Transactional Interference-less Balanced Tree

Ahmed Hassan Roberto Palmieri Binoy Ravindran
Virginia Tech
hassan84@vt.edu robertop@vt.edu binoy@vt.edu

Abstract

The concurrent balanced tree is one of the most well-studied data structures since the transition to the multicore era. The last decade witnessed the design of many concurrent, and sometimes relaxed, versions of AVL and Red-Black trees. However, most of those designs do not support transactional accesses to the tree. In this paper, we present *TxCF-Tree*, a transactional balanced tree whose design is optimized to cope with the transactional nature of its operations. The operations of TxCF-Tree are mainly optimized by: *i)* having a traversal phase that does not use any locks and/or speculation, and deferring any lock acquisition or physical modification to the transaction's commit phase; *ii)* isolating the structural operations (such as re-balancing) in an optimized, interference-less housekeeping thread; and *iii)* 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 of designing transactional trees and we show that the optimized design of TxCF-Tree pays off in most of the workloads.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features; E.1 [*Data Structures*]: Concurrent Data Structures

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 the well-known data structures, such as lists, queues, stacks, and hash tables have been proposed [11, 15, 19, 22, 23, 29, 32].

Balanced binary search trees, such as AVL [1] and Red-Black [5] trees are data structures whose self-balancing guarantees a logarithmic-time complexity for their `add`, `remove`, and `contains` operations. One of the main issues in balanced trees is the need for *rotations*, which are complex housekeeping operations that re-balance the tree to ensure the aforementioned logarithmic-time complexity. Although rotations complicate the design of efficient concurrent balanced trees, many solutions have already been proposed: some of them are lock-based [6, 8, 9], while others are non-blocking [7, 25, 30]. Lock-based solutions are easier to design than non-blocking algorithms, but their performance could suffer from: *i)* the blocking nature of their operations; and *ii)* the possibility of delaying and/or stalling the lock holders (e.g., due to adverse operating system's scheduling). On the other hand, non-blocking algorithms use the atomic primitives (e.g. CAS operations) in a more efficient way in order to provide higher progress guarantees, e.g., wait-freedom [20] or obstruction-freedom [23], which are otherwise prevented in the lock-based approaches [14].

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 operations, 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 – often needed for improving performance, functionality, fault-management – difficult. In this paper we focus on composable (transactional) balanced trees.

Although the research has reached an advanced point in the direction of 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) [28], which natively allows composability as it speculates every memory access inside an *atomic* block; *2)* The second approach is Transactional Boosting [21] (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 the 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 *i)* prevents any further customization; and *ii)* 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 Consistency Oblivious Programming (COP) [2–4], Partitioned Transactions (ParT) [33], and Optimistic Transactional Boosting (OTB) [16, 17]. 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 (COP, ParT, and OTB) discusses in detail how they can be applied to balanced trees without nullifying the significant body of work related to highly optimized concurrent (non-transactional) balanced trees. As a result, the programmer can only benefit from the general, yet inefficient, solutions as the only available alternatives so far.

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 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 the OSS approach.

In addition to the new transactional capabilities, TxCF-Tree claims one major innovation with respect to the concurrent 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, already decoupled as in CF-Tree, *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.

In the transactional context, handling the interference between structural and semantic operations (i.e., accessing the same object or meta-data so that an abort or a stall is triggered to preserve correctness) is more costly than in the concurrent context because it could result in aborting the whole transaction, which includes multiple (possibly non-conflicting) semantic operations, and not just one operation as in the concurrent version.

TxCF-Tree uses two new terms which help 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.

Among the other innovations, TxCF-Tree exploits the fact that structural operations are encapsulated in a *helper* thread, which uses an infinite loop to scan the tree until it finds any node that needs to be rotated or physically removed. We propose to further reduce the interference of this *helper* thread by adopting a simple heuristic to detect if the tree is *almost balanced*. If so, we manage to increase the back-off time between two *helper* thread's iterations. Our heuristic uses a simple hill-climbing mechanism that tunes the back-off time according to the number of rotations and physical deletions observed during the previous iterations.

