Transactional Interference-less Balanced Tree
Technical Report

Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran

Virginia Tech, Blacksburg, VA, USA.

Abstract. In this paper, we present *TxCF-Tree*, a balanced tree whose design is optimized to support transactional accesses. The core optimizations of TxCF-Tree's operations are: providing a traversal phase that does not use any lock and/or speculation, and deferring the lock acquisition or physical modification to the transaction's commit phase; isolating the structural operations (such as re-balancing) in an interference-less housekeeping thread; and minimizing the interference between structural operations and the critical path of semantic operations (i.e., additions and removals on the tree). We evaluated TxCF-Tree against the state-of-the-art general methodologies for designing transactional trees and we show that TxCF-Tree's design pays off in most of workloads.
Keywords: Balanced Trees, Transactional Memory, Semantic Synchronization, Concurrent Data Structures

1 Introduction

With the growing adoption of multi-core processors, the design of efficient data structures that allow concurrent accesses without sacrificing performance and scalability becomes more critical than before. In the last decade, different designs of the concurrent version of well-known data structures (e.g., lists, queues, hash tables) have been proposed [19]. Balanced binary search trees, such as AVL and Red-Black trees are data structures whose self-balancing guarantees an appealing logarithmic-time complexity for their operations.

One of the main issues in balanced trees is the need for *rotations*, which are complex housekeeping operations that re-balance the data structure to ensure its logarithmic-time complexity. Although rotations complicate the design of concurrent balanced trees, many solutions have already been proposed: some of them are lock-based [5, 8, 9, 2, 3, 12], while others are non-blocking [7, 13, 20, 22].

One of the main limitations of concurrent data structures is that they do not compose. For example, atomically inserting two elements in a tree is difficult: if the method internally uses locks, issues like managing the dependency between operations executed in the same transaction, and the deadlock that may occur because of the chain of lock acquisitions, may arise. Similarly, composing non-blocking operations is challenging because of the need to atomically modify different places in the tree using only basic primitives, such as a CAS operation. Lack of composability is a serious limitation of the current designs, especially for

legacy systems, as it makes their integration with third-party software difficult. In this paper we focus on composable (transactional) balanced trees.

Although the research has reached an advanced point in designing concurrent trees, transactional trees have not reached this point yet. There are two practical approaches, to the best of our knowledge, that enable transactional accesses on a tree: *1)* The first approach is Transactional Memory (TM) [18] which natively allows composability as it speculates every memory access inside an *atomic* block; *2)* The second approach is Transactional Boosting [17] (TB), which protects the transactional access to a concurrent data structure with a set of *semantic* locks, eagerly acquired before executing the operation on the concurrent data structure. Both TM and TB have serious limitations when used for designing transactional trees. Those limitations originate from the same reason: they are both generic, and they do not consider the specific characteristics of balanced trees, which instead are heavily investigated in literature. For example, TM considers every step in the operation, including the rotations, as low-level memory reads/writes, which clearly increases the number of false conflicts. On the other hand, TB uses the underlying concurrent tree as a black-box, which prevents any further customization, and may nullify the internal optimizations of the concurrent tree due to the eagerly acquired semantic locks.

Recently, a third trend, which we name *Optimistic Semantic Synchronization* (OSS), has emerged to overcome the limitations of the above approaches. Examples of this new approach include methodologies like [1, 4, 24, 16, 15, 6]. We used the word *optimistic* because all of these solutions share a fundamental optimism. In fact, the common idea behind the aforementioned methodologies is to split data structures' operations into a *traversal* phase and a *commit* phase. A transaction *optimistically* executes the traversal phase without any locking and/or speculation, and it defers the commit phase to the commit time of the enclosing transaction. Unlike TM and TB, OSS only provides guidelines to design transactional data structures, and it leaves all the development details to the data structure designer, thus enabling the possibility of adding further (data structure-specific) optimizations.

OSS is clearly less programmable than TM and TB, but it has the potential to provide better performance and scalability, especially when applied to complex data structures, like the case of balanced trees. Due to their high abstraction level, none of the methodologies listed above discusses in detail how they can be applied to balanced trees without nullifying the body of work related to highly optimized concurrent (non-transactional) balanced trees.

Inspired by OSS, in this paper we present TxCF-Tree, the first balanced tree that is accessible in a *transactional*, rather than just a *concurrent*, manner without monitoring (speculating) the whole traversal path (like in TM) or nullifying the benefits of the efficient concurrent designs (like in TB). TxCF-Tree offers a set of design and low-level innovations, but roughly it can be seen as the transactional version of the recently introduced *Contention Friendly Tree* (CF-Tree) [9]. The main idea of CF-Tree is to decouple the *structural operations* (e.g. rotations and physical deletions) from the *semantic operations* (e.g. queries,

logical removals, and insertions), and to execute those structural operations in a dedicated *helper* thread. This separation makes the semantic operations (that need to be transactional in TxCF-Tree) simple: each operation traverses the tree non-speculatively (i.e., without instrumenting any accessed memory location); then, if it is a write operation, it locks and modifies only one node. In an abstract way, the TxCF-Tree's semantic operations can be seen as composed of a *traversal* and *commit* phases, which makes CF-Tree a good candidate for being transactionally boosted using OSS.

In addition to the new transactional capabilities, TxCF-Tree claims one major innovation with respect to CF-Tree, which is fundamental for targeting high performance in a transactional (not only concurrent) data structure. Although CF-Tree decouples the structural operations, those operations are executed in the *helper* thread with the same priority as the semantic operations, and without any control on their interference. With TxCF-Tree, we make the structural operations *interference-less* (when possible) with respect to semantic operations. This property is highly desirable because structural operations do not alter the abstract (or semantic) state of the tree, thus they should not force any transaction to abort. To reduce this interference, one operation should behave differently if it conflicts with a structural operation rather than with a semantic operation.

TxCF-Tree uses two new terms, which help to identify those *false-interleaving* cases and alleviate their effect: *structural lock*, which is a type of lock acquired if the needed modifications on the node do not change its abstract (semantic) state; and *structural invalidation*, which is a transactional invalidation raised only because of a structural modification on the tree rather than having actual conflicts at the abstract level. In TxCF-Tree, transactions do not abort if they face structural locks or false-invalidations during the execution of their operations. We further reduce the interference of the *helper* thread by adopting a simple heuristic to detect if the tree is *almost balanced*. If so, we increase the back-off time between two *helper* thread's iterations.

