RCU-HTM: Combining RCU with HTM to Implement Highly Efficient Concurrent Binary Search Trees

Dimitrios Siakavaras, Konstantinos Nikas, Georgios Goumas and Nectarios Koziris
National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory
{jimsiak,knikas,goumas,nkoziris}@cslab.ece.ntua.gr

***Abstract*—In this paper we introduce *RCU-HTM*, a technique that combines Read-Copy-Update (RCU) with Hardware Transactional Memory (HTM) to implement highly efficient concurrent Binary Search Trees (BSTs). Similarly to RCU-based algorithms, we perform the modifications of the tree structure in private copies of the affected parts of the tree rather than in-place. This allows threads that traverse the tree to proceed without any synchronization and without being affected by concurrent modifications. The novelty of *RCU-HTM* lies at leveraging HTM to permit multiple updating threads to execute concurrently. After appropriately modifying the private copy, we execute an HTM transaction, which atomically validates that all the affected parts of the tree have remained unchanged since they've been read and, only if this validation is successful, installs the copy in the tree structure.**

We apply *RCU-HTM* on AVL and Red-Black balanced BSTs and compare their performance to state-of-the-art lock-based, non-blocking, RCU- and HTM-based BSTs. Our experimental evaluation reveals that BSTs implemented with *RCU-HTM* achieve high performance, not only for read-only operations, but also for update operations. More specifically, our evaluation includes a diverse range of tree sizes and operation workloads and reveals that BSTs based on *RCU-HTM* outperform other alternatives by more than 18%, on average, on a multi-core server with 44 hardware threads.

***Index Terms*—Concurrent Data Structures; Binary Search Trees; HTM; RCU;**

I. Introduction

With the dominance of multi-core architectures concurrent data structures have become a crucial component of many multi-threaded applications. In order to benefit from the rapidly increasing number of cores provided by every new processor generation, programmers need to carefully design, implement and tune these data structures. To this direction, *performance* is typically the factor that drives most of the design decisions. In concurrent data structures, performance is sought in three directions: (i) by exposing as much parallelism as possible (e.g., by allowing threads that access disjoint parts of the data structure to execute concurrently), (ii) by minimizing the extra work imposed by the parallelization scheme, and (iii) by minimizing the synchronization overheads.

Beyond performance, three more factors are additionally taken into consideration: *Robustness;* i.e., the ability of a concurrent data structure to react to unexpected thread actions (e.g., thread failure, scheduler decisions). A classic example is when a thread holding a lock is scheduled out and other threads can make no progress until the lock is released. Even worse is the case when a thread fails unexpectedly. The more locks a data structure requires, the more complex it is to handle such cases. *Programmability;* i.e., the effort required to implement a concurrent data structure. Efficient concurrent data structures are usually difficult to implement. Complex algorithms and synchronization patterns typically lead to complex and error-prone implementations that additionally cannot be reproduced for alternative problems in a straightforward way. *Memory requirements;* i.e., the size of the memory footprint of a concurrent data structure. Besides size, in concurrent configurations it is important to ensure that freed memory is not being accessed by other threads. In this work our focus is on performance, however, we also briefly discuss how our approach affects the other three factors.

We work on concurrent binary search trees (BSTs), a family of data structures that is widely used in a diverse set of applications. More specifically, we use BSTs to implement the dictionary abstract data type that stores a set of key-value pairs and supports three operations: `lookup(key)`, `insert(key,value)` and `delete(key)`. We identify the following characteristics to be of utmost importance in order to implement a highly efficient concurrent BST:

Balance: Maintaining the height of the tree almost balanced is crucial for performance, as highly unbalanced trees may lead to longer path traversals. There are several balanced binary search trees (BBST) like Red-Black [14], left-leaning Red-Black [26], AVL [1], radix trees [22] and B-trees [5]. Concurrent BBSTs are more challenging than BSTs because the rebalancing operations may modify multiple parts of the tree, making it hard to correctly synchronize them.