Our implementation of TxCF-Tree is released as an open-source library[1]. We conducted an extensive evaluation study to assess the effectiveness of TxCF-Tree. Our experiments show that TxCF-Tree performs better than the other transactional approaches (TB and STM) in almost all of the cases.

2. Background

2.1 Optimistic Semantic Synchronization

We use the term *Optimistic Semantic Synchronization* (OSS) to represent a set of recent methodologies that leverage the idea of splitting the transaction execution into phases and optimistically executing some of them without any instrumentation (also called *unmonitored* phases). In this section, we briefly recall three of those approaches: Optimistic Transactional Boosting (OTB) [16, 17]; Consistency Oblivious Programming (COP) [2–4]; and Partitioned Transactions (ParT) [33].

OTB methodology 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, 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 [13]. However, opacity in this case is only preserved at the semantic level, 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 is broken at the memory level (which is clear in OTB because of the unmonitored traversal phase). The third guideline of OTB is to optimize the data structure according to its specific characteristics.

COP splits the operations into the same three phases as OTB (but under different names). However, 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 during the commit phase. Instead, it enforces atomicity and isolation by executing both the *validation* and *commit* phases using either STM [2] or HTM [3] transactions.

ParT also uses the same trend of splitting the operations into a *traversal* (they call it *planning*) phase and a *commit* (they call it *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, as a result of being more general, ParT does not give details about handling dependent operations in the same transaction (like OTB's second guideline) and only defines a generic *validator* object that is associated with each *planning*

[1] http://www.hyflow.org/downloads-details.html

phase. Finally, 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.

Summarizing, those three approaches share a common idea: splitting the execution into phases that are executed differently and independently. Hence, TxCF-Tree can be seen as a solution along the same line. However, TxCF-Tree 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 execution of the transaction is *semantically opaque*.

2.2 Contention Friendly Tree

Contention Friendly Tree (CF-Tree) [9] is an efficient concurrent lock-based 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 be executed in a helper thread. Briefly, in the following we report the details of CF-Tree:

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 of the cases, each operation locks at most one node, which is the last node of the traversal phase.

Rotations: re-balancing operations are isolated in a *helper* thread that infinitely 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 [6, 27]. A rotation locks: the node to be rotated down; its parent node; and its left or right child (depending on the type of rotation). Additionally, 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 is marked as *physically removed*, and finally its left and right children are modified to be pointing at its parent. This way, concurrent semantic operations can traverse the tree non-speculatively without being lost.

3. Motivation

This paper is mainly motivated by the lack in literature of an efficient transactional balanced tree, which we believe is highly desirable given the wide diffusion of its concurrent version. We fill this gap by designing TxCF-Tree. As we briefly mentioned before, TxCF-Tree is efficient because it minimizes the interference between semantic operations belonging to a transaction (e.g., insert) and structural operations (e.g., rotations and physical deletions). Before going into the details of the design of TxCF-Tree, in this section we answer two important questions: why we consider CF-Tree as the best candidate, and why we should care about the interference of the structural operations when designing a transactional balanced tree.

3.1 Why CF-Tree as baseline?

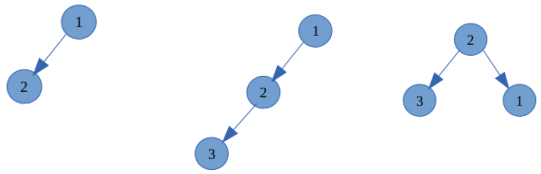
Among the concurrent trees presented in literature, we select CF-Tree as a candidate to be transactionally boosted because it provides the following two properties that fit the OSS principles. First, it uses a lock-based technique for synchronizing the operations, which simplifies the applicability of the OSS methodology. Several approaches are used to design non-blocking trees [7, 25, 30], which provide stronger progress guarantees than CF-Tree. However, making such trees transactional is very complex because each transaction can access many places in the tree and all those accesses are required to be atomic. Historically, lock-based STM algorithms (e.g., [10, 12, 31]) were similarly shown to be more practical than the first obstruction-free STM proposal [24]. Also, contention managers play an important role to guarantee that transactions are executed without deadlocks, and with higher levels of fairness. 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.

