2013 IEEE 33rd International Conference on Distributed Computing Systems

Non-blocking Patricia Tries with Replace Operations

Niloufar Shafiei
Department of Computer Science and Engineering
York University
Toronto, Canada
niloo@cse.yorku.ca

***Abstract*—This paper presents a non-blocking Patricia trie implementation for an asynchronous shared-memory system using Compare&Swap. The trie implements a linearizable set and supports three update operations: insert adds an element, delete removes an element and replace replaces one element by another. The replace operation is interesting because it changes two different locations of trie atomically. If all update operations modify different parts of the trie, they run completely concurrently. The implementation also supports a wait-free find operation, which only reads shared memory and never changes the data structure. Empirically, we compare our algorithms to some existing set implementations.**

***Keywords*-Patricia trie; non-blocking; shared memory; lock-free; concurrent data structure; dictionary; set;**

I. Introduction

A Patricia trie [1] is a tree that stores a set of keys, which are represented as strings. The trie is structured so that the path from the root to a key is determined by the sequence of characters in the key. So, the length of this path is at most the length of the key (and will often be shorter). Thus, if key strings are short, the height of the trie remains small without requiring any complicated balancing. The simplicity of the data structure makes it a good candidate for concurrent implementations. Patricia tries are widely used in practice. They have applications in routing systems [2], data mining [3], machine learning [4], bioinformatics [5], etc. Allowing concurrent access is essential in some applications and can boost efficiency in multicore systems.

We present a new concurrent implementation of Patricia tries that store binary strings using single-word *Compare&Swap* (CAS). The operations on the trie are linearizable, meaning they appear to take place atomically [6]. They are also non-blocking (lock-free): *some* process completes its operation in a finite number of steps even if other processes fail. *Wait-free* algorithms satisfy the stronger guarantee that *every* process completes its operation in a finite number of steps.

Our implementation provides wait-free find operations and non-blocking insertions and deletions. We also provide a non-blocking replace operation that makes two changes to the trie atomically: it deletes one key and inserts another. If all pending updates are at disjoint parts of the trie, they do not interfere with one another.

A Patricia trie can be used to store a set of points in $\mathbb{R}^d$. For example, a point in $\mathbb{R}^2$ whose coordinates are (x, y) can be represented by a key formed by interleaving the bits of x and y. (This yields a data structure very similar to a quadtree.) Then, the replace operation can be used to move a point from one location to another atomically. This operation has applications in Geographic Information System [7]. The replace operation would also be useful if the Patricia trie were adapted to implement a priority queue, so that one could change the priority of an element in the queue.

Search trees are another class of data structures that are commonly used to represent sets. When keys are not uniformly distributed, balanced search trees generally outperform unbalanced ones. The reverse is often true when keys are uniformly distributed due to the simplicity of unbalanced search trees. Our empirical results show that the performance of our trie is consistently good in both scenarios. This is because our trie implementation is as simple as an unbalanced search tree but also keeps trees short. For simplicity, we rely on a garbage collector (such as the one provided in Java implementations) that deallocates objects when they are no longer accessible.

For our Patricia trie algorithms, we extend the scheme used in [8] for binary search trees to coordinate processes. Thus, we show that the scheme is more widely applicable. In particular, we extend the scheme so that it can handle update operations that make more than one change to the tree structure. When a process p performs an update, it first creates a descriptor object that contains enough information about the update, so that other processes can help complete the update by reading the descriptor object. As in [8], before p changes the tree, it flags a small number of nodes to avoid interference with other concurrent updates. A node is flagged by setting a pointer in the node to point to the update's descriptor object using a CAS. The CAS fails if the node is already flagged by another update; in this case, p helps the other update before retrying its own update. (This ensures the non-blocking property.) When an update is complete, nodes that are still in the tree are unflagged by removing the pointers to the update's descriptor object. Searches do not need to check for flags and can therefore traverse the tree very efficiently simply by reading child pointers. Searches in our Patricia trie are wait-free, unlike the searches in [8],

1063-6927/13 $26.00 © 2013 IEEE
DOI 10.1109/ICDCS.2013.43

because the length of a search path in a Patricia trie is bounded by the length of the key.

There are several novel features of this work. In our implementation, we design one fairly simple routine that is called to perform the real work of all update operations. In contrast, insert and delete operations in [8] are handled by totally separate routines. This makes our proof of correctness more modular than the proof of [8]. Our techniques and correctness proof can be generalized to other tree-based data structures.

In [8], modifications were only made at the bottom of the search tree. Our new Patricia trie implementation also copes with modifications that can occur anywhere in the trie. This requires proving that changes in the middle of the trie do not cause concurrent search operations passing through the modified nodes to go down the wrong branch. Howley and Jones [9] introduced changes in the middle of a search tree but only to keys stored in internal nodes, not the structure of the tree itself.

In [8], atomic changes had to be done by changing a single pointer. Our replace operation makes two changes to the trie atomically. Both changes become visible at the first CAS operation on a child pointer. This new scheme can be generalized to make several changes to the trie atomically by making all changes visible at a single linearization point. Cederman and Tsigas [10] proposed a non-blocking replace operation for a tree-based data structure, but they require double-CAS, which modifies two non-adjacent locations.

II. Related Work

Most concurrent data structures are lock-based. However, lock-based implementations have drawbacks such as priority inversion, deadlock and convoying. Two state of the art examples of lock-based implementations of set data structures are the AVL tree by Bronson et al. [11], which maintains an approximately balanced tree, and the self-adjusting binary search tree by Afek et al. [12], which moves frequently accessed nodes closer to the root. Aref and Ilyas [13] described how lock-based implementations could be designed for a class of space-partitioning trees that includes Patricia tries. Lock-coupling can also be applied to implement a concurrent Patricia trie [14].

In this paper, we focus on non-blocking algorithms, which do not use locks. There are two general techniques for obtaining non-blocking data structures: universal constructions (see the related work section of [15] for a recent survey of work on this) and transactional memory [16] (see [17] for a survey). Such general techniques are usually not as efficient as algorithms that are designed for specific data structures.

Tsay and Li [18] gave a general wait-free construction for tree-based data structures. To access a node, a process makes a local copy of the path from the root to the node, performs computations on the local copy, and then atomically replaces the entire path by its local copy. Since this approach copies many nodes and causes high contention at the root, their approach is not very efficient. Barnes [19] presented another general technique to obtain non-blocking implementations of data structures in which processes cooperate to complete operations.