Effective Node organization: Internal BSTs store key-value pairs in every node of the tree. While they are efficient with respect to memory requirements, they have the disadvantage that the deletion of a node with two children is complex. External trees overcome this complexity by storing key-value pairs on leaf nodes, i.e., nodes with no children. Non-leaf nodes contain only keys and are used for routing to the appropriate leaves. External trees simplify the deletion of a key-value pair; however, they have two disadvantages: first, all traversals end at a leaf node, which leads to longer traversal paths, and, second, they occupy twice as much memory as the internal ones.

On-time deletion: An alternative way to avoid the complexity of deleting a node with two children is lazy deletion, i.e., to mark a node as deleted without physically removing it from the tree. Several lock-based and lock-free BSTs adopt this technique; however, it has the side effect of leaving dummy nodes in the tree, resulting in longer traversal paths. On the other hand, with on-time deletion, i.e., physically removing the node from the tree at the time it is logically removed, traversals don't encounter dummy nodes.

Asynchronized traversals: Of utmost importance for the performance of a BST is the efficiency of traversals [17], [11]. This is the case for two reasons: first, all three supported operations start with traversing the tree, and second, lookup is typically the most common operation. It is thus critical to avoid synchronization during traversals, as it induces high overheads.

Minimal synchronization overhead for updates: Although the performance of traversals is the most important, it is also desired to minimize the synchronization required for updates [17], [11]. Updates that modify disjoint parts of the tree should be allowed to execute concurrently. Moreover, for a concurrent implementation to be robust, locks should better be avoided or the number of necessary lock acquisitions per operation be kept minimal.

While the above characteristics are highly beneficial in a concurrent BST, as validated also by our experimental evaluation, combining all of them in a single implementation is a challenging task. Several recent related works have implemented BSTs with various combinations of these characteristics [2], [4], [6], [7], [8], [9], [10], [12], [13], [19], [20], [25]; however, none of them combines all five characteristics.

Non-blocking BSTs [8], [13], [20], [25] are either totally unbalanced or partially balanced [7]. This is due to the difficulty of mapping the multiple modifications performed during rebalance to several distinct atomic modifications that should be executed using atomic operations (e.g., CAS). Moreover, non-blocking trees are extremelly hard to implement. Lock-based BSTs [6], [10], [12] are also partially balanced because the rebalancing of the tree requires several locks to be acquired. Additionally, during rebalancing, threads may try to acquire locks in opposite directions, making it hard to avoid deadlocks.

BSTs based on Read-Copy-Update (RCU) [24], [19], [2] provide asynchronized traversals; however, they either prohibit multiple writers using a single global lock [19], or permit multiple writers by using fine-grain locking at the expense of not rebalancing the tree [2]. BSTs that exploit Transactional Memory (TM) [16] allow updates on different parts of the tree to be executed concurrently but traversals still need to synchronize with concurrent modifications. Recent TM-based BSTs fail to provide strictly balanced trees [9] and asynchronized traversals [9], [28], [27], [4].

In this work we introduce *RCU-HTM*, a technique that combines RCU with HTM and manages to get the best of both worlds. From RCU, we adopt the following technique that enables asynchronized read-only operations: we perform the modifications of the tree structure in private copies of the affected parts of the tree rather than in-place. This allows threads that traverse the tree to proceed without any synchronization and without being affected by concurrent modifications. What distinquishes *RCU-HTM* from previous RCU-only BSTs is the exploitation of HTM to permit multiple updating threads to execute concurrently. More specifically, we use HTM in the following way: prior to installing the modified private copy in the shared tree, we atomically validate that all the nodes to be replaced have remained unchanged since they were read. Both the validation and the installation step are enclosed in a single HTM transaction and are thus performed in a single atomic step. If validation fails due to some of the nodes to be replaced having been modified by other threads, we restart the operation.