The other lock-based solutions [6, 8], as opposed to CF-Tree, have neither a similar unmonitored traversal phase that scans the tree without any instrumentation nor a similar simple commit phase that only changes one flag or attaches one node. In addition, the authors of [9] showed significant performance gains against other recent lock-based tree algorithms.

3.2 On the importance of having interference-less 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. 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.

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.



(a) Initial state (b) Insert 3 (c) Right rotation

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

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 the interference between structural and semantic operations in TxCF-Tree, we propose the following simple guideline:

(G-Pr) Always give the highest priority to semantic operations over structural operations.

This guideline simply allows the semantic operations to proceed when a conflict occurs with structural modifications. Our rationale is that, delaying or aborting a semantic operation affects the overall performance, whereas delaying or aborting a structural operation only defers the step of optimizing the data structure to the near future.

4. TxCF-Tree

In this section, we discuss how to boost CF-Tree to be transactional using the OSS principles. As we mentioned in Section 3, the key additions of TxCF-Tree over CF-Tree are: *i)* supporting transactional accesses; and *ii)* minimizing the interference between semantic and structural operations. For the sake of clarity, 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.

Our TxCF-Tree implements a *set* interface with the traditional semantic operations: `add`; `remove`; and `contains`. Extending TxCF-Tree to have key-value pairs is straightforward. However, 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. TxCF-Tree aims at proposing this innovation as a technique that can also be applied in other solutions, and it does not prevent the usage of other (more complex) heuristics for tuning the back-off delay.

4.1.1 Physical Deletions

The *helper* thread's goal is to physically unlink a node N_n that is marked as *deleted*. To do so, both the N_n and its parent N_p have to be locked. 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 the first CAS operation fails, then the whole structural operation is aborted and the *helper* thread resumes scanning the tree. If the locks are successfully acquired, the node's left and right children are modified to point back to the parent and then N_n is unlinked by changing N_p child to be N_n's child instead of N_n. This way the concurrent operations can still traverse the tree, without experiencing any interruption.

4.1.2 Rotations

Rotations also use a less intrusive locking mechanism. In a right rotation (without losing generality), three nodes are locked in CF-Tree: the parent node N_p, the node to be rotated down N_n, and its left child N_l. In CF-Tree, rotation is done by cloning N_n and linking the cloned node at the tree instead of N_n. Subsequently N_n is marked as `REMOVED` (in case of left-rotation it is marked as `REMOVED-BY-LEFT-ROTATION`). In TxCF-Tree, using the same relaxation as physical deletion, 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*).

While designing how to handle rotations in TxCF-Tree, 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. Semantic operations, on the other hand, 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, so it is safe to make this modification without locking. For the same reason, if all the sub-trees of N_n and N_l are not `null`, then no structural locks are acquired at all, 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 (in this case we use the terminology introduced by OTB [16]). We follow this division in our presentation and discuss each of those three phases separately in the following sections.

4.2.1 Traversal

The tree is traversed similarly to the sequential way, by following the classical rules of the 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*: which is the searched key that needs to be inserted, removed, or looked up.
- *Node*: which is 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 left (right) child is `null` and its item is greater (less) than op-key.
- *Op-type*: which is 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 (through the transaction's 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 an `add` operation is followed by a `remove` operation of the same item (or vice versa), they locally eliminate each other. This elimination saves the useless access to the shared tree in such cases. 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.