We assessed the effectiveness of TxCF-Tree[1] through an evaluation study. Our experiments show that TxCF-Tree performs better than the other transactional approaches (TB and STM) in almost all of the cases.

2 Background

Optimistic Semantic Synchronization. We use the term *Optimistic Semantic Synchronization* (OSS) to represent a set of recent methodologies that leverage the idea of dividing the transaction execution into phases and optimistically executing some of them without any instrumentation (also called *unmonitored* phases). In this section, we overview some of those approaches.

Optimistic Transactional Boosting (OTB) methodology [15, 16] is the optimistic version of TB. It lists three guidelines to convert any optimistic concurrent data structure into a transactional one. According to OTB's first guideline,

[1] The implementation of TxCF-Tree is available at www.hyflow.org.

every data structure's operation is split into three phases: *traversal*, which is executed without any instrumentation and/or locking until reaching the position of interest in the data structure; *validation*, which checks the validity of the unmonitored traversal's outcome; and *commit*, which acquires the necessary locks and performs the actual modifications. OTB provides transactional capabilities by *i)* saving the outcome of the *traversal* phase into local *semantic* read/write-sets to be used during the *validation* and *commit* phases; and *ii)* deferring operation's commit phase until the commit of the whole transaction. The unmonitored traversal phase is the actual source of OTB's performance gains as it clearly reduces false conflicts. The second guideline of OTB discusses the necessary and sufficient steps to make this transactional version *semantically* opaque [14], which means that if the data structure is only accessed using its defined APIs, then all of its operations are semantically consistent at any time of the transaction execution, even though opacity may not be ensured at the memory level (e.g., due to the unmonitored traversal phase). The third guideline of OTB is to optimize the data structure internally.

Consistency Oblivious Programming (COP) [1, 4] splits the operations into the same three phases as OTB (but under different names). We observe two main differences between COP and OTB. First, COP is introduced mainly to design concurrent data structures and it does not natively provide composability unless changes are made at the hardware level [4]. Second, COP does not use locks at commit. Instead, it enforces atomicity and isolation by executing both the *validation* and *commit* phases using TM transactions.

Partitioned Transactions (ParT) [24] also uses the same trend of splitting the operations into a *traversal* (called *planning*) phase and a *commit* (called *update*) phase, but it gives more general guidelines than OTB. Specifically, ParT does not restrict the planning phase to be a traversal of a data structure and it allows this phase to be any generic block of code. Also, ParT does not obligate the planning phase to be necessarily unmonitored, as in OTB and COP. Instead, it allows both the planning and update phases to be transactions.

Transactional Predication (TP) [6] applies a similar methodology to the aforementioned approaches. However, it solves the specific problem of boosting concurrent sets and maps to be transactional.

Although TxCF-Tree complies with OSS, it is closer to OTB because it uses a well-defined concurrent tree as a base for its design (which fits the terminology of transactional boosting), and it follows the second guideline of OTB to guarantee that the transaction execution is *semantically opaque*.

Contention Friendly Tree. Contention Friendly Tree (CF-Tree) [9] is an efficient concurrent lock-based (internal) tree, which finds its main innovation on decoupling the semantic operations (i.e., search, logical deletion, and insertion) from the structural operations (i.e., rotation and physical deletion). The semantic operations are eagerly executed in the original process, whereas the structural operations are deferred to a helper thread. More in details:

Semantic Operations: each semantic operation starts by traversing the tree until it reaches a node that matches the requested key or it reaches a leaf node

(indicating that the searched node does not exist). After that, a search operation returns immediately with the appropriate result without any locking. For a deletion, if the node exists and it is not marked as deleted, the node is locked and then the *deleted* flag is set (only a logical deletion), otherwise the operation returns false. For a successful insertion, the *deleted* flag is cleared (if the node already exists but marked as *deleted*) or a new node is created and linked to the leaf node (if the node does not exist). An unsuccessful insertion simply returns false. In all cases, each operation locks at most one node.

Rotations: re-balancing operations are isolated in a *helper* thread that scans the tree seeking for any node that needs either a rotation or a physical removal. Rotation in this case is relaxed, namely it uses local heights. Although other threads may concurrently modify these heights (resulting in a temporarily unbalanced tree), past work has shown that a sequence of localized operations on the tree eventually results in a strictly balanced tree [5, 21]. A rotation locks: the node to be rotated down; its parent node; and its left or right child (depending on the type of rotation). Also, rotations are designed so that any concurrent semantic operation can traverse the tree without any locking and/or instrumentation. To achieve that, the rotated-down node is cloned and the cloned node is linked to the tree instead of the original node.

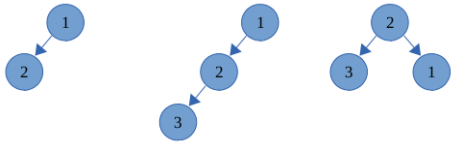
Physical Deletion: The physical deletion is also decoupled and executed separately in the *helper* thread. In addition, a node's deletion is relaxed by leaving a "routing" node in the tree when the deleted node has two children (it is known that deleting a node with two children requires modifying nodes that are far away from each other, which complicates the operation). The physical deletion is done as follows: both the deleted node and its parent are locked, then the node's left and right children links are modified to be pointing at its parent, and finally the node is marked as *physically removed*. This way, concurrent semantic operations can traverse the tree non-speculatively without being lost.

Among the concurrent trees presented in literature, we select CF-Tree as a candidate to be transactionally boosted because it provides the following two properties that fit the OSS principles. First, it uses a lock-based technique for synchronizing the operations, which simplifies the applicability of the OSS methodology. Second, CF-Tree is traversed without any locking and/or speculation, allowing the separation of an unmonitored traversal phase. Also, the semantic operations (`add`, `remove`, and `contains`) are decoupled from the complex structural operations (although they can interfere with each other), like rotations and physical removals, allowing a simple commit phase.