Ellen et al. [8] presented a non-blocking binary search tree data structure from CAS operations. Their approach has some similarity to the cooperative technique of [19]. As discussed in Section I, our Patricia trie implementation extends the approach used in [8]. Brown and Helga [20] generalized the binary search trees of [8] to non-blocking k-ary search trees and compared the non-blocking search trees with the lock-based search tree of Bronson et al. [11] empirically.

Howley and Jones [9] presented a non-blocking search tree from CAS operations using a cooperative technique similar to [8]. Their tree stores keys in both leaf and internal nodes. However, search operations sometimes help update operations by performing CASs. Braginsky and Petrank [21] proposed a non-blocking balanced B+tree from CAS operations.

Recently, Prokopec et al. [22] described a non-blocking hash trie that uses CAS operations. Their approach is very different from our implementation. Unlike Patricia tries, in their trie implementation, an internal node might have only one child. In their implementation, nodes have up to 2^k children (where k is a parameter) and extra intermediate nodes are inserted between the actual nodes of the trie. With $k = 5$, the height of their trie is very small, making their implementation very fast when contention is low. However, our experiments suggest that it is not very scalable under high contention. Unlike our implementation, their search operation may perform CAS steps.

Non-blocking implementations of set data structures have also been proposed based on skip lists using CAS operations [23], [24], [25]. A non-blocking skip list (ConcurrentSkipListMap) has been implemented in the Java class library by Doug Lea.

III. Algorithm Description

We assume an asynchronous shared-memory system with single-word CAS operations. We first give the sequential specification of the operations. The trie stores a set D of keys from a finite universe U. If $v \notin D$, insert(v) changes D to $D \cup \{v\}$ and returns true; otherwise, it returns false. If $v \in D$, delete(v) changes D to $D - \{v\}$ and returns true; otherwise, it returns false. If $v \in D$ and $v' \notin D$, replace(v, v') changes D to $D - \{v\} \cup \{v'\}$ and returns true; otherwise, it returns false. If $v \in D$, find(v) returns true; otherwise, it returns false. In either case, find(v) does not change D. We assume keys are encoded as ℓ-bit binary strings. (In Section VI, we describe how to handle unbounded length keys.)

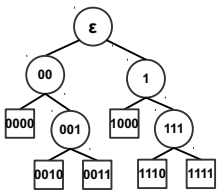

Figure 1. An example of a Patricia trie. Leaves are represented by squares and internal nodes are represented by circles.

A. Data Structures

First, we describe the structure of a binary Patricia trie. (See Figure 1.) Each internal node has exactly two children. The elements of D are stored as labels of the leaves of the trie. Each internal node has a label that is the longest common prefix of its children's labels. If a node's label has length $k-1$, then the kth bit of the node's left and right children's labels are 0 and 1, respectively. Since keys are ℓ-bit binary strings, the height of the trie is at most ℓ.

For simplicity, our trie initially contains just two leaf nodes with labels 0^ℓ and 1^ℓ and a root node whose label is the empty string. We assume the keys 0^ℓ and 1^ℓ cannot be elements of D. This ensures that the trie always has at least two leaf nodes and avoids special cases of update operations that would occur if the root were a leaf.

Next, we describe the objects that are used in the implementation (Figure 2). The Patricia trie is represented using Leaf and Internal objects which are subtypes of Node objects. Each Node object has a *label* field, which is never changed after initialization and stores the node's label. An Internal object has an array *child* of two Node objects that stores pointers to the children of the node.

Each Node object also has an *info* field that stores a pointer to an Info object that serves as the descriptor of the update operation that is in progress at the node (if any). An Info object contains enough information to allow other processes to help the update to complete. Info objects have two subtypes: Flag and Unflag. An Unflag object is used to indicate that no update is in progress at a node. Unflag objects are used instead of null pointers to avoid the ABA problem in the *info* field of a node. Initially, the *info* field of each Node object is an Unflag object. We say a node is *flagged* or *unflagged*, depending on whether its *info* field stores a Flag or Unflag object. The *info* and *child* fields of an internal node are changed using CAS steps. However, a leaf node gets flagged by writing a Flag object into its *info* field.

To perform an update operation, first some internal nodes get flagged, then some *child* fields are changed and then nodes that are still in the trie get unflagged. The nodes that must be flagged to perform an update operation are the internal nodes whose *child* field will be changed by the update or that will be removed from the trie by the update.

```
1)  Leaf: (subtype of Node)
2)    label: String
3)    info: Info                                  ▷ descriptor of update

4)  Internal: (subtype of Node)
5)    label: String
6)    child: Node[2]                              ▷ left and right child
7)    info: Info                                  ▷ descriptor of update

8)  Flag: (subtype of Info)
9)    flag: Internal[4]                           ▷ nodes to be flagged
10)   oldI: Info[4]                               ▷ expected values of CASs that flag
11)   unflag: Internal[2]                         ▷ nodes to be unflagged
12)   par: Internal[2]                           ▷ nodes whose children to be changed
13)   old: Node[2]                               ▷ expected children of nodes par
14)   new: Node[2]                               ▷ children of par to be changed to
15)   rmvLeaf: Leaf                              ▷ leaf to be flagged
16)   flagDone: Boolean                          ▷ set to true if flagging is successful

17) Unflag: (subtype of Info)                     ▷ has no fields
```

Figure 2. Data types used in the implementation

Flagging nodes is similar to locking nodes: it avoids having other operations change the part of the trie that would be changed by the update.

A Flag object has a number of fields. The *flag* field stores nodes to be flagged before the trie is changed and the *unflag* field stores nodes to be unflagged after the trie is changed. Before creating a Flag object, an update reads the *info* field of each node that will be affected by the update before reading that node's *child* field. This value of the *info* field is stored in the Flag's *oldI* field, and is used as the expected value by the CAS that flags the node. This ensures that if the node is successfully flagged, it has not changed since its children were read. The boolean *flagDone* field is set to true when the flagging for the update has been completed. In the case of a replace operation, the *rmvLeaf* field points to the leaf to be removed by the update after flagging is complete. The actual changes to the trie to be made are described in three more array fields of the Flag object: *par*, *old* and *new*. For each i, the update should CAS the appropriate *child* pointer of *par*$[i]$ from *old*$[i]$ to *new*$[i]$. As we shall see, once all nodes are successfully flagged, the CAS on each *child* pointer will be guaranteed to succeed because that pointer cannot have changed since the old value was read from it. Thus, like locks, the *info* field of a node is used to give an operation exclusive permission to change the *child* field of that node.