4.2.2 Validation

The second phase of TxCF-Tree's operation is the *validation* phase. Validation in TxCF-Tree has two goals. The first goal is the same as the concurrent CF-Tree, which is ensuring that the locked nodes are still the same as they appear at the end of the traversal phase. The second goal is to guarantee the consistency of the overall transaction. 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.

In Algorithm 1, 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).

In TxCF-Tree, this validation procedure is not enough because it has to also ensure that the operation's result is not changed, otherwise, the transaction's consistency is broken. 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 4 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 5 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 6). 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 1 Operation's validation in CF-Tree.

```
1:  procedure VALIDATE(read-set-entry)
2:      if node.removed ¡¿ NOT-REMOVED then
3:          return false
4:      else if entry.node.k = entry.k then
5:          return true
6:      else if entry.node.k ¿ entry.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
```

Algorithm 2 Example of semantic opacity.

```
1:                                                  ▷ initially the tree is empty
2:  @Atomic
3:  procedure T1
4:      if tree.contains(x) = false then
5:          if tree.contains(y) = true then
6:                                                          ▷ hazardous action
7:                                  ▷ like an infinite loop or a division by zero
8:              ...
9:  end procedure
10: procedure T2
11:     tree.add(x)
12:     tree.add(y)
13: end procedure
```

Moreover, if the node is physically removed or its child becomes no longer `null` (which are the invalidation's cases of CF-Tree), this 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 24), and the actual invalidation cases are considered *semantic-invalidations* (lines 14 and 21).

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.

The next step is to validate the entry itself (line 10). If the entry is *semantically-invalidated*, then the transaction aborts (line 18). If it is *structurally-invalidated*, the traversal is continued in the same way as 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

Algorithm 3 Operation's validation in TxCF-Tree.

```
1:  procedure VALIDATE(read-set-entry)
2:                                          ▷ the variable item-existed is true if the item
3:                                          ▷ was in the tree during the operation's return
4:      if entry.op-type ∈ (unsuccessful add, successful remove, successful
        contains) then
5:          item-existed = true
6:      else
7:          item-existed = false
8:      if entry.node.removed ¡¿ NOT-REMOVED then
9:          return STRUCTURALLY-INVALID
10:     else if entry.node.k = entry.k then
11:         if entry.node.deleted XOR item-existed then
12:             return VALID
13:         else
14:             return SEMANTICALLY-INVALID
15:     else if entry.node.k ¿ entry.k then
16:         next = node.right
17:     else
18:         next = node.left
19:     if next = null then
20:         if item-existed then
21:             return SEMANTICALLY-INVALID
22:         else
23:             return VALID
24:     return STRUCTURALLY-INVALID
25: end procedure
```

