1 Introduction

With the increase in the number of cores on the processor, ocncurrent...
stuff about concurrency issues
lock free lookups
Author's contribution
Importance of STM
Our reimplimentation
Quick outline of paper

2 Background

2.1 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 2.1.

Theorem 2.1. *(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.

2.2 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 succint 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 formal defintion that author's provided is stated below.

Definition 2.1. (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 [4]. 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.

2.3 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.

2.4 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.

3 Data Structure Design Principles

3.1 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

```
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;
}
```

Figure 1: The Node data structure implemented within the BinarySearchTree class

.

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 3.1. 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. 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.

3.2 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.

3.3 Traverse/Contains

Algorithm **??** 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 $f = \textbf{nextField}(n, key)$ then $n.f = null$. If the node with key is found then the node is locked. 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.

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 15. 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 traveresed to the current node which indicates the procedure to restart.

3.4 Contains

The contains method

Algorithm 1 Contains (key)

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

3.5 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.

Algorithm 2 Insert (key)

```
 1: $n \leftarrow \textbf{traverse}(root, key)$
 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: $newNode \leftarrow \textbf{new}Node(key)$
 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 rebalance$(n)$
```

3.6 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 9. If n has one child, then c takes n's place. Now n's snapshots will be updated to contain each other at line 26. 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 $(S_R)_s$

Algorithm 3 Remove (key)

```
 1: $n \leftarrow \textbf{traverse}(root, key)$
 2: if $n.k \neq key$ then $unlock(n)$; return
 3: $p \leftarrow n.P$
 4: if $!tryLock(p)$ then $unlockAll()$ then restart
```

```
$l \leftarrow n.L$; $r \leftarrow n.R$
**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**
    $\alpha \leftarrow l = null$
    $c \leftarrow (\alpha\ ?\ r\ :\ l)$
    $lock(c)$
    $S_{\max} \leftarrow n.S_R$; $S_{\min} \leftarrow n.S_L$
    $snap \leftarrow (\alpha\ ?\ S_{\max}\ :\ S_{\min})$
    $lock(snap)$
    **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** $r_s \leftarrow r_s$
$S_{\min}.S_R \leftarrow S_{\max}$; $S_{\max}.S_L \leftarrow S_{\min}$
```

```
50: $unlockALl()$
51: if $AVL$ then
52:     rebalance($s$); rebalance($p_s$)
```

3.7 Rebalance

Algorithm 4 Rebalance (node)

```
 1: $p \leftarrow n.P$
 2: while $n \neq root$ do
 3:     $lock(p)$
 4:     if $n.P \neq p$ then
 5:         $unlock(p)$
 6:         if $n.mark = true$ then return
 7:     $lock(n)$
 8:     if $n.mark = true$ then
 9:         $unlockAll()$ return
10:     $h' \leftarrow$ height($n$)
11:     $bf \leftarrow$ height($n.L$) $-$ height($n.R$)
12:     if $h' \neq n.h$ then
13:         $n.h \leftarrow h'$
14:     else if $bf \leq 1$ then
15:         $unlockAll()$ then return
16:     if $bf > 1$ then
17:         $c \leftarrow n.R$; $g_c \leftarrow c.L$
18:         $lock(c)$
19:         $bf_c \leftarrow$ height($c.L$) $-$ height($c.R$)
20:         if $bf_c < 0$ then
21:             $lock(g_c)$
22:             rotateRight($g_c, c, n$)
23:             rotateLeft($g_c, n, p$)
24:             $unlockAll()$
25:             $n \leftarrow g_c$
26:         else
27:             rotateLeft($c, n, p$)
28:             $unlockAll()$
29:             $n \leftarrow c$
30:     else if $bf < -1$ then
31:         $c \leftarrow n.L$; $g_c \leftarrow c.R$
32:         $lock(c)$
33:         $bf_c \leftarrow$ height($c.L$) $-$ height($c.R$)
34:         if $bf_c > 0$ then
35:             $lock(g_c)$
36:             rotateLeft($g_c, c, n$)
37:             rotateRight($g_c, n, p$)
38:             $unlockAll()$
39:             $n \leftarrow g_c$
40:         else
```

```
            **rotateRight**$(c, n, p)$
            $unlockAll()$
            $n \leftarrow c$
    **else**
        $unlockAll()$
        $n \leftarrow p$
        $p \leftarrow n.P$
```

3.8 Software Transactional Memory Implementation

4 Performance Evaluation

5 Discussion

References

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

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

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

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