RCU-HTM is applicable to all types of BBSTs, as well as to unbalanced BSTs. We implement and evaluate two *RCU-HTM* based BBSTs, a Red-Black and an AVL tree. To the best of our knowledge, these are the first BBSTs to combine all five aforementioned desired characteristics of a concurrent BST implementation. We compare the performance of *RCU-HTM* based BBSTs to two state-of-the-art lock-based [6] and non-blocking [25] trees, as well as RCU- [19], [2] and HTM-based [4] BSTs. As our experimental evaluation reveals, *RCU-HTM* based BSTs achieve high performance, not only for read-only operations, but also for updaters, i.e., threads that modify the tree structure. More specifically, our experiments on a multi-core server with 44 hardware threads using a diverse range of tree sizes and operation workloads indicate that *RCU-HTM* based BSTs are 18% faster, on average, than other alternatives.

II. Background

A. Balanced Binary Search Trees

BBSTs are BSTs in which the difference between the shortest and the longest path is kept bounded. By maintaining the path lengths under certain limits, BBSTs provide efficient traversals. Several BBSTs have been proposed including, but not limited to, Red-Black [14], left-leaning Red-Black [26], AVL [1], radix trees [22] and B-trees [5].

In our work we use BBSTs to implement the *dictionary* abstract data type that stores key-value pairs and supports three operations: (a) *lookup(key)* searches whether the given key exists in the dictionary and returns true if it is found, (b) *insert(key, value)* inserts a key-value pair in the dictionary and returns true if it is not already in the tree, otherwise it returns false, and (c) *delete(key)* removes a key-value pair and returns true if the key is in the tree, otherwise it returns false.

Each BBST imposes different balancing criteria and rebalancing actions. However, the logic is the same: when adding/removing a node to/from the tree, the balance criteria may be broken and a series of node modifications needs to be performed to restore the balance of the tree. For example, in Red-Black trees these modifications include node color changes and rotations, while in AVL trees node height changes and rotations. The generic structure of an insert procedure in a BBST is presented in Figure 1. The first step is a traversal

```
1 int bbst_insert(bbst_t *bbst, int key,
                  void *value)
2 {
3   traverse_bbst(bbst,key);
4   if (key was found) return 0;
5   insert_node_and_rebalance(bbst,key,value);
6   return 1;
7 }
```

Figure 1: A sketch of an insert operation in a BBST. The only difference between various BBSTs is the way in which the rebalancing is performed.

of the tree to locate the point where the new node is to be added. The set of traversed nodes is called the *access path*. The second step is to insert the new node and, if necessary, restore the balancing conditions. The rebalancing step may include several node modifications on or near the access path.

B. Read-Copy-Update

Read-Copy-Update (RCU) [24] is a synchronization pattern in which updaters first read and copy the parts of the data structure they will modify and, after modifying their private copy, replace the old version of the data structure with their new modified version. This replacement is performed in a single atomic step, so that other threads observe either the old or the new version. This way, RCU enables read-only operations to proceed without synchronization. However, updaters still have to be synchronized, otherwise one thread may discard the modifications performed by another one.

C. Hardware Transactional Memory

Intel provides HTM support in commercial processors with the introduction of Transactional Synchronization Extensions (TSX) [1] in Haswell processors and all their successors. TSX is a set of assembly instructions that are used by the programmer to enclose critical sections of code which need to be executed atomically. These critical sections are executed as *transactions* whose memory reads and writes are being tracked by the underlying HTM system. The read memory locations are kept in the transaction's *read-set* and the written ones in the *write-set*. If the read- and write-sets do not conflict with memory accesses from other threads, the transaction commits. Otherwise, the transaction aborts and none of its memory writes become visible to other threads. Moreover, a transaction may suffer from capacity aborts, due to the bounded size of the hardware buffers that store the read- and write-sets. When a transaction aborts, either a lock is acquired or the programmer decides what should be done, depending on the execution mode of TSX. Apart from conflict and capacity aborts, a TSX transaction may fail for other reasons such as cache line eviction, interrupt and/or unsupported instructions.