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 lock.Locked && lockholder ∉ (myID,helperID) then
9:                  return false
10:             result = VALIDATE(entry)
11:             if result = STRUCTURALLY-INVALID then
12:                 newNode = CONTINUE-TRAVERSAL(entry)
13:                 entry.node = newNode
14:                 write-entry = write-set.get(entry.key)
15:                 if write-entry ¡¿ null then
16:                     write-entry.node = newNode
17:             else if result = SEMANTICALLY-INVALID then
18:                 return false
19:             else
20:                 break;
21:     return true
22: end procedure
```

successful *add/remove* operation, the corresponding write-set entry is also updated with the new node (line 16)[2].

4.2.3 Commit

The *commit* phase (shown in Algorithm 5) uses the classical two-phase locking mechanism. The nodes in the read/write sets are locked and/or validated first (lines 5-20), then the tree is modified (lines 23-35), and finally the locks are released (line 37).

From the commit procedure of TxCF-Tree it is worth highlighting the following two 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, in lines

[2] In our implementation, the mapping between write-set and read-set entries is easy because they are implemented using a key-value-based map whose key is the operation's key.

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

31-35 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 Section 4.2.1).

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 (line 16). 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

To ensure the correctness of TxCF-Tree, we prove the following theorem:

THEOREM 1. *A history $\mathcal{H}$ of TxCF-Tree's operations is opaque.*

We leave the proof of this theorem to the technical report [18]. The proof outline is, in some sense, similar to that of OTB-Set [17], the first data structure boosted using OTB methodology. Throughout the whole proof we assumed a history that contains only TxCF-Tree operations (which means that direct reads/writes to the memory as well as accesses that do not use the tree's APIs are prevented). We also assume that each operation contributes only with its return value in $\mathcal{H}$. We leave the general case, where we *i)* allow any kind of accesses in the transaction and *ii)* include the progress guarantees at memory level, as a future work.

6. Evaluation

In our experiments we compared the performance of TxCF-Tree with the performance of TB and some STM approaches (all implemented in Java). Our implementation of TB uses CF-Tree as the underlying (black-box) tree, which makes a fair comparison. For STM, we used Java's Deuce framework [26]. To show that the STM performance issue does not depend on the used protocol, we tested three different algorithms: LSA [31]; TL2 [12]; and NOrec [10], and we experienced almost the same performance for all of them (which is not close to the other non-STM competitors). Thus, to make the plots clear, we only show the curve corresponding to the best STM algorithm in each plot.