B. Update Operations

The implementation has three update operations: insert, delete and replace. All three have the same overall structure. The pseudo-code for our implementation is given in Figure 3 and 4. An update *op* uses the search routine to find the location(s) in the trie to be changed. It then creates a new Flag object I containing all the information required to complete the update by calling newFlag. If newFlag sees that some node that must be flagged is already flagged

with a different Flag I', it calls help(I') at line 108 to try completing the update described by I', and then op retries its update from scratch. Otherwise, op calls help(I) to try to complete its own update.

As mentioned earlier, flagging nodes ensures exclusive access for changing *child* pointers. Thus, an update operation flags the nodes whose *child* pointers it wishes to change and permanently flags any node that is removed from the trie to ensure update operations are not applied to a deleted portion of the trie.

Unlike locks, the Info objects store enough information, so that if a process performing an operation crashes while nodes are flagged for it, other processes can attempt to complete the operation and remove the flags. This ensures that a failed operation cannot prevent others from progressing. To ensure the non-blocking property, if an update must flag more than one internal node, we order the internal nodes by their *label*s.

The help(I) routine carries out the real work of an update operation using the information stored in the Flag object I. It first uses *flag CAS* steps (line 88) to flag some nodes by setting their *info* fields to I. If all nodes are flagged successfully, help(I) uses *child CAS* steps (line 96) to change the *child* fields of the internal nodes in $I.par$ to perform the update. Then, it uses *unflag CAS* steps (line 99) to unflag nodes that were flagged earlier, except those that were removed from the trie, by setting their *info* fields to new Unflag objects. In this case, any nodes deleted by the update remain flagged forever. If any node is not flagged successfully, the attempt to perform the update has failed and *backtrack CAS* steps (line 103) are used to unflag any nodes that were already flagged.

If any child CAS step is executed inside help(I), the update operation is successful and it is linearized at the first such child CAS. If a replace operation performs two different child CAS steps, it first executes a child CAS to insert the new key, and then a child CAS to delete the old key. In this case, the replace operation also flags the leaf node of the old key before the first child CAS step. We say the leaf is *logically removed* from the trie at the first child CAS step. Any operation that reaches the leaf node after this determines that the key is already removed. We say a node is *reachable* at time T if there is a path from the root to the node at time T. We say a leaf node is *logically in the trie* at time T if the node is reachable and not logically removed at time T. The leaves that are logically in the trie contain exactly the set of keys in the set D.

Whenever a child pointer is changed, the old child is permanently flagged and it is removed from the trie to avoid the ABA problem. (In some cases, this requires the update to add a new copy of the old child to the trie.) When a call to help(I) performs a child CAS on $I.par[i]$ (for some i), it uses $I.old[i]$ as the old value. Since there is no ABA problem, only the first such CAS on $I.par[i]$ can succeed. Moreover, we prove that the flagging mechanism ensures that this first CAS does succeed. Since processes might call help(I) to help each other to complete their operations, there might be a group of child CASs on each node. However, the *child* pointer is changed exactly once for the operation.

C. Detailed Description of Algorithms

First, we explain the routines that operations call. A **search**(v) is used by updates and find operation to locate key v within the trie. The search(v) starts from the root node and traverses down the trie. At each step of the traversal, search(v) chooses the child according to the appropriate bit of v (line 78). The search(v) stops if it reaches an internal node whose *label* is not a prefix of v. We show that any node visited by the search was reachable at some time during the search. If the search(v) does not return a leaf containing v, there was a time during the search when no leaf containing v was reachable. Moreover, the node that is returned is the location where an insert would have to put v. If search(v) reaches a leaf node and the leaf node is logically in the trie, search(v) sets $keyInTrie$ to true (line 81).

As we shall see, update operations must change the *child* pointers of the parent or grandparent of the node returned by search. The search operation returns gp, p and $node$, the last three nodes reached (where p stands for parent and gp stands for grandparent). A search also returns the values gpI and pI that it read from the *info* fields of gp and p before reading their *child* pointers. More formally, if search(v) returns $\langle gp, p, node, gpI, pI, keyInTrie\rangle$, it satisfies the following postconditions.

(1) At some time during the search, $gp.info$ was gpI (if gp is not null).
(2) Later during the search, p was a child of gp (if gp is not null).
(3) Later during the search, p.*info* was pI.
(4) Later during the search, p.*child*$[i]$ = *node* for some i, and $(p.label) \cdot i$ is a prefix of v.
(5) If $node$ is internal, $node.label$ is not a prefix of v.
(6) If $keyInTrie$ is true, the node whose *label* is v is logically in the trie at some time during search(v).
(7) If $keyInTrie$ is false, then at some time during the search, no node containing v is logically in the trie.

After performing search, an update calls **newFlag** to create a Flag object. For each node that the update must flag, a value read from the *info* field during search of the node is passed to newFlag as the old value to be used in the flag CAS step. The old value for a flag CAS was read before the old value for the corresponding child CAS, so if the flag CAS succeeds, then the node's *child* field has not been changed since the last time its old value was read. The newFlag routine checks if all old values for *info* fields are Unflag objects (line 107). If some *info* field is not an Unflag object, then there is some other incomplete update operating on that node. The newFlag routine tries to complete the incomplete update (line 108), and then returns

18) **insert**($v \in U$)
19) while(true)
20) $I \leftarrow$ null
21) $\langle -, p, node, -, pI, keyInTrie \rangle \leftarrow$ **search**(v)
22) if $keyInTrie$ then return false
23) $nodeI \leftarrow node.info$
24) $copy \leftarrow$ new copy of $node$
25) $new \leftarrow$ **createNode**($copy$, new Leaf containing v, $nodeI$)
26) if $new \neq$ null then
27) if $node$ is Internal then
28) $I \leftarrow$ **newFlag**($[p, node]$, $[pI, nodeI]$, $[p]$, $[p]$, $[node]$, $[new]$, null)
29) else $I \leftarrow$ **newFlag**($[p]$, $[pI]$, $[p]$, $[p]$, $[node]$, $[new]$, null)
30) if $I \neq$ null and **help**(I) then return true

31) **delete**($v \in U$)
32) while(true)
33) $I \leftarrow$ null
34) $\langle gp, p, node, gpI, pI, keyInTrie \rangle \leftarrow$ **search**(v)
35) if $\neg keyInTrie$ then return false
36) $sibling \leftarrow p.child[1 - (|p.label| + 1)$th bit of $v]$
37) if $gp \neq$ null then
38) $I \leftarrow$ **newFlag**($[gp, p]$, $[gpI, pI]$, $[gp]$, $[gp]$, $[p]$, $[sibling]$, null)
39) if $I \neq$ null and **help**(I) then return true