TSX can be used in two modes, namely HLE and RTM. We only use RTM because it provides the flexibility to choose what actions are taken upon a transaction's abort.

[1]https://en.wikipedia.org/wiki/Transactional_Synchronization_Extensions

TSX is a *best-effort* HTM implementation and provides no guarantees that any transaction will eventually commit; persistent aborts may lead to livelock. It is thus the programmer's responsibility, when using RTM, to provide an alternative path of execution that uses no transactions, i.e., a *non-transactional fallback path*. The most common practice is to retry a transaction for a given amount of times and, if it fails to commit, fallback to the acquisition of a lock that allows only a single thread to enter the critical section.

III. Related Work

In this section we provide an overview of related work on concurrent BSTs with respect to the type of synchronization used and compare them on the basis of the five desired concurrent BST characteristics, discussed in Section I. Table I presents an overview of concurrent BSTs.

Non-blocking BSTs: Non-blocking BSTs use hardware atomic operations such as Compare-and-Swap (CAS) to synchronize accesses to the tree [7], [8], [13], [20], [25]. Using CAS, only a single memory location can be atomically modified. The rebalance phase of BBSTs, on the other hand, requires several memory locations to be updated in an atomic step. This limitation of CAS instructions causes the majority of non-blocking BSTs to be unbalanced [8], [13], [20], [25]. However, even these unbalanced BSTs are extremely hard to design and implement. The same CAS restriction is the reason why some non-blocking BSTs use external trees [13], [25] and others do not support on-time deletion of nodes [8], [20]. Finally, in some cases [8], [20] traversals may need to restart as they have to help other pending operations.

Lock-based BSTs: Locks have made it possible to build relaxed balance BSTs [6], [10], [12]. Typically, these relaxed implementations do not allow the height of the tree to become as long as in the completely unbalanced trees. However, they too fail to provide the logarithmic limits of strictly balanced BSTs when updates are executing. The contention-friendly AVL tree proposed in [10] performs the rebalancing steps in a dedicated thread which makes the situation even worse as a balance violation may not be restored for a long time, especially in large trees. Finally, most of the lock-based BSTs fail to provide asynchronized traversals.

RCU-based BSTs: RCU enables asynchronized traversals but updaters need to be synchronized. Current RCU-based BSTs [2], [19] sacrifice either updaters' concurrency to provide strict balance [19], or the balance of the tree in order to facilitate the synchronization among updaters using fine-grained per node locks [2]. Alternative RCU schemes have been proposed [3], [23] which improve the performance of RCU-based BSTs. However, these trees are also unbalanced due to the difficulty of synchronizing updaters.

TM-based BSTs: TM enables the modification of several memory locations in a single atomic step. This allows the rebalance phase of a BBST to be performed atomically in its entirety. HTM-based BST by Avni et al. [4] is strictly balanced; however, traversals also use HTM transactions and may restart. Crain et al. [9] use STM in their relaxed AVL tree

	Name	Type	Balanced	Internal	On-time deletion	Asynchronized traversals	Synchronization of updaters
Non-blocking	Ellen [13]	BST	No	No	Yes	Yes	CAS
	Howley [20]	BST	No	Yes	No	No	CAS
	Natarajan [25]	BST	No	No	Yes	Yes	CAS
	Chatterjee [8]	BST	No	Yes	No	No	CAS
	Brown [7]	RBT	Relaxed	No	Yes	Yes	LLX/SCX/VLX (CAS-based primitives)
Locks	Bronson [6]	AVL	Relaxed	No	No	No	fg locks
	Crain-lb [10]	AVL	Relaxed	No	No	Yes	fg locks
	Drachsler [12]	BST/AVL	Relaxed	Yes	Yes	Yes	fg locks
RCU	Howard [19]	RBT	Yes	Yes	No	Yes	RCU (single updater)
	Arbel [2]	BST	No	Yes	No	Yes	RCU (multiple updaters with fg locks)
TM	Crain-tm [9]	AVL	Relaxed	Yes	No	No	STM
	Avni [4]	All	Yes	Yes	Yes	No	HTM
	RCU-HTM	All	Yes	Yes	Yes	Yes	RCU+HTM