All the experiments were conducted on a 64-core machine, which has 4 AMD Opteron (TM) Processors, each with 16 cores running at 1400 MHz, 32 GB of memory, and 16KB L1 data cache. Throughput is measured as the number of semantic operations (not transactions) per second to have consistent data points. Each plotted data point is the average of five runs. Each run starts with a warm-up phase of five seconds, followed by an execution phase during which we collect the statistics.

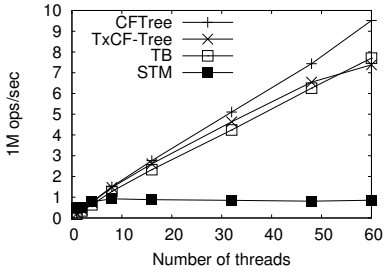


Figure 2. Throughput of tree-based set with 10K elements, 50% `add/remove` operations, and one operation per transaction.

In Figure 2 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). In this experiment, all of the overheads added by the transactional algorithms (i.e. STM, TB, and TxCF-Tree) are not actually exploited because there is only one operation enclosed in the executed transactions, thus we leverage this plot to show the cost of having 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 any transactional semantics.

Figure 3 shows 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%, 50% and 80% of `add/remove` operations). The plots 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 `add/remove` operations. This is reasonable because, the conflict level becomes higher, and it best fits the more *pessimistic* approach (as TB). However, even in the most conflicting case (having 80% `add/remove` operations) TxCF-Tree still performs better (except for the last two data points in Figure 3(i)) than 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 reason for this behavior is that, in the 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 3(g) and Figure 3(i) (which differ only for the size of the tree) confirms this claim. In Figure 3(g), both algorithms scale well up to 16 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 3(i) 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 improvement may increase if a more effective heuristic is used.

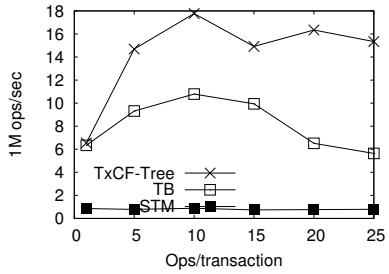


Figure 4. Throughput of tree-based set with 10K elements, 50% `add/remove` operations, and 32 threads.

In Figure 4 we report the behavior of TxCF-Tree's while changing the size of the transactions. We can observe a significant gap between TxCF-Tree and TB for all of the tested sizes, which confirms our conclusion: reducing operations' interference is important in order to avoid unnecessary aborts.

The last experiment we report regards the capability of TxCF-Tree to reduce interferences with structural operations. Although

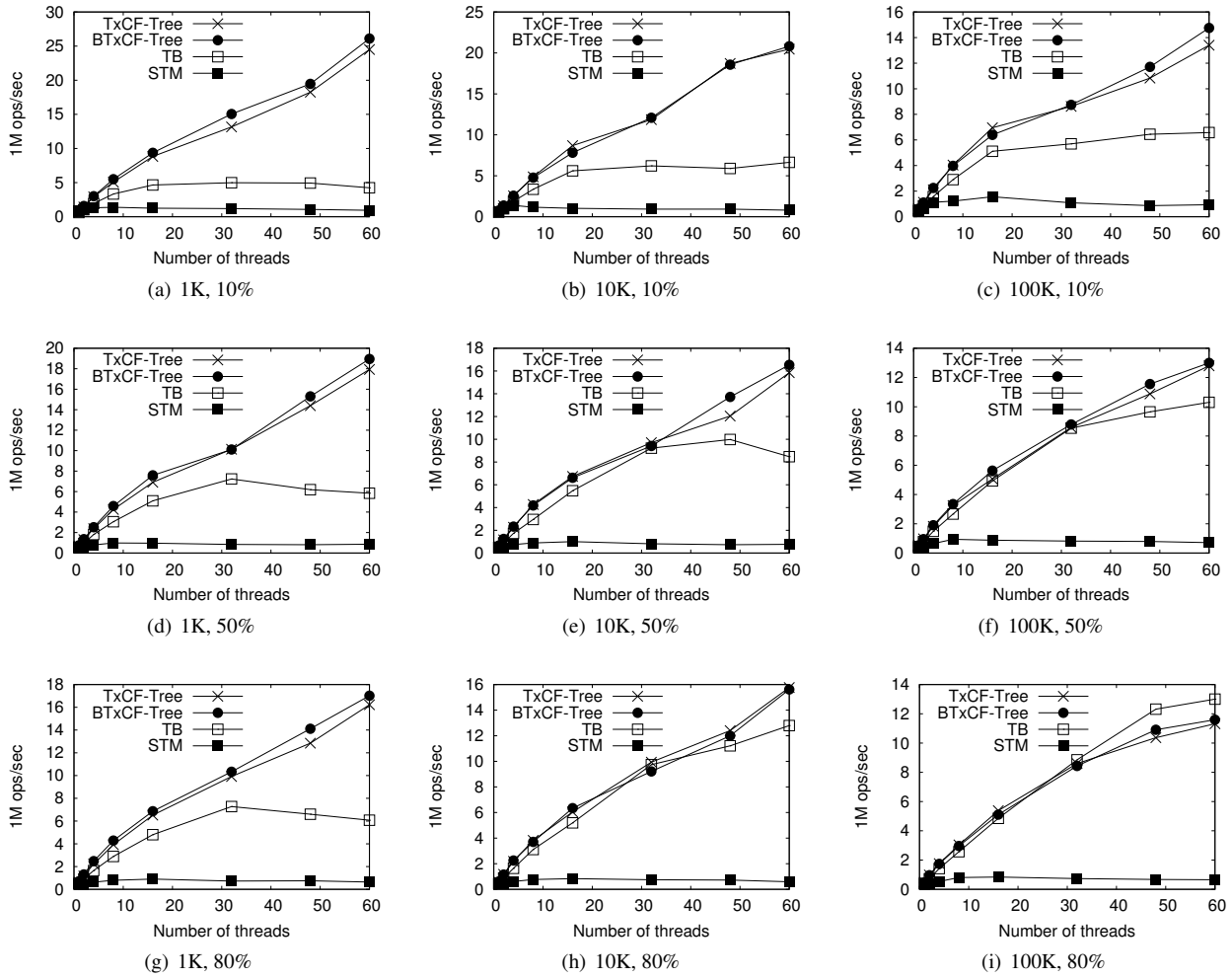


Figure 3. Throughput of tree-based set with five operations per transaction (labels indicate the size of the tree and the % of the `add/remove` operations).

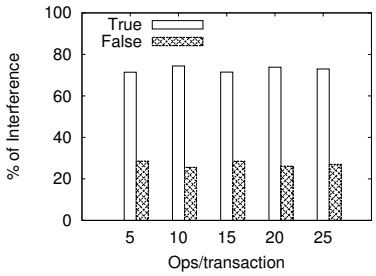


Figure 5. The percentage of the two interference types on a tree-based set with 10K elements, 50% `add/remove` operations, and 32 threads.

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 5 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 justify that the optimized design of TxCF-Tree allows it to perform better than the general approaches.

Acknowledgments

Authors thank Vincent Gramoli for sharing the source code of CF-Tree (which is now public) in an early stage of the development.

References