40) **replace**($v_d \in U$, $v_i \in U$)
41) while(true)
42) $I \leftarrow$ null
43) $\langle gp_d, p_d, node_d, gpI_d, pI_d, keyInTrie_d \rangle \leftarrow$ **search**(v_d)
44) if $\neg keyInTrie_d$ then return false
45) $\langle -, p_i, node_i, -, pI_i, keyInTrie_i \rangle \leftarrow$ **search**(v_i)
46) if $keyInTrie_i$ then return false
47) $nodeI_i \leftarrow node_i.info$
48) $sibling_d \leftarrow p_d.child[1 - (|p_d.label| + 1)$th bit of $v_d]$
49) if $gp_d \neq$ null and $node_i \notin \{ node_d, p_d, gp_d \}$ and $p_i \neq p_d$ then
50) $copy_i \leftarrow$ new copy of $node_i$
51) $new_i \leftarrow$ **createNode**($copy_i$, new Leaf containing v_i, $nodeI_i$)
52) if $new_i \neq$ null and $node_i$ is Internal then
53) $I \leftarrow$ **newFlag**($[gp_d, p_d, p_i, node_i]$, $[gpI_d, pI_d, pI_i, nodeI_i]$, $[gp_d, p_i]$, $[p_i, gp_d]$, $[node_i, p_d]$, $[new_i, sibling_d]$, $node_d$)
54) else if $new_i \neq$ null and $node_i$ is Leaf then
55) $I \leftarrow$ **newFlag**($[gp_d, p_d, p_i]$, $[gpI_d, pI_d, pI_i]$, $[gp_d, p_i]$, $[p_i, gp_d]$, $[node_i, p_d]$, $[new_i, sibling_d]$, $node_d$)
56) else if $node_i = node_d$ then
57) $I \leftarrow$ **newFlag**($[p_d]$, $[pI_d]$, $[p_d]$, $[p_d]$, $[node_i]$, [new Leaf containing v_i], null)
58) else if ($node_i = p_d$ and $p_i = gp_d$) or ($gp_d \neq$ null and $p_i = p_d$) then
59) $new_i \leftarrow$ **createNode**($sibling_d$, new Leaf containing v_i, $sibling_d.info$)
60) if $new_i \neq$ null then
61) $I \leftarrow$ **newFlag**($[gp_d, p_d]$, $[gpI_d, pI_d]$, $[gp_d]$, $[gp_d]$, $[p_d]$, $[new_i]$, null)
62) else if $node_i = gp_d$ then
63) $pSib_d \leftarrow gp_d.child[1 - (|gp_d.label| + 1)$th bit of $v_d]$
64) $child_i \leftarrow$ **createNode**($sibling_d$, $pSib_d$, -)
65) if $child_i \neq$ null then
66) $new_i \leftarrow$ **createNode**($child_i$, new Leaf containing v_i, -)
67) if $new_i \neq$ null then $I \leftarrow$ **newFlag**($[p_i, gp_d, p_d]$, $[pI_i, gpI_d, pI_d]$, $[p_i]$, $[p_i]$, $[node_i]$, $[new_i]$, null)
68) if $I \neq$ null and **help**(I) then return true

Figure 3. Update operations

69) **find**($v \in U$)
70) $\langle$ -, -, -, -, -, $keyInTrie \rangle \leftarrow$ **search**(v)
71) return $keyInTrie$

72) **search**($v \in U$)
73) $\langle p, pI \rangle \leftarrow \langle$null, null$\rangle$
74) $node \leftarrow root$
75) while ($node$ is Internal and $node.label$ is prefix of v)
76) $\langle gp, gpI \rangle \leftarrow \langle p, pI \rangle$
77) $\langle p, pI \rangle \leftarrow \langle node, node.info \rangle$
78) $node \leftarrow p.child[(|p.label| + 1)$th bit of $v]$
79) if $node$ is Leaf then ▷ if Leaf is replaced
80) $rmvd \leftarrow$ **logicallyRemoved**($node.info$)
81) $keyInTrie \leftarrow (node.label = v$ and $rmvd =$ false)
82) else $keyInTrie \leftarrow$ false
83) return $\langle gp, p, node, gpI, pI, keyInTrie \rangle$

84) **help**(I: Flag)
85) $i \leftarrow 0$
86) $doChildCAS \leftarrow$ true
87) while ($i < |I.flag|$ and $doChildCAS$)
88) CAS($I.flag[i].info$, $I.oldI[i]$, I) ▷ **flag CAS**
89) $doChildCAS \leftarrow (I.flag[i].info = I)$
90) $i \leftarrow i + 1$
91) if $doChildCAS$ then
92) $I.flagDone \leftarrow$ true
93) if $I.rmvLeaf \neq$ null then $I.rmvLeaf.info \leftarrow I$
94) for $i \leftarrow 0$ to $(|I.par| - 1)$
95) $k \leftarrow (|I.par[i].label| + 1)$th bit of $I.new[i].label$
96) CAS($I.par[i].child[k]$, $I.old[i]$, $I.new[i]$) ▷ **child CAS**
97) if $I.flagDone$ then
98) for $i \leftarrow (|I.unflag| - 1)$ down to 0
99) CAS($I.unflag[i].info$, I, new Unflag) ▷ **unflag CAS**
100) return true
101) else
102) for $i \leftarrow (|I.flag| - 1)$ down to 0
103) CAS($I.flag[i].info$, I, new Unflag) ▷ **backtrack CAS**
104) return false

105) **newFlag**(*flag*, *oldI*, *unflag*, *par*, *old*, *new*, *rmvLeaf*)
106) for $i \leftarrow 0$ to $(|oldI| - 1)$,
107) if *oldI*[i] is Flag then
108) **help**(*oldI*[i])
109) return null
110) if *flag* has duplicates with different values in *oldI* then
111) return null
112) else remove duplicates in *flag* and *unflag* (and corresponding entries of *oldI*)
113) sort elements of *flag* and permute elements of *oldI* accordingly
114) return new Info(*flag*, *oldI*, *unflag*, *par*, *old*, *new*, *rmvLeaf*, false)

115) **createNode**($node_1$: Node, $node_2$: Node, *info*: Info)
116) if $node_1.label$ is prefix of $node_2.label$ or $node_2.label$ is prefix of $node_1.label$ then
117) if *info* is Flag then **help**(*info*)
118) return null
119) else return new Internal whose children are $node_1$ and $node_2$

120) **logicallyRemoved**(I: Info)
121) if I is Unflag then return false
122) return ($I.old[0]$ not in $I.par[0].child$)