3 Reducing the interference of structural operations

Balanced trees store data according to a specific balanced topology so that their operations can take advantage of the efficient logarithmic-time complexity. More specifically, operations are split into two parts: a "semantic" part, which modifies the abstract state of the tree, and a structural part, which maintains the efficient

organization of the tree. For example, consider the balanced tree in Figure 1[2]. The tree initially represents the abstract set {1, 2} (Figure 1(a)). If we want to insert 3, we first create a new node and link it to the tree in the proper place (Figure 1(b)). Subsequently, the tree is re-balanced because this insertion unbalanced a part of it (Figure 1(c)). Semantically, we can observe the new abstract set, {1, 2, 3}, right after the first step and before the re-balancing step. However, without the re-balancing step, the tree structure itself may become eventually skewed, and any traversal operation on the tree would take linear time rather than logarithmic time.



(a) Initial state (b) Insert 3 (c) Rotate

Fig. 1. An insertion followed by a right rotation in a balanced tree.

Although the structural operations are important, like the aforementioned rotations in our case, they represent the main source of conflicts when concurrent accesses on the tree occur. Two independent operations (like inserting two nodes in two different parts of the tree) may conflict only because one of them needs to re-balance the tree. This additional conflict generated by structural operations can significantly slow down the performance of transactional data structures more than their concurrent versions due to two reasons. First, in long transactions, the time period between the tree traversal and the actual modification during commit may be long enough to generate more conflicts because of the concurrent re-balancing. Second, in transactional data structures, any conflict can result in the abort and re-execution of the whole transaction, which possibly includes several non-conflicting operations, unlike concurrent operations that just re-traverse the tree if a conflict occurs.

Although CF-Tree decouples the structural operations in a dedicated *helper* thread, which forms an important step towards shortening the critical path of the processing (i.e., the semantic operations), it does not prevent the structural operations running in the *helper* thread from interfering with the semantic operations and delaying/aborting them. To minimize such a interference, we propose the following simple guideline (named *G-Pr*):

"*Semantic operations should have higher priority than structural operations.*"

This guideline allows semantic operations to proceed if a conflict with structural modifications occurs. Our rationale is that, delaying (or aborting) semantic operations affects the performance, whereas delaying (or aborting) structural operations only defers the step of optimizing the tree to the near future.

[2] We assume that higher keys are in the left sub-tree to match CF-Tree's design.

4 TxCF-Tree

In this section, we discuss how to boost CF-Tree to be transactional using the OSS principles. The key additions of TxCF-Tree over CF-Tree are: *i)* supporting transactional accesses; and *ii)* minimizing the interference between semantic and structural operations. to simplify the presentation, we focus on the changes made on CF-Tree to achieve those two goals, and we briefly mention the unchanged parts whose details can be found in [9].

Each node in TxCF-Tree contains the same fields as CF-Tree: a key (with no duplication allowed), two pointers to its left and right children, a boolean *deleted* flag to indicate the logical state of the node, and an integer *removed* flag to indicate the physical state of the node (a value from the following: `NOT-REMOVED`, `REMOVED`, or `REMOVED-BY-LEFT-ROTATION`). The node structure in TxCF-Tree is only different in the locking fields. In CF-Tree, each node contains only one lock that is acquired by any operation modifying the node. In TxCF-Tree, each node has two different locks: a *semantic-lock*, which is acquired by the operations that modify its semantic state (either the *deleted* or the *removed* flag); and a *structural-lock*, which is a acquired by the operations that modify the structure of the tree without affecting the node itself (i.e. modifying the right or left pointers). Each lock is associated with a *lock-holder* field that saves the ID of the thread that currently holds the lock, which is important to avoid deadlocks.

TxCF-Tree implements a *set* interface with the semantic operations: `add`, `remove`, and `contains`. Extending TxCF-Tree to have key-value pairs is simple, but for clarity we assume that the value of the node is the same as its key.

4.1 Structural Operations

The *helper* thread repeatedly calls a recursive depth-first procedure to traverse the entire tree. During this procedure, any unbalanced node is rotated and any logically removed node is physically unlinked from the tree. To minimize the interference of this *housekeeping* procedure, we use an adaptive back-off delay after each traversal iteration. We use a simple hill-climbing mechanism that increases (decreases) the back-off time if the number of housekeeping operations in the current iteration is less (greater) than the most recent iteration. While acknowledging the simplicity of the adopted heuristic, it showed effectiveness in our evaluation study.

Physical Deletions. We start by summarizing how the *helper* thread in CF-Tree physically deletes a node N_n (marked as *deleted* and at least one of its children is `null`). First, both N_n and its parent N_p are locked. Then, the node's left and right children fields are modified to point back to the parent (so that the concurrent operations currently visiting N_n can still traverse the tree, without experiencing any interruption) and then N_n is marked as `REMOVED` and unlinked by changing N_p child to be N_n's child instead of N_n.

TxCF-Tree modifies this mechanism by providing *less-interfering* locking. Specifically, we only acquire the *structural-lock* of N_p because its semantic state

will not change. On the other hand, both the *semantic-lock* and the *structural-lock* have to be acquired on N_n because N_n's *removed* flag, which is part of its semantic state, should be set as `REMOVED`. To further minimize the interference, the locking mechanism uses only one CAS trial. If it fails, then the whole structural operation is aborted and the *helper* thread resumes scanning the tree.

Rotations. In CF-Tree, a right rotation (without losing generality) locks three nodes: the parent node N_p, the node to be rotated down N_n, and its left child N_l. Then, rotation is done by cloning N_n and linking the cloned node at the tree instead of N_n (similar to physical deletion, this cloning protects operations whose "unmonitored" traversal phase is concurrently visiting the same nodes. More details are in [9]). Subsequently N_n is marked as `REMOVED` (in case of left-rotation it is marked as `REMOVED-BY-LEFT-ROTATION`) and nodes are unlocked.

In TxCF-Tree, rotations also use a less intrusive locking mechanism. Both N_p and N_l acquire only the *structural-lock* because the rotated-down node N_n is the only node that will change its semantic state (and thus needs to acquire the *semantic-lock*). Also, we found that there is no need to lock the parent node (i.e., N_p) at all. This is because the only change to N_p is to make its left (or right) child pointing to N_l rather than N_n. This means that N_p's child remains not `null` before and after the rotation. Only the *helper* thread can change it to `null` in a later operation by rotating the node down or physically deleting its children. On the other hand, semantic operations only concern about reading/changing the *deleted* flag of a node, if the searched node exists in the tree, or reading/changing a (`null`) link of a node, if the searched node does not exist in the tree. Thus, modifying the child link of N_p cannot conflict with any concurrent semantic operation, thus it is safe to make this modification without locking. Similarly, if all the sub-trees of N_n and N_l are not `null`, then no structural locks are acquired, and the only lock acquired is the *semantic-lock* on N_n.

