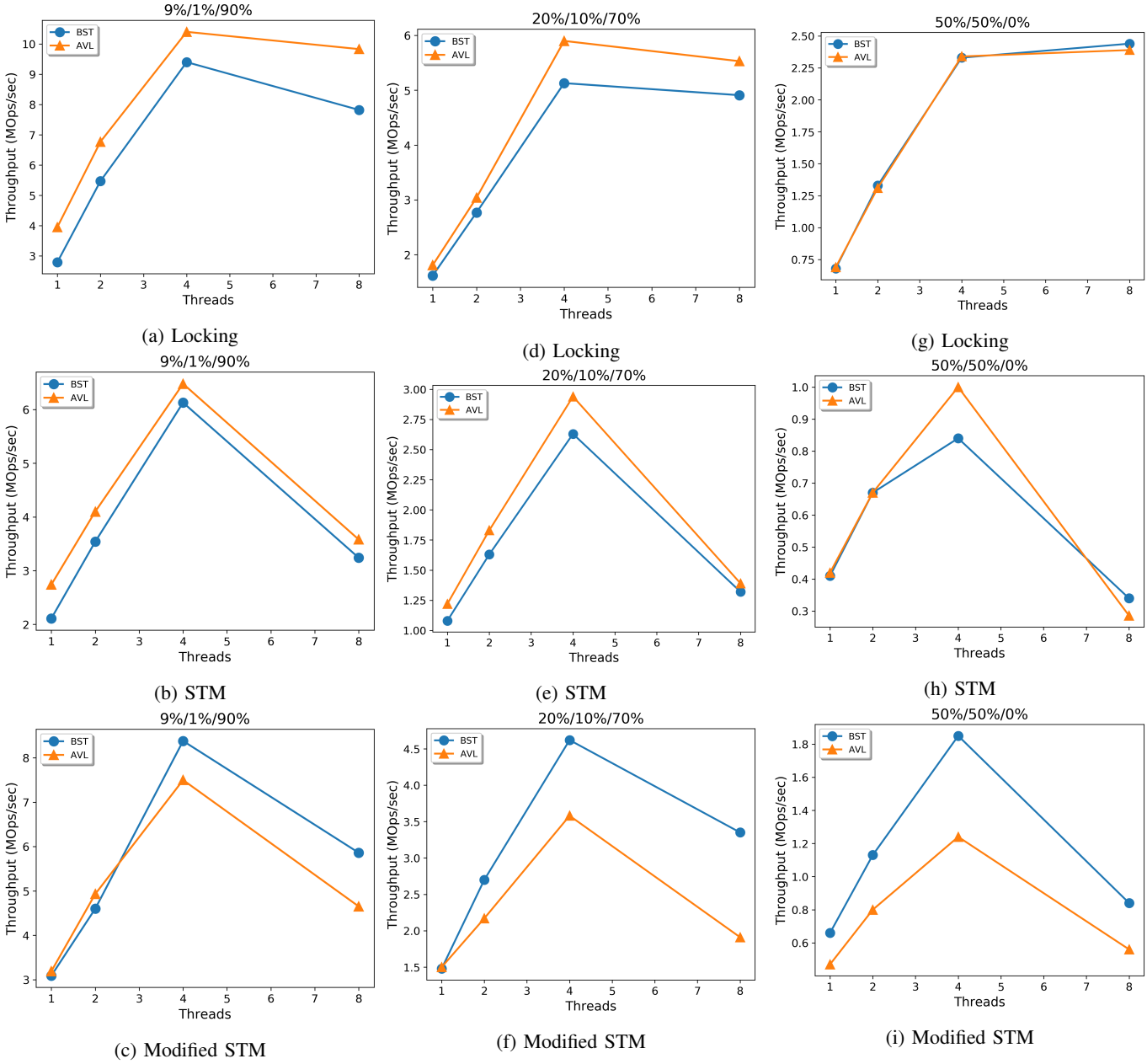
V. Discussion

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



(a) Locking (d) Locking (g) Locking

(b) STM (e) STM (h) STM

(c) Modified STM (f) Modified STM (i) Modified STM

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

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

[2] 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

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

[4] 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]

[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.