Figure 4. The find operation and additional subroutines

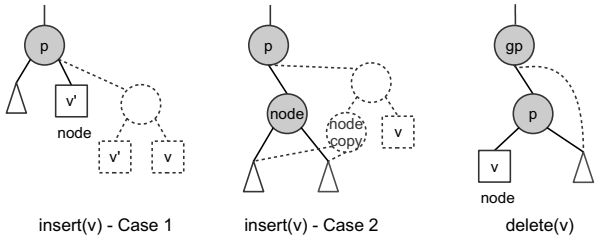

Figure 5. Different cases of insert(v) and delete(v). Triangles are either a leaf node or a subtree. The grey circles are flagged nodes. The dotted lines are the new child pointers that replace the old child pointers (solid lines) and the dotted circles and squares are newly created nodes.

null, which causes the update to restart. In some cases, a replace operation must change two parts of the trie, and if those parts overlap, the list of nodes to be flagged by the replace might contain duplicates. If the duplicate elements do not have the same old values, their *child* fields might have changed since the operation read them, so newFlag returns null and the operation starts over (line 110-111). Otherwise, only one copy of each duplicate element is kept (line 112). The newFlag routine sorts the nodes to be flagged (to ensure progress) and returns the new Flag object (line 113-114).

After an update u creates a Flag object I, it calls **help**(I). This routine attempts to complete the update. First, it uses CAS steps to put the Flag object I in the *info* fields of the nodes to be flagged (line 88). If all nodes are flagged successfully, the *flagDone* field of the Flag object is set to true (line 92). The value of the *flagDone* field is used to coordinate processes that help the update. Suppose a process p is executing help(I). After p performs a flag CAS on a node x, if it sees a value different from I in the x's *info* field, there are two possible cases. The first case is when all nodes were already successfully flagged for I by other processes running help(I), and then x was unflagged before p tries to flag x. (Prior to this unflagging, some process performed the child CAS steps of I successfully.) The second case is when no process flags x successfully for I. Since the *flagDone* field of I is only set to true after all nodes are flagged successfully, p checks the value of the *flagDone* field on line 97 to determine which case happened. If *flagDone* is true, the modifications to the trie for update u have been made. If *flagDone* is false, the update operation cannot be successfully completed, so all internal nodes that got flagged earlier are unflagged by the backtrack CAS steps at line 102-103 and the update u will have to start over.

After flagging all nodes successfully and setting $I.flagDone$, if $I.rmvLeaf$ is non-null, its *info* field is set to I (line 93). Only the two-step replace operations flag a leaf. Then, help(I) changes the *child* fields of nodes in $I.par$ using child CASs (line 94-96). Finally, help(I) uses unflag CASs to unflag the nodes in $I.unflag$ and returns true (line 97-100).

An **insert**(v) operation first calls search(v). Let $\langle$-, p, $node$, -, -, $keyInTrie\rangle$ be the result returned by search(v). If $keyInTrie$ is true, insert(v) returns false since the trie already contains v (line 22). Otherwise, the insertion attempts to replace $node$ with a node created at line 119, whose children are a new leaf node containing v and a new copy of $node$. (See Figure 5.) Thus, the parent p of $node$ must be flagged. A new copy of $node$ is used to avoid the ABA problem. If $node$ is an internal node, since $node$ is replaced by a new copy, insert(v) must flag $node$ permanently (line 28).

A **delete**(v) operation first calls search(v). Let $\langle gp$, p, $node$, -, -, $keyInTrie\rangle$ be the result returned by the search(v). If $keyInTrie$ is false, delete(v) returns false since the trie does not contain v (line 35). Then, delete(v) replaces p by the sibling of $node$. (See Figure 5.) So, delete(v) must flag the grandparent gp and the parent p of $node$ (line 38). Since p is removed from the trie, only gp must be unflagged after the deletion is completed.

A **replace**(v_d, v_i) operation first calls search(v_d) and search(v_i), which return $\langle gp_d$, p_d, $node_d$, -, -, $keyInTrie_d\rangle$ and $\langle$-, p_i, $node_i$, -, -, $keyInTrie_i\rangle$. The replace checks that v_d is in the trie and v_i is not, as in the insert and delete operations (line 43-46). If either test fails, the replace operation returns false.

If insert(v_i) and delete(v_d), as described in Figure 5, would not overlap, replace(v_d, v_i) is done by two child CAS steps and is linearized at the first of these two changes. This is called the general case of replace. Situations when the insertion and deletion would occur in overlapping portions of the trie are handled as special cases as shown in Figure 6. In the special cases, the replace operation changes the trie with one child CAS.

In the general case of the replace operation (line 49-55), we create a Flag object which instructs the help routine to perform the following actions. The replace flags the same nodes that an insert(v_i) and a delete(v_d) would flag. After flagging these nodes, the leaf $node_d$ also gets flagged. Then, v_i is added to the trie, as in insert(v_i). When the new leaf node is added, the leaf $node_d$, which contains v_d, becomes logically removed, but not physically removed yet. Then, $node_d$ is physically deleted as in delete(v_d). After $node_d$ is flagged, any search that reaches $node_d$ checks if p_i is a parent of the old child of p_i using $node_d.info$. If it is not, it means the new leaf containing v_i is already inserted and the operation behaves as if v_d is already removed.

There are four special cases of replace(v_d, v_i) where the changes required by the insertion and deletion are on the overlapping portions of the trie and the replace operation is done using one child CAS step. Although the code for these cases looks somewhat complicated, it simply implement the actions described in Figure 6 by creating a Flag object and calling help. The insertion of v_i replaces $node_i$ by a new node. The cases when the deletion must remove $node_i$ or

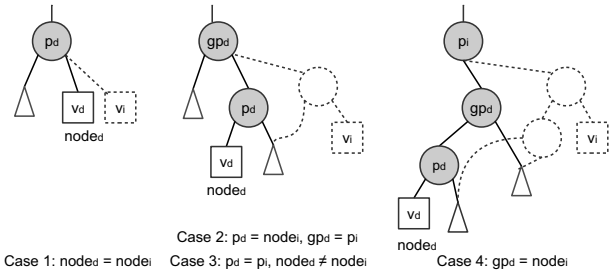

Figure 6. Special cases of replace(v_d, v_i).

change $node_i.child$ are handled as special cases. So, the case that $node_d = node_i$ is one special case (line 56-57). In the deletion, p_d is removed, so the case that $p_d = node_i$ or $p_d = p_i$ are also handled as a special case (line 58-61). In the deletion, $gp_d.child$ is changed. So, the last special case is when $gp_d = node_i$ (line 62-67). In all special cases, $node_i$ is replaced by a new node. Here, we explain one special case in detail. The others are handled in a similar way. In case 2, $p_d = node_i$ and $gp_d = p_i$ (line 58). So, replace(v_d, v_i) creates an Info object that contains instructions to flag gp_d and $node_i$, replace $node_i$ with a new internal node whose non-empty children are a new leaf node containing v_i and the sibling of $node_d$, and then unflag gp_d (line 58-61).