4.2 Semantic Operations

According to OSS, each operation is divided into the *traversal*, *validation*, and *commit* phases. We follow this division in our presentation.

Traversal. The tree is traversed by following the classical rules of the sequential binary search tree. Traversal ends if we reach the searched node or a `null` pointer. To be able to execute the operation transactionally, the outcome of the traversal phase is not immediately returned. Instead it is saved in a local semantic read/write sets. Each entry of those sets consists of the following three fields. *Op-key*: the searched key that needs to be inserted, removed, or looked up. *Node*: the last node of the traversal phase. This node is either a node whose key matches op-key (no matter if it is marked as *deleted* or not) or a node whose right (left) child is `null` and its item is greater (less) than op-key. *Op-type*: an integer that indicates the type of the operation (`add`, `remove`, or `contains`) and its result (`successful` or `unsuccessful`).

Those fields are sufficient to verify (by the transaction validation) that the result of the operation is not changed since the execution of the operation, and to

modify the tree at commit time. All the operations add an entry to the read-set, but only successful `add` and `remove` operations add entries to the write-set.

Before traversal, the local write-set is scanned for detecting read-after-write hazards. If the key exists in the write-set, the operation returns immediately without traversing the shared tree. Moreover, if a successful `add` operation is followed by a successful `remove` operation of the same item (or vice versa), they locally eliminate each other, in order to save the useless access to the shared tree. The elimination is done only on the write-set, and the entries are kept in the read-set so that the eliminated operations are guaranteed to be consistent.

Validation. The second phase of TxCF-Tree's operation is the *validation* phase. To have a comprehensive presentation, we show first the validation procedure in CF-Tree, and then we show how it is modified in TxCF-Tree.

Algorithm 1 Operation's validation in CF-Tree.

```
1:  procedure VALIDATE(node, k)
2:    if node.removed ≠ NOT-REMOVED then
3:      return false
4:    else if node.k = k then
5:      return true
6:    else if node.k > k then
7:      next = node.right
8:    else
9:      next = node.left
10:   if next = null then
11:     return true
12:   return false
13: end procedure
```

In Algorithm 1, the validation in CF-Tree succeeds if the node's key is not physically removed and either the node's key matches the searched key (line 5) or its child (right or left according to the key) is still `null` (line 11). Otherwise, the validation fails (lines 3 and 12). This validation is used during `add/remove` operations as follows (details are in [9]): each operation traverses the tree until it reaches the involved node, then it locks and validates it (using Algorithm 1). If the validation succeeds, the operation stops its traversal loop and starts the actual insertion/deletion. If the validation fails, the node is unlocked and the operation continues the traversal. In [9], it has been proven that continuing the traversal is safe even if the node is physically deleted or rotated by the *helper* thread, due to the mechanism used in the deletion/rotation, as discussed in Section 4.1 (e.g., modifying the left and right links of the deleted node to be pointing to its parent before unlinking it).

Algorithm 2 Example of semantic opacity.

```
1:  @Atomic                    ▷ initially the tree is empty
2:  procedure T1
3:    if tree.contains(x) = false then
4:      if tree.contains(y) = true then
5:        ...                  ▷ hazardous action
6:  end procedure

7:  @Atomic
8:  procedure T2
9:    tree.add(x)
10:   tree.add(y)
11: end procedure
```

In TxCF-Tree, this validation procedure is modified to achieve two goals.
The first goal regards the correctness: since TxCF-Tree is a transactional tree, validation has also to ensure that the operation's result is not changed until

transaction commits; otherwise, the transaction consistency is compromised. As an example, in Algorithm 2 let us assume the following invariant: y exists in the tree if and only if x also exists. If we use the same validation as Algorithm 1, $T1$ may execute line 3 first and return false. Then, let us assume that $T2$ is entirely executed and committed. In this case, $T1$ should abort right after executing line 4 because it breaks the invariant. Aborting the doomed transaction $T1$ should be immediate and it cannot be delayed until the commit phase because it may go into an infinite loop or raise an exception (line 5). To prevent those cases, all of the read-set's entries have to be validated (using Algorithm 3 instead of Algorithm 1) after each operation as well as during commit.

Algorithm 3 Operation's validation in TxCF-Tree.

```
1:  procedure VALIDATE(read-set-entry)
2:    if entry.op-type ∈ (unsuccessful add,
3:  successful remove/contains) then
4:      item-existed = true
5:    else
6:      item-existed = false
7:    if entry.node.removed ≠
8:  NOT-REMOVED then
9:      return STRUCTURALLY-INVALID
10:   else if entry.node.k = entry.op-key then
11:     if entry.node.deleted xor
12: item-existed then
13:       return VALID
14:     else
15:       return SEMANTICALLY-INVALID
16:   else if entry.node.k > entry.op-key then
17:     next = node.right
18:   else
19:     next = node.left
20:   if next = null then
21:     if item-existed then
22:       return SEMANTICALLY-INVALID
23:     else
24:       return VALID
25:   return STRUCTURALLY-INVALID
26: end procedure
```

The second goal regards performance: if the node is physically removed or its child becomes no longer `null` (which are the invalidation cases of CF-Tree), that does not mean that the transaction is not consistent anymore. It only means that the traversal phase has to continue and reach a new node to be validated. It is worth noting that aborting the transaction in those cases does not impact the tree's correctness, while its performance will be affected. In fact, this conservative approach increases the probability of structural operations' interference. For this reason we distinguish between those types of invalidations and the actual *semantic* invalidations, such as those depicted in Algorithm 2. The modified version of the validation is shown in Algorithm 3. The cases covered in CF-Tree are considered *structural-invalidations* (lines 9 and 25), and the actual invalidation cases are considered *semantic-invalidations* (lines 15 and 22).

Algorithm 4 shows how to validate the read-set. For each entry, we firstly check if the entry's node is not locked (lines 4-9). In this step we exploit our lock separation by checking only one of the two locks because each operation validates either the *deleted* flag or the child link. Specifically, if the node's key matches op-key, node's *semantic-lock* is checked, otherwise the *structural-lock* is checked. Moreover, if the entry's node is locked by the *helper* thread, we consider it as unlocked because the helper thread cannot change the abstract state of the tree. The only effect of the *helper* thread is to make the operation structurally invalid, which can be detected in the next steps.