TABLE I: Comparison of concurrent BSTs. The term "asynchronized traversals" refers to traversals that neither use any synchronization nor is there any possibility to restart.

but they split the rebalance phase in multiple atomic steps in order to minimize contention among transactions. They also need to synchronize traversals.

Wang et al. [29] use HTM to implement a concurrent B+-tree, which is employed in an OLTP database system. Contrarily to our approach, they use HTM in a straightforward way, i.e., they enclose each tree operation, including lookup, in a single HTM transaction. As previous research suggests [27], this is far from the most efficient way to use HTM for parallelizing the operations of BSTs due to the large size of the transactions. They also use HTM to ensure the atomicity of their OLTP transactions. They apply Optimistic-Concurrency-Control (OCC) [21] which differs from our RCU-based approach in two ways: (i) OCC does not straightforwardly support asynchronized lookups; extra programming effort and orchestration of readers and updaters is required. (ii) If OCC was applied in BBSTs instead of RCU, the writes performed during rebalancing would be included inside the HTM transactions, while in the case of RCU only a single write is performed in each transaction.

IV. RCU-HTM Algorithm

As shown in Table I, state-of-the-art concurrent BSTs fail to combine the five characteristics that we define for a highly efficient implementation. Trees implemented with *RCU-HTM* are the first to combine all of them, and, as our experimental evaluation reveals, this translates to performance benefits.

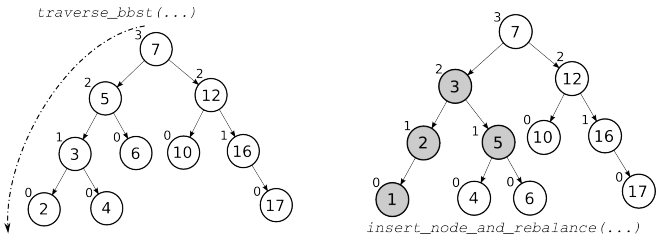
A. General Concept

RCU-HTM combines RCU and HTM in order to pursue two goals: (i) allow read-only operations to proceed without synchronization and restarting, and (ii) allow multiple updaters to modify different parts of the data structure concurrently.

The first goal is a direct outcome of applying RCU in BBSTs. Figure 2 presents an example of inserting key 1 in a BBST using the function of Figure 1. The traversal phase locates the point where the new node will be added (Figure 2a). The second phase inserts the new node in its position and rebalances the tree (Figure 2b). Intuitively, the modifications that are performed during the call of `insert_node_and_rebalance()` can be regarded as replacing a set of nodes W with a modified set W'.

In *RCU-HTM*, a thread that needs to modify a part of the tree, creates a private copy of this part, modifies it accordingly, and, finally, installs its modified private copy by swapping the child pointer of a single tree node. We call this node the *connection point*. These steps are shown in Figure 3 for the insertion of key 1. In this example the set W consists of nodes with keys 2, 3 and 5 and the connection point is node with key 7. By using private copies the entire set of modifications becomes visible to other threads atomically at the time when the child pointer of node 7 is swapped.