[1] G. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences*, volume 145, pages 263–266, 1963.

[2] Y. Afek, H. Avni, and N. Shavit. Towards consistency oblivious programming. In *OPODIS'11*, pages 65–79.

[3] H. Avni and B. C. Kuszmaul. Improving HTM scaling with consistency-oblivious programming. In *TRANSACT*, 2014.

[4] H. Avni and A. Suissa-Peleg. Brief announcement: Cop composition using transaction suspension in the compiler. In *DISC*, pages 550–552, 2014.

[5] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta informatica*, 1(4):290–306, 1972.

[6] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *PPoPP*, pages 257–268, 2010.

[7] T. Brown, F. Ellen, and E. Ruppert. A general technique for non-blocking trees. In *PPoPP*, pages 329–342, 2014.

[8] T. Crain, V. Gramoli, and M. Raynal. A speculation-friendly binary search tree. In *PPoPP*, pages 161–170, 2012.

[9] T. Crain, V. Gramoli, and M. Raynal. A contention-friendly binary search tree. In *Euro-Par*, pages 229–240, 2013.

[10] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *PPoPP*, pages 67–78, 2010.

[11] M. David. A single-enqueuer wait-free queue implementation. In *DISC*, pages 132–143, 2004.

[12] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, pages 194–208, 2006.

[13] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP'08*, pages 175–184.

[14] R. Guerraoui and M. Kapalka. The semantics of progress in lock-based transactional memory. In *POPL*, pages 404–415, 2009.

[15] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC'11*, pages 300–314.

[16] A. Hassan, R. Palmieri, and B. Ravindran. Optimistic transactional boosting. In *PPoPP'14*, pages 387–388.

[17] A. Hassan, R. Palmieri, and B. Ravindran. On developing optimistic transactional lazy set. In *OPODIS*, pages 437–452, 2014.

[18] A. Hassan, R. Palmieri, and B. Ravindran. Transactional interference-less balanced tree. Technical report, ECE Dept., Virginia Tech, January 2015. `www.hyflow.org/pubs/spaa15-hassan-TR.pdf`.

[19] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *OPODIS'05*, pages 3–16.

[20] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[21] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *PPoPP*, pages 207–216, 2008.

[22] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[23] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pages 522–529, 2003.

[24] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, 2003.

[25] J. H. Kim, H. Cameron, and P. Graham. Lock-free red-black trees using cas. *CCPE*, pages 1–40, 2006.

[26] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In *MULTIPROG*, 2010.

[27] K. S. Larsen. AVL trees with relaxed balance. In *IPPS*, pages 888–893, 1994.

[28] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan and Claypool, 2006.

[29] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82, 2002.

[30] A. Natarajan, L. Savoie, and N. Mittal. Concurrent wait-free red black trees. In *SSS*, pages 45–60, 2013.

[31] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *DISC*, pages 284–298, 2006.

[32] N. Shavit. Data structures in the multicore age. *Commun. ACM*, 54 (3):76–84, 2011.

[33] L. Xiang and M. L. Scott. Composable partitioned transactions. In *WTTM*, 2013.