IV. Algorithm Correctness

A detailed proof of correctness is provided in [26]. It is quite lengthy, so we can only provide a brief sketch here. First, we explain how linearization points are chosen for each operation. Let $\langle$ -, -, $node$, -, -, $keyInTrie\rangle$ be the result returned by search(v). If $keyInTrie$ is true, postcondition (6) of the search says there is a time during the search when $node$ is logically in the trie and the search is linearized at that time. Otherwise, postcondition (7) of the search ensures there is a time during the search when no leaf containing v is logically in the trie and the search is linearized at that time. If an update returns false, it is linearized at the linearization point of the search that caused the update to fail. Let I be a Flag object created by an update. If a child CAS performed by any call to help(I) is executed, the update is linearized at the *first* such child CAS. Next, we sketch the correctness proof in four parts.

Part 1 is the heart of the proof. Consider the flag object I created by some update operation. The goal of Part 1 is to prove that the successful CAS steps performed by all calls to help(I) proceed in the expected order. (See Figure 7.) More precisely, we prove that, first, the flag CAS steps are performed on nodes I.*flag*, ordered according to the nodes' *label*s. We prove that only the first flag CAS (by any of the helpers of I) on each node can succeed. If one of these flag CAS steps fails, then the nodes that have been flagged are unflagged by backtrack CAS steps and all calls to help(I) return false, indicating that the attempt at performing the update has failed. Otherwise, the child CAS steps are performed, and then the unflag CAS steps remove flags from nodes in I.*unflag*. If several helpers perform one of these CAS steps, we prove that the first helper succeeds and no others do. In this case, all calls to help(I) return true.

To do this, we first prove simple properties of search.

Lemma 1. *Search satisfies its postconditions (1) to (5), described in Section III-C.*

Lemma 1 is used to prove that each update preserves the following invariant, ensuring the structure is a trie.

Invariant 2. *If $x.child[i] = y$, then $(x.label) \cdot i$ is a prefix of $y.label$.*

We next show that the ABA problem on the *info* fields is avoided because whenever an *info* field is changed, it is set to a newly created Flag or Unflag object.

Lemma 3. *The info field of a node is never set to a value that has been stored there previously.*

Thus, if several helpers of an Info object try to perform a flag CAS, backtrack CAS or unflag CAS on a node, only the first one can succeed. It follows from the code that these CAS steps proceed in the order shown in Figure 7. Next, we consider child CAS steps.

Lemma 4. *The first child CAS performed by a helper of I on each node in I.par must succeed.*

Lemma 5. *A child field of a node is never set to a node that has been stored there before.*

Lemma 4 and 5 are proved together. They require reasoning about the way flags act as locks. Roughly speaking, we show that a node x must be flagged with a pointer to I when the first child CAS on x by help(I) is performed. Since the flag CAS on x succeeded, this means that x has not been flagged by any other update since the time the update that created I read x.*info* during its search (by Lemma 3). It follows that no other update has flagged x between that read and the child CAS, and hence the child field has not changed during that interval. We prove using Lemma 1 that the old value used by the child CAS is still in x.*child* when the child CAS occurs, so it succeeds. The ABA problem on the *child* fields is avoided because whenever a child pointer is changed, the old child is permanently removed from the trie.

Part 1 of the proof is mostly focused on the structure of the help routine. So, any new update that preserves the main invariants of the trie can be added with minor changes to the correctness proof.

Part 2 proves that search operations are linearized correctly. First, we prove all updates satisfy the following.

Lemma 6. *After a node is removed from the trie, it remains flagged forever.*

Next, we prove the following lemma by induction.

Lemma 7. *Each node a search visits was reachable at some time during the search.*

Lemma 6 and 7 are used to prove postconditions (6) and (7) of search. If a search(v) reaches a leaf containing v and it is not logically removed, there was a time during

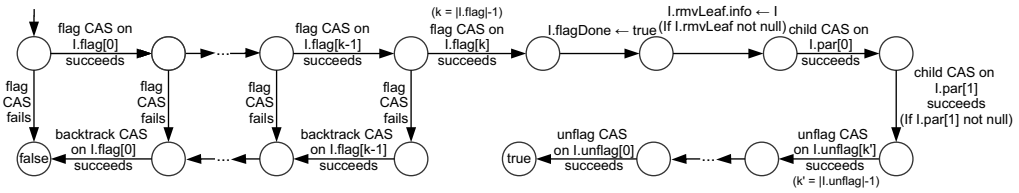

Figure 7. The correct order of steps inside help(I) for each Flag object I. (Steps can be performed by different calls to help(I).)

the search when it was in the trie. It follows from Invariant 2 and Lemma 7 that if a search(v) does not reach a leaf containing v (or it is logically removed), there was a time when v was not in the trie.

Part 3 proves that update operations are linearized correctly. Let T be the linearization point of a successful update operation, which is the first successful child CAS performed by any helper of the operation. We argue, using Lemma 4, that this first child CAS has the effect of implementing precisely the change shown in Figure 5 or 6 atomically. In the case of a replace operation, the linearization point of a successful replace adds a new leaf to the trie. If another operation accesses the leaf node that would be deleted by the replace after that and before the second child CAS, the test performed by logicallyRemoved ensures that it behaves as if the leaf is not in the trie. This is used to establish an invariant that proves all operations return correct results.

Invariant 8. *The leaf nodes that are logically in the trie at time T contain exactly those keys in D, according to the sequence of updates that are linearized before T.*

Finally, part 4 of the proof establishes progress.

Lemma 9. *The implementation is non-blocking.*

To derive a contradiction, assume after time T, no operation terminates or fails. Let I be a Flag object created by an update that is running after T. If a call to help(I) returns true, the update terminates, so after T, all calls to help(I) return false. Thus, all calls to help(I) set $doChildCAS$ to false because they failed to flag an internal node successfully after T. Consider the group of all calls to help(I). We say the group blames an internal node which is the first node that no call to help(I) could flag successfully. Let g_0, ..., g_m be all these groups ordered by the *label*s of the nodes that they blame. Since g_m blames an internal node x, x is flagged by some other group g_i where $0 \leq i < m$. Thus, g_i blames some other node y whose *label* is less than x. So, g_i flags x before attempting to flag y, contradicting the fact that g_i flags internal nodes in order.