By exploiting RCU, *RCU-HTM* allows read-only operations to proceed without synchronization or restarts. However, the novelty of *RCU-HTM* lies at the way we use HTM to enable multiple concurrent updaters. In this case, the operation gets complicated, because there may exist cases where nodes in W have been modified by another thread before they have been replaced by W'. This leads to modifications being discarded. To avoid such erroneous executions, we add a validation step just before installing the modified copy in the tree. More specifically, when updaters copy nodes of the tree, they also track down the state of these nodes and, before they install their modified copy, they validate that all the nodes in W have remained in the same state. If any of these nodes has changed, we restart the operation. The validation and installation of the private copy need to be performed atomically, therefore we execute these two steps inside an HTM transaction.



(a) Insertion phase 1: Traversal of the tree.

(b) Insertion phase 2: Insertion of new node and rebalance the tree.

Figure 2: The two phases when inserting key 1 in an AVL tree. Nodes in gray have been modified during phase 2.

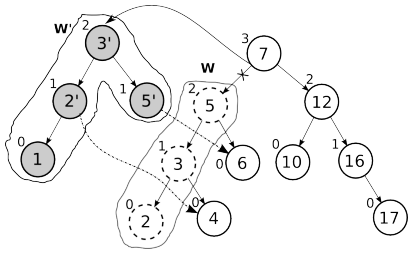


Figure 3: Performing the modifications in copies rather than in-place.

B. Implementation Details

Figures 5-10 present the code of our *RCU-HTM* based AVL tree. Due to space limitations we provide the code only for lookup and insert operations but both our AVL and Red-Black tree implementations are publicly available in https://github.com/rcu-htm/rcu-htm.

In the code we use the macros `TX_BEGIN`, `TX_END` and `TX_ABORT` which are wrappers of the corresponding TSX assembly instructions to begin, commit and abort a transaction.

Figure 5 presents the structures and helper functions used by our AVL tree. The structure of the tree node (lines 8–14) is the classic AVL tree node; apart from the key-value pair it stores the height of the node and pointers to its two children. To represent the avl tree we use the `avl_t` structure (lines 16–19), where we maintain a pointer to the root node of the tree and a lock which is acquired by updaters in the non-transactional fallback path. The helper functions `height()` and `balance()` return the height and balance of a node, and `rotate_l()` performs a left rotation over the given node. The respective one for right rotation is very similar so we omit it. Finally, `copy_node()` creates a copy of a node.

Lookup. The lookup operation of our *RCU-HTM* based AVL is given in Figure 6 and is identical to serial AVL trees, as there is no need for synchronization or restarting. Thanks to the fact that updaters work on private copies, lookups can safely traverse the tree and always access consistent parts of the tree without ever being obstructed or delayed by concurrent insert and/or delete operations.

Insert. Figure 7 presents the function that inserts a new key-value pair in our AVL tree. The common execution path of insertion starts with a traversal (line 90) which, as is the case for lookups, uses no synchronization or restarts. This is performed in function `traverse_with_stack()`. The only difference from lookups is that, while moving downwards, we also keep track of the traversed nodes by storing pointers to them in a stack. This stack is later used for the reverse traversal of the tree in the rebalance phase. If the key is found in the tree, the function immediately returns (lines 91–92).

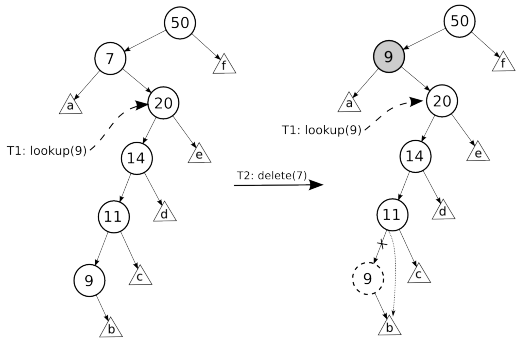
If the key is not found, we call `insert_node_and_rebalance()` (line 94, Figure 7) that performs the insertion of the new node and the rebalance dictated by the AVL tree rules. However, instead of directly modifying the affected nodes, it first copies them locally. Upon completion, it returns a pointer to the connection point (`conn_point` parameter) and a pointer to the root of its private modified copy (`tree_cp_root` parameter). Moreover, while copying nodes, in `insert_node_and_rebalance()` we maintain a hash table where we store the children pointers of the copied nodes. This information is used in the validation phase.