Algorithm 4 Read-set validation in TxCF-Tree.

```
1:  procedure VALIDATE-READSET(read-set)
2:    for all entries in the read-set do
3:      while true do
4:        if entry.op-item = entry.node.item then
5:          lock = semantic-lock
6:        else
7:          lock = struct-lock
8:        if lockedNotByMeOrHelper(lock) then
9:          return false
10:       r = VALIDATE(entry)
11:       if r = STRUCTURALLY-INVALID then
12:         newNode = CONT-TRAVERSE(entry)
13:         entry.node = newNode
14:         write-entry = write-set.get(entry.op-key)
15:         if write-entry ≠ null then
16:           write-entry.node = newNode
17:       else if r = SEMANTICALLY-INVALID
      then
18:         return false
19:       else
20:         break;
21:   return true
22: end procedure
```

The next step is to validate the entry itself (line 10). If it is *semantically-invalidated*, then the transaction aborts (line 18). If it is *structurally-invalidated*, the traversal continues as in CF-Tree and the entry is updated with the new node (lines 12-16), then the node is re-validated. If the operation is a successful *add/remove*, the related write-set entry is also updated (line 16).

Commit. The *commit* phase (Algorithm 5) is similar to the classical two-phase locking mechanism. The nodes in the read/write sets are locked and/or validated first, then the tree is modified, and finally locks are released.

Algorithm 5 Commit in TxCF-Tree.

```
1:  procedure COMMIT
2:    for all entries in the write-set do
3:      while true do
4:                          ▷ Try to acquire the lock
5:        if entry.op-item = entry.node.k then
6:          lock = semantic-lock
7:        else
8:          lock = struct-lock
9:        if lockholder ≠ myID
10:   and !lock.acquire then
11:         if lockholder ≠ helperID then
12:           ABORT
13:         else
14:           continue
15:                              ▷ Inline Validation
16:                       ▷ Similar to Algorithm 4
17:                  ▷ But unlock before retrying
18:       result = VALIDATE(entry)
19:       ...
20:     ▷ Validate the remaining read-set entries
21:                       ▷ Exactly like Algorithm 4
22:     ▷ But skips the entries that are also in the
      write-set
23:   VALIDATE-READ-OPERATIONS(read-
      set)
24:                                ▷ Publish write-sets
25:   for all entries in the write-set do
26:     if entry.op-type = remove then
27:       entry.node.deleted = true
28:     else                              ▷ add operation
29:       if entry.op-item = entry.node.item then
30:         entry.node.deleted = false
31:       else
32:         newNode = CREATE-NODE(entry.key)
33:         node = CONT-TRAVERSE(entry)
34:         if node.key > entry.k then
35:           node.right = newNode;
36:         else
37:           node.left = newNode;
38:                                          ▷ Unlock
39:   UNLOCK(write-set)
40:   return true
41: end procedure
```

From the commit procedure of TxCF-Tree it is worth mention the following points. The first point is how TxCF-Tree solves the issue of having two dependent operations in the same transaction. For example, if two *add* operations are using

the same node (e.g. assume a transaction that adds both 3 and 4 to the tree shown in Figure 1). The effect of the first operation (add 3) should be propagated to the second one (add 4). To achieve that, the add operation uses the node in the write-set only as a starting point and keeps traversing the tree from this node until reaching the new node. Also, the operations lock the added nodes (3 and 4 in our case) before linking them to the tree. Those nodes are unlocked together with the other nodes at the end of the commit phase. Any interleaving transaction or structural operation running in the *helper* thread cannot force the transaction to abort because all the involved nodes are already locked. Also, the other cases of having dependent operations, such as adding (or removing) the same key twice and adding a key and then removing it, are solved earlier during the operation itself (as mentioned in the traversal phase).

The second point is how TxCF-Tree preserves the reduced interferences between the structural and the semantic operations without hampering the two-phase locking mechanism. The main issue in this regard is that *structural invalidations* may not abort the transaction. Thus, a transaction cannot lock the nodes in the write-set and then validate the nodes in the read-set because, if so, in case of a *structural invalidation*, the invalidated operation (which can be a write operation) would continue traversing the tree and reach a new node (which is not yet locked). To solve this problem, we use an *inline* validation of the entries in the write-set. The write-set entries are both locked and validated at the same time. If the write operation fails in its validation: *1)* it unlocks the node; *2)* re-traverses the tree; *3)* locks the new node; and *4)* re-validates the entry.

5 Correctness

Since we use the OTB methodology to make CF-Tree transactional, the correctness of TxCf-Tree will be inherited from the correctness of CF-Tree. The main difference is that the transactions in TxCF-Tree (rather than the operations in Cf-Tree) are serialized (rather than linearized in CF-Tree) as described in [15]. The serialization point of a read-write transaction is the point right after acquiring the locks and before the (successful) validation during commit. For a read-only transaction, the serialization point is the return of its last read operation. Both those points are immediately followed by a validation procedure (Algorithm 4). If this validation succeeds, then all the transaction operations are guaranteed to be consistent.

More in detail, the correctness of TxCf-Tree can be viewed in two steps. First, we show, without loss of generality, that if each transaction is composed of only one TxCF-Tree operation then the operations are linearizable. Then, we prove that if the transaction contains more than one operation then it is opaque.

Theorem 1. *The linearization of CF-Tree's operations is preserved in TxCF-Tree if it is accessed non-transactionally (one operations per transaction).*

Proof. Each operation traverses the tree following the same rules as in CF-Tree. After the traversal, we can distinguish between write and read operations'

behavior. A write operation, instead of acquiring the locks on the involved nodes instantaneously after the traversal, it acquires the same locks, but at transaction commit time. Since the transaction is validated after the locks acquisition using the same validation done by CF-Tree, the linearization points of each write operation is just shifted to the commit phase of the transaction (rather than after the operation as in CF-Tree). Linearizing the read operations is easier in TxCF-Tree, because they become no longer wait-free, and the transaction checks (during the validation procedure) that the involved node is both unlocked and valid. The step of checking the lock of each operation's node is the return point that can be safely used for linearizing the operation.