V. Empirical Evaluation

We experimentally compared the performance of our implementation (PAT) with non-blocking binary search trees (BST) [8], non-blocking k-ary search trees (4-ST) [20], ConcurrentSkipListMap (SL) of the Java library, lock-based AVL trees (AVL) [11] and non-blocking hash tries (Ctrie) [22]. For the k-ary search trees, we use the value $k = 4$, which was found to be optimal in [20]. Nodes in Ctrie have up to 32 children.

The experiments were executed on a Sun SPARC Enterprise T5240 with 32GB RAM. The machine had two UltraSPARC T2+ processors, each with eight 1.2GHz cores, for a total of 128 hardware threads. The experiments were run in Java. The Sun JVM version 1.7.0_3 was run in server mode. The heap size was set to 2G. This ensures the garbage collector would not be invoked too often, so that the measurements reflect the running time of the algorithms themselves. Using a smaller heap size affects the performance of BST, 4-ST and PAT more than AVL and SL since they create more objects.

We evaluated the algorithms in different scenarios. We ran most experiments using uniformly distributed random keys. We ran the algorithms using uniformly distributed keys in two different ranges: $(0, 10^2)$ to measure performance under high contention and $(0, 10^6)$ for low contention. In the range $(0, 10^2)$, the tree is very small and operations are more likely to access the same part of tree. (We also ran the experiments for the key range of $(0, 10^3)$ and the results were very similar to the low contention case.) We ran experiments with two different operation ratios: 5% inserts, 5% deletes and 90% finds (i5-d5-f90), and 50% inserts, 50% deletes and 0% finds (i50-d50-f0). (We also ran the experiments with ratio of 15% inserts, 15% deletes and 70% finds (i15-d15-f70). Since the results were similar to the experiments with the ratio of i5-d5-f90, we do not present them here.)

Since the replace operation is not used in these sets of experiments, we made some minor optimizations to the pseudo-code. For example, we eliminated the $rmvd$ variable in search operations.

Since the Java compiler optimizes its running code, before each experiment, we run the code for ten seconds for each implementation. We start each experiment with a tree initialized to be half-full, created by running updates in the ratio i50-d50-f0 until the tree is approximately half-full. Each data point in our graphs is the average of eight 4-second trials. (The error bars in the charts show the standard deviation.)

For uniformly distributed keys, algorithms scale well under low contention (key range of $(0, 10^6)$). (See Figure 8.) Under very high contention (key range of $(0, 10^2)$), most

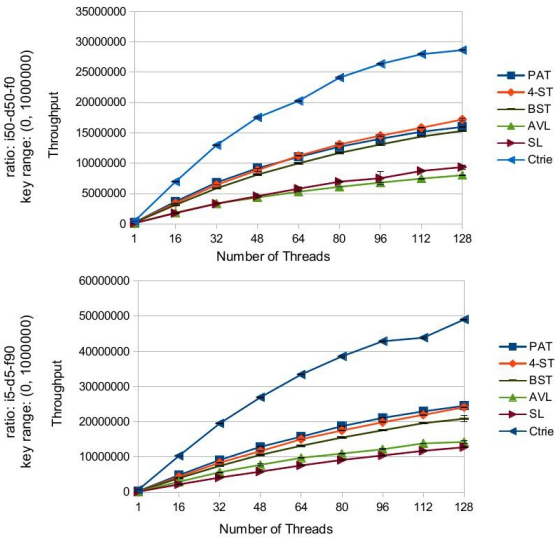

Figure 8. Uniformly distributed keys

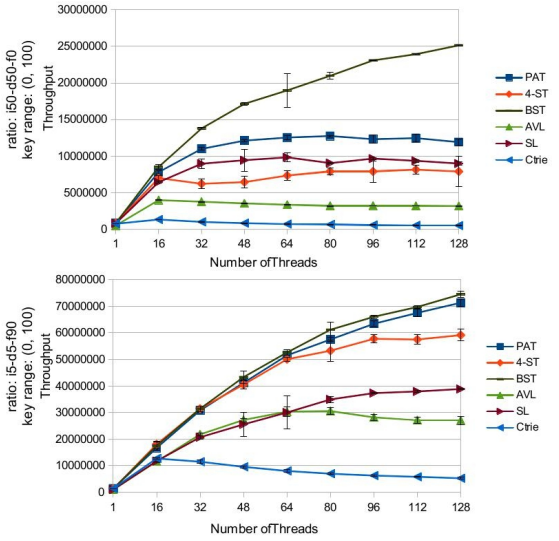

Figure 9. Uniformly distributed keys

scale reasonably well when the fraction of updates is low, but experience problems when all operation are updates. (See Figure 9.) When the range is $(0, 10^6)$, Ctrie outperforms all others because the height of the Ctrie is very small by having nodes with 32 children. However when the range is $(0, 10^2)$ and the contention is very high, Ctrie does not scale. Excluding Ctrie, when the range is $(0, 10^6)$, PAT, 4-ST and BST outperform AVL and SL. Since updates are more expensive than finds, the throughput is greater for i5-d5-f90 than for i50-d50-f0.

To evaluate the replace operations, we ran an experiment with 10% inserts, 10% deletes and 80% replace operations (i10-d10-r80) and a key range of $(0, 10^6)$ on uniformly

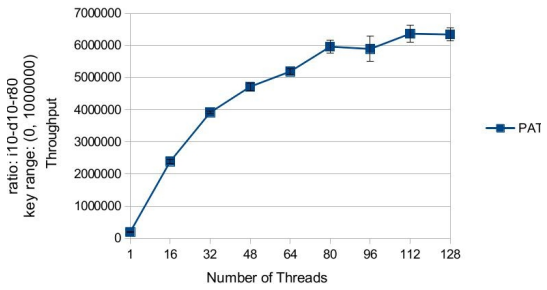

Figure 10. Replace operations of PAT

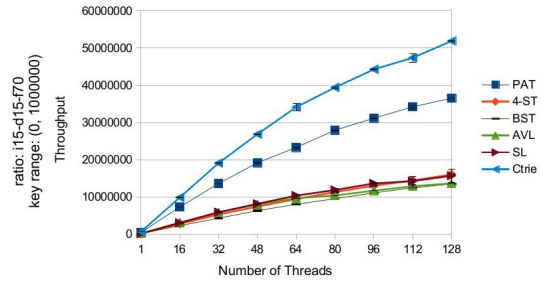

Figure 11. Non-uniformly distributed keys (The lines for 4-ST, BST, AVL and SL overlap.)

random keys. (See Figure 10.) We could not compare these results with other data structure since none provides atomic replace operations. As the chart shows, the replace operation scales well.