The correctness of the mechanisms used to achieve *interference-less* structural operations can be inferred as follows.

i) Splitting locks into *structural* and *semantic* locks does not affect correctness by any mean, because any two conflicting operations (e.g., two operations that attempt to delete the same node, or two operations that attempt to insert new nodes on the same link) acquire the same type of lock.

ii) Structural invalidations are raised and handled in the same way as CF-Tree (as we show in Algorithms 1 and 3). Since we use the same approach for rotation and physical deletion (e.g., cloning the rotated down node and linking the physically deleted node to its parent), re-traversing the tree after a *structural invalidation* is guaranteed to be safe as in CF-Tree itself (see [9] for the complete proof of validation in CF-Tree).

iii) Semantic invalidations preserve the consistency among the operations within the same transaction. Unlike *structural invalidations*, in those cases, the whole transaction is aborted.

iv) The *inline* validation during commit does not affect the correctness (although it violates two-phase locking) because every *inline-validated* node is locked before being validated and cannot be invalidated anymore if the validation succeeds.

v) Validating the whole read-set after each operation and before committing preserves consistency in the presence of concurrent structural operations. For example, assuming the scenario where a structural operation physically removes a node that is used by a running transaction T1, which can be followed by a semantic operation (executed in another transaction T2) that adds this node in a different place of the tree. Although this new addition will not be detected by T1's validation, the expected race condition will be solved because T1 will detect during the validation (after the next operation or at commit) that the *removed* flag of the node has been changed (line 8 in Algorithm 3) and will continue traversing the tree. At this point, T1 will reach the same new node as T2, and they will be serialized independently from the structural operation.

It is clear that TxCF-Tree's operations are not opaque at memory level (i.e., in a history composed of all the memory locations accessed while performing the semantic operations). This is mainly because each operation in TxCF-Tree traverses the tree non-speculatively and all the reads during this traversal phase can be invalidated by any concurrent transaction. However, our target is to make

TxCF-Tree *semantically* consistent. To prove that, we first define the return points of TxCF's operations as follows:

Definition 1. *The return point of a read operation (i.e. a* `contains` *operation or an unsuccessful* `add/remove` *operation) is a point that exists at any place in the execution between the invocation and the commit of the operation and reflects the return value of the operation (either true or false).*

In other words, this *return point* is the linearization point of the operation in the concurrent (non-transactional) CF-Tree, and the corresponding serialization point (that we mentioned above) in the transactional TxCF-Tree. In the same way, we define the return point of the write operation as follows:

Definition 2. *The return point of a write operation (i.e. a successful* `add/remove` *operation) is a point that exists at any place in the execution between the invocation and the commit of the operation and reflects an atomic action of reading the old state of the key and modifying it.*

For example, if the `add(x)` operation is successful, its *return point* is the point that reflects atomically changing the abstract state of the tree from being *not including* `x` to be *including* x.

As in most two-phase-locking-based concurrency control, the serialization point of a read-write transaction is the point right after acquiring the locks and before the (successful) validation during commit (which corresponds to line 23 of Algorithm 5). This point is typically a combination of all its operations' return points. For a read-only transaction, the serialization point is the return of its last read operation. Both points are immediately followed by a validation procedure (Algorithm 4). If this validation succeeds, then all the transaction's operations are guaranteed to be consistent. Also, for a read-write transaction, all its write operations lock their nodes so they cannot interfere with other transactions. This is somewhat similar to the serialization point of the memory-based TL2 algorithm [11]. However, in TL2, read-write transactions increment a global timestamp after locking the nodes, while in TxCF-Tree we replace that mechanism with a validation of the whole read-set after each operation and during commit.

Then we define a history $\mathcal{H}$ of TxCF-Tree operations as the history of each operation's return point (we can also add the invocation point of each operation. However, removing those points will not affect the proof, so we excluded them for clarity). Any history $\mathcal{H}$ is opaque if there is a legal sequential history $\mathcal{S}$ equivalent to $complete(\mathcal{H})$ (which includes the non-committed transactions as well) and respects the real-time order of $\mathcal{H}$. We finally report our theorem that proves the correctness of TxCF-Tree. Throughout the whole proof of this theorem, we assumed a history that contains only TxCF-Tree operations (which means that accesses that do not use the tree's APIs are prevented). We leave the general case, where we allow any kind of accesses in the transaction as a future work.

Theorem 2. *A history $\mathcal{H}$ of TxCF-Tree's operations is opaque.*

Proof. Opacity was proposed in [14] to formally proof the correctness of TM implementations, and most STM algorithms are proven to guarantee opacity [10, 11, 23]. Intuitively, as mentioned in [14], opacity is guaranteed if three requirements are captured: *i)* every committed transaction atomically appears in a single indivisible point in the history of the committed transactions, *ii)* live transactions do not see the intermediate results of any aborted transaction, *iii)* transactions are always consistent even if they will eventually abort. We show that those three requirements are preserved in TxCF-Tree. We borrow the same terminology used in [14]. However, for brevity, instead of having two points in the history for each operation (the *invocation* point and the *return* point), we will only show one point which reflects the *return* point, as we showed earlier.

Equivalence to a legal sequential history:. The first requirement for a history $\mathcal{H}$ to be opaque is that if we remove all non-committed transaction, the resulting sub-history $\mathcal{H}'$ is equivalent to a legal sequential history $\mathcal{S}'$ that preserves the real-time order of the transactions in $\mathcal{H}'$. In TxCF-Tree, $\mathcal{H}'$ preserves the real-time order because all operations are linearized during the commit phases of their transactions. For that reason, a committing transaction can be serialized in one point, right after the transaction successfully acquires its semantic locks. After this serialization point, if the transaction successfully validates its read-set, all conflicting transactions in $\mathcal{H}'$ will be serialized after it. If it fails in validation, it will simply abort.

Precisely, we have five cases to cover for proving the legality of any sub-history $\mathcal{H}'$ of some committed transactions $T_1, T_2, ..., T_n$:

1. Transaction are executed serially: which means that each transaction starts after the previous transaction commits. The real-time order in this case is natively preserved because after T_i commits, all its writes are immediately visible to the following transactions (threads are not caching any state of the objects).
2. Concurrent transactions are independent (which means that they have no intersection in their read/write-sets or the intersection is only between read-sets). Natively, they can be serialized in any order. The history of each transaction as a standalone transaction is kept legal because we check the local write-set first and we continue traversing the tree during commit. Also, the elimination mechanism we used is safe because we only eliminate the operations from the write-set and leave them in the read-set. For example, in the following history:
   $H1 =< add(T_i, x, true), contains(T_i, x, true), remove(T_i, x, true), tryC_{T_i}, C_{T_i} >$
   Both the `contains` and the `remove` operations cannot return an illegal value (which is false in this case) during the execution of the transaction because the `add` operation is saved in the write-set. Also, the `remove` operation eliminates the `add` but they will be validated correctly at commit because their entries are in the read-set.
3. The write-sets of two concurrent transactions, T_i and T_j, intersect. Clearly the commit phases of those transactions can never execute concurrently. Either one of them will fail in acquiring the semantic locks and thus will

abort, or T_j will start its commit after T_i entirely finishes its commit and releases its locks, which allows serializing T_i before T_j.

4. The read-set of T_i intersects with the write-set of T_j and the read-set of T_j does not intersect with the write-set of T_i. In this case, T_i will either abort during the commit-time validation, or it will successfully finish its validation before T_j acquires the "conflicting" semantic locks. In the latter case, T_i can be safely serialized before T_j.
5. The read-set of T_i intersects with the write-set of T_j and the read-set of T_j intersects with the write-set of T_i. In this case, any scenario where both transactions concurrently commit is illegal. For example, in the following two histories[3]:
   $H2 =< remove(T_k, x, true), remove(T_k, y, true), tryC_{T_k}, C_{T_k}, add(T_i, x, true),$
   $contains(T_j, x, false), add(T_j, y, true), contains(T_i, y, false), tryC_{T_i}, C_{T_i}, tryC_{T_j}, C_{T_j} >$
   $H3 =< remove(T_k, x, true), remove(T_k, y, true), tryC_{T_k}, C_{T_k}, add(T_i, x, true),$
   $contains(T_j, x, true), add(T_j, y, true), contains(T_i, y, true), tryC_{T_i}, C_{T_i}, tryC_{T_j}, C_{T_j} >$
   Both histories are illegal because the `contains` operations in T_i and T_j cannot return both false or both true[4]. A possible legal case is that the `contains` operation of T_i returns false and the one of T_j returns true (which allows T_i to be legally serialized before T_j).
   Our validation process in Algorithm 4 prevents that all these illegal scenarios can happen. As we validate that the nodes in the read-set are both *unlocked* and *valid*. T_i and T_j cannot both successfully acquire the semantic locks and then successfully validate their read-sets before starting to write. At least one transaction will abort because some entries in its read-set is locked by the other transaction.

The effect of the aborted transactions:. Aborted transactions in TxCF-Tree have no effect on the live transactions. This is simply because transactions do not publish any writes until their commit phase. During commit, if a transaction successfully acquires the semantic locks and then it successfully validates its read-set, it cannot abort anymore. Accordingly, it is safe at this point to start writing on the shared tree.

Consistency of live transactions:. Transactions which guarantee opacity should always observe a consistent state. This also includes the *live* transactions, which are the transactions that did not yet commit or abort. Theoretically, as mentioned in [14], we can transform any history which contains some live transactions to a complete history by either committing or aborting those live transactions. The challenge here is to prove that this completed history is still legal (which means that the operations executed so far inside the live transactions are legal). In TxCF-Tree we guarantee that live transactions always observe a consistent state by the *post-validation* procedure which validates, after each operation, that the entire read-set is still valid.

[3] We put the first two operations of T_k to enforce that x and y are both in the *set* before T_i and T_j start.

[4] This case is an example of producing a cyclic opacity graph which is mentioned in [14].

Precisely, in a history $\mathcal{H}$, an operation:
$< op(T_i, x, true/false) >$
can be implicitly extended to either:
$< op(T_i, x, true/false), validate(T_i, succeeded) >$
or:
$< op(T_i, x, true/false),\ validate(T_i, failed), A_{T_i} >$
according to whether its validation succeeds or fails, which guarantees preserving the legality of $\mathcal{H}$.

6 Evaluation

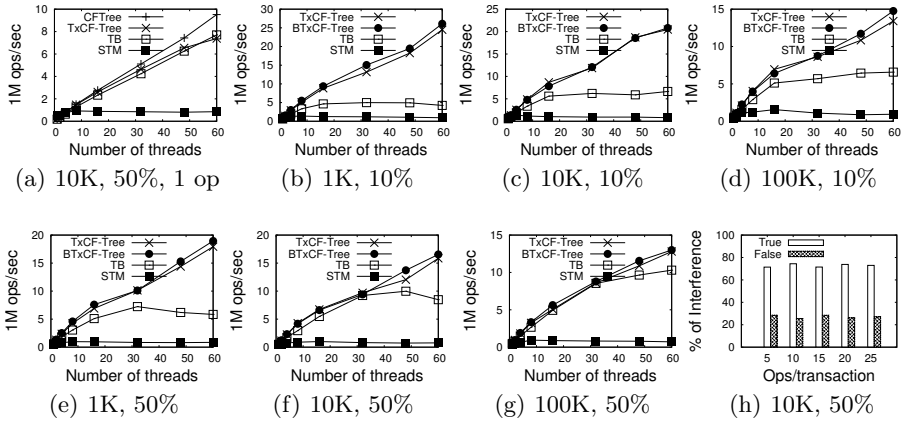
In our experiments we compared the performance of TxCF-Tree with the performance of TB and some STM approaches. Our implementation of TB uses CF-Tree as the underlying (black-box) tree, which makes a fair comparison. Regarding STM, we tested three different algorithms: LSA [23]; TL2 [11]; and NOrec [10], and, to make plots clear, we reported the best performance collected.

All experiments were conducted on a 64-core machine, which has 4 AMD Opteron (TM) Processors, each with 16 cores running at 1.4 GHz, 32 GB of RAM, and 16KB L1 data cache. Throughput is measured as the number of semantic operations (not transactions) per second to have consistent data points. However, since the benchmark executes 256 `no-op` instructions in between two transactions, this may result in different throughput ranges for different sizes of transactions. Each data point is the average of five runs.