We also performed some experiments on non-uniformly distributed random keys. (See Figure 11.) To generate non-uniform keys, processes performed operations on sequence of 50 consecutive keys from the range $(0, 10^6)$, starting from a randomly chosen key. In this experiment, since tries maintain a fixed height without doing expensive balancing operations, Ctrie outperforms all others and PAT outperforms others except Ctrie greatly. Since the results of these experiments for other operations ratios were similar, only the chart for the ratio i15-d15-f70 is presented here. Longer sequences of keys degrade the performance of BST and 4-ST even further.

VI. Conclusion

Our algorithms can also be used to store unbounded length strings. One approach would be to append \$ to the end of each string. To encode a binary string, 0, 1 and \$ can be represented by 01, 10 and 11. Then, every encoded key is greater than 00 and smaller than 111, so 00 and 111 can be used as keys of the two dummy leaves (instead of 0^ℓ and 1^ℓ). Moreover, since *label*s of nodes never change, they need not fit in a single word.

The approach used in the replace operation can be used for operations on other data structures that must change several pointers atomically. Future work includes providing

the general framework for doing this on any tree-based structure. Such a framework would have to guarantee that all changes become visible to query operations at the same time. Brown et al. [27] have recently proposed a general technique for non-blocking trees that support one change to the tree atomically.

Since our algorithms create many Flag objects to avoid using locks, finding more efficient memory management techniques is an important area for future work.

ACKNOWLEDGMENT

I thank my supervisor, Eric Ruppert for his great guidance, advice and support, Trevor Brown for providing lots of help and code for the experiments, Michael L. Scott for giving us access to his multicore machines and the anonymous reviewers for helpful comments.

REFERENCES

[1] D. R. Morrison, "PATRICIA - practical algorithm to retrieve information coded in alphanumeric," *J. ACM*, vol. 15, no. 4, pp. 514–534, 1968.

[2] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proc. Workload Characterization, IEEE Intl Workshop*, 2001, pp. 3–14.

[3] A. Pietracaprina and D. Zandolin, "Mining frequent itemsets using Patricia tries," in *Frequent Itemsets Mining Implem.*, 2003.

[4] M. Gan, M. Zhang, and S. Wang, "Extended negative association rules and the corresponding mining algorithm," in *Proc. Intl Conf. on Advances in Machine Learning and Cybernetics*, 2006, pp. 159–168.

[5] P. Bieganski, J. Riedl, J. V. Carlis, and E. F. Retzel, "Generalized suffix trees for biological sequence data: Applications and implementation." in *Proc. Hawaii Intl Conf. on Syst. Sciences*, 1994, pp. 35–44.

[6] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[7] M. F. Goodchild, "Geographic information systems and science: today and tomorrow," *Annals of GIS*, vol. 15, no. 1, pp. 3–9, 2009.

[8] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, "Non-blocking binary search trees," in *Proc. ACM Symp. on Principles of Distributed Computing*, 2010, pp. 131–140.

[9] S. V. Howley and J. Jones, "A non-blocking internal binary search tree," in *Proc. ACM Symp. on Parallelism in Algorithms and Architectures*, 2012, pp. 161–171.

[10] D. Cederman and P. Tsigas, "Supporting lock-free composition of concurrent data objects," in *Proc. ACM Intl Conf. on Computing Frontiers*, 2010, pp. 53–62.

[11] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun, "A practical concurrent binary search tree," in *Proc. ACM Symp. on Principles and Practice of Parallel Programming*, 2010, pp. 257–268.

[12] Y. Afek, H. Kaplan, B. Korenfeld, A. Morrison, and R. E. Tarjan, "CBTree: A Practical Concurrent Self-Adjusting Search Tree," in *Proc. Intl Symp. on Distributed Computing*, 2012, pp. 1–15.

[13] W. G. Aref and I. F. Ilyas, "SP-GiST: An extensible database index for supporting space partitioning trees," *J. Intel. Inf. Syst.*, vol. 17, no. 2-3, pp. 215–240, 2001.

[14] P. Zijlstra and R. Hat, "Concurrent pagecache," in *Linux Symp.*, 2007, p. 311.

[15] F. Ellen, P. Fatourou, E. Kosmas, A. Milani, and C. Travers, "Universal constructions that ensure disjoint-access parallelism and wait-freedom," in *Proc. ACM Symp. on Principles of Distributed Computing*, 2012, pp. 115–124.

[16] N. Shavit and D. Touitou, "Software transactional memory," in *Proc. ACM Symp. on Principles of Distributed Computing*, 1995, pp. 204–213.

[17] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2010.

[18] J.-J. Tsay and H.-C. Li, "Lock-free concurrent tree structures for multiprocessor systems," in *Proc. Intl Conf. on Parallel and Distributed Systems*, 1994, pp. 544–549.

[19] G. Barnes, "A method for implementing lock-free shared-data structures," in *Proc. ACM Symp. on Parallel Algorithms and Architectures*, 1993, pp. 261–270.

[20] T. Brown and J. Helga, "Non-blocking k-ary search trees," in *Proc. Intl Conf. on Principles of Distributed Syst.*, 2011, pp. 207–221.

[21] A. Braginsky and E. Petrank, "A lock-free B+tree," in *Proc. ACM Symp. on Parallelism in Algorithms and Architectures*, 2012, pp. 58–67.

[22] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky, "Concurrent tries with efficient non-blocking snapshots," in *Proc. ACM Symp. on Principles and Practice of Parallel Programming*, 2012, pp. 151–160.

[23] M. Fomitchev and E. Ruppert, "Lock-free linked lists and skip lists," in *Proc. ACM Symp. on Principles of Distributed Computing*, 2004, pp. 50–59.

[24] K. Fraser, "Practical lock freedom," Ph.D. dissertation, Cambridge University Computer Laboratory, 2003.

[25] H. Sundell and P. Tsigas, "Scalable and lock-free concurrent dictionaries," in *Proc. ACM Symp. on Applied Computing*, 2004, pp. 1438–1445.

[26] N. Shafiei, "Non-blocking Patricia tries with replace operations," 2012, http://arxiv.org/abs/1303.3626.

[27] T. Brown, F. Ellen, and E. Ruppert, "A general technique for non-blocking trees," 2013, manuscript.