In Figure 2(a) we show the results for a scenario that mimics the concurrent (non-transactional) case (i.e., each transaction executes only one operation on the tree). We leverage this plot to show the cost of adopting a transactional solution over a pure concurrent tree. Clearly STM does not scale because it "blindly" speculates on all the memory reads and writes. This poor scalability of STM is confirmed in all the experiments we made. On the other hand, both TB and TxCF-Tree scale better than STM and close to CF-Tree (TxCF-Tree is slightly closer). This behavior shows an overhead that is affordable in case one wants to use the TxCF-Tree library even for just handling the concurrency of atomic semantic operations without transactions.

Figures 2(b)-2(g) show the transactional case, in which we deployed five operations per transaction for different sizes of the tree (1K, 10K, and 100K) and different read/write workloads (10% and 50% of `add/remove` operations). We do not include CF-Tree because it only supports concurrent operations and thus it cannot handle the execution of transactions. TxCF-Tree performs generally better than TB. The gap between the two algorithms decreases when we increase the percentage of the write operations. This is reasonable because the conflict level becomes higher, and it best fits the more *pessimistic* approach (as TB).

Increasing the size of the tree also decreases the gap between TxCF-Tree and TB. At first impression it appears counterintuitive because increasing the size of the tree means generally decreasing the overall contention, which should be better for optimistic approaches like TxCF-Tree. The actual reason is that, in the



(a) 10K, 50%, 1 op (b) 1K, 10% (c) 10K, 10% (d) 100K, 10%

(e) 1K, 50% (f) 10K, 50% (g) 100K, 50% (h) 10K, 50%

Fig. 2. Throughput with one operation (2(a)), and five operations (2(b)-2(g)) per transaction (labels indicate the size of the tree and the % of the `add/remove` operations). Figure 2(h) shows the percentage of the two interference types using 32 threads.

case of very low contention, most of the transactions do not conflict with each other and both algorithms linearly scale. Then, when the conflict probability increases, the difference between the algorithms becomes visible. A comparison between Figure 2(e) and Figure 2(g) (which differ only for the size of the tree) confirms this claim. In Figure 2(e), both algorithms scale well up to 32 threads because threads are almost non-conflicting, then TB starts to suffers from its non-optimized design while TxCF-Tree keeps scaling. On the other hand, in Figure 2(g) both algorithms scale until 60 threads because the tree is large.

Summarizing, analyzing the above results we can identify two points that allow TxCF-Tree to outperform competitors: *i)* having an optimized unmonitored traversal phase that reduces false conflicts, and *ii)* having optimized validation/commit procedures that minimize the interferences between structural and semantic operations. Both TB and TxCF-Tree gain performance by exploiting the first point, in fact TB itself performs (up to an order of magnitude) better than STM. However, only TxCF-Tree uses an *optimized* design for a balanced tree data structure, and it makes its performance generally (much) better than TB. In the aforementioned experiments we use two versions of TxCF-Tree, one with the adaptive back-off time in between two *helper* thread iterations (named BTxCF-Tree), and one without. The results show that this optimization further enhances the performance, especially in the small tree (the cases of 10% `add/remove` operations). This gain may increase with a more effective heuristic.

The last experiment we report regards the capability of TxCF-Tree to reduce interferences with structural operations. Although breaking down TxCF-Tree's operations to measure this gain is not straightforward, we roughly estimated the gain by quantifying two metrics: the *true* interferences count, which is simply the actual transactional aborts count; and the *false* interferences count, which is

the count of the cases in which the transaction does not abort because the tree is re-traversed instead or because the operations in TxCF-Tree acquire only one (structural or semantic) lock. In Figure 2(h) the false-interferences are 25%-30% of the total interferences for different sizes of the transactions.

7 Conclusions

We presented TxCF-Tree, the first interference-less transactional balanced tree. Unlike the former general approaches, it uses an optimized conflict management mechanism that reacts differently according to the type of the operation. Our experiments confirm that TxCF-Tree performs better than the general approaches.

Acknowledgement Authors would thank Vincent Gramoli and anonymous reviewers for the invaluable comments. This work is partially supported by Air Force Office of Scientific Research (AFOSR) under grant FA9550-14-1-0187.

References

1. Yehuda Afek, Hillel Avni, and Nir Shavit. Towards consistency oblivious programming. In *OPODIS*, pages 65–79, 2011.
2. Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert Endre Tarjan. The CB tree: a practical concurrent self-adjusting search tree. *DISC*, 27(6):393–417, 2014.
3. Maya Arbel and Hagit Attiya. Concurrent updates with RCU: search tree as an example. In *PODC*, pages 196–205, 2014.
4. Hillel Avni and Adi Suissa-Peleg. Brief announcement: Cop composition using transaction suspension in the compiler. In *DISC*, pages 550–552, 2014.
5. Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *PPoPP*, pages 257–268, 2010.
6. Nathan Grasso Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. Transactional predication: high-performance concurrent sets and maps for STM. In *PODC*, pages 6–15, 2010.
7. Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *PPoPP*, pages 329–342, 2014.
8. Tyler Crain, Vincent Gramoli, and Michel Raynal. A speculation-friendly binary search tree. In *PPoPP*, pages 161–170, 2012.
9. Tyler Crain, Vincent Gramoli, and Michel Raynal. A contention-friendly binary search tree. In *Euro-Par*, pages 229–240, 2013.
10. Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *PPoPP*, pages 67–78, 2010.
11. Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC*, pages 194–208, 2006.
12. Dana Drachsler, Martin T. Vechev, and Eran Yahav. Practical concurrent binary search trees via logical ordering. In *PPoPP*, pages 343–356, 2014.
13. Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *PODC*, pages 131–140, 2010.

14. Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *PPoPP*, pages 175–184, 2008.
15. Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. On developing optimistic transactional lazy set. In *OPODIS*, pages 437–452, 2014.
16. Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. Optimistic transactional boosting. In *PPoPP*, pages 387–388, 2014.
17. Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *PPoPP*, pages 207–216, 2008.
18. Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.
19. Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint.* Elsevier, 2012.
20. Shane V. Howley and Jeremy Jones. A non-blocking internal binary search tree. In *SPAA*, pages 161–171, 2012.
21. Kim S Larsen. AVL trees with relaxed balance. In *IPPS*, pages 888–893, 1994.
22. Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *PPoPP*, pages 317–328, 2014.
23. Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *DISC*, pages 284–298, 2006.
24. Lingxiang Xiang and Michael L. Scott. Software partitioning of hardware transactions. In *PPoPP*, pages 76–86, 2015.