When `insert_node_and_rebalance()` returns, we proceed with the validation of the private copy and its installation in the shared tree. If the validation phase has already been retried for a configurable number of times, we restart the operation (lines 98–99). The while loop in line 101 is used to avoid starting a transaction when the lock is taken. The if statement in lines 103–115 checks whether a transaction has just began or has been aborted. In the former case, we check whether the lock is taken (lines 104–105) and if so we explicitly abort the transaction. By reading the lock variable we guarantee that when another thread acquires the lock, all concurrent transactions are aborted (due to conflict on the lock variable). Next, we validate the modified copy, install it in the shared tree and commit the transaction (lines 107–109).

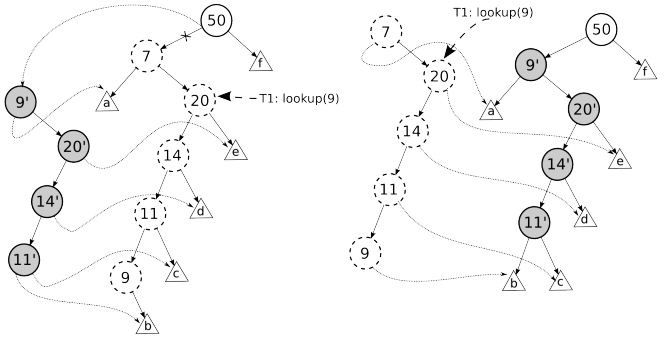
When a transaction aborts, lines 111–114 are executed. Depending on the abort reason we perform the following actions: if the transaction aborted due to failure in validating the copy the operation must be restarted; otherwise, we can safely retry the validation and installation step.

If the operation has been retried `TX_NUM_RETRIES` times, the non-transactional fallback path (Lines 77–88) is executed. By acquiring the lock, we guarantee that only a single updater is active (readers are not affected) so there is no need to perform the validation step prior to installing the modified copy in the tree.

Delete. The *delete* operation in *RCU-HTM* is performed in a way similar to insertion when the node n to be removed has less than two children. However, when n has two children, its removal is more complicated. In this case we need to locate the successor node s of n (i.e., the node containing the smallest key that is larger than n's key), replace n's key with s's and finally remove s from the tree. An example of deleting a node with two children is depicted in Figure 4a. Key 7 is replaced by the successor's key, which in this case is key 9, and the successor node is removed from the tree. When moving the successor's key to the position of the deleted key, traversing threads searching for that key may incorrectly fail to find it



(a) Inconsistency created when thread T1 searches for key 9 which is moved up in the tree by a concurrent deletion. T1 fails to find key 9 albeit it is present in the tree.



(b) Avoiding incosistency with *RCU-HTM* by replacing the path between nodes 7 and 9 to allow thread T1 to find key 9 in its previous position.

Figure 4: Incosistency due to concurrent delete and lookup.

in its old position. Such an erroneous execution is shown in Figure 4a, where thread T1 is searching for key 9. T1 is not notified of the repositioning of key 9 and fails to locate it.

To avoid such problematic cases in *RCU-HTM*, when a node with two children is to be removed, we copy the whole path from the node that contains the key to be deleted to the successor node. If the copies performed during the rebalance phase already include all the nodes in this path, no further action needs to be performed. Figure 4b presents an example of deleting a node with two children using *RCU-HTM*. In this case, T1 is navigated to the node with key 9 even though in the new version of the tree key 9 is located higher in the tree.

V. Correctness

In this section we provide an informal correctness proof for *RCU-HTM*. To do so, we use the well-known correctness condition of linearizability [18]. For each of the three operations we define the linearization point, i.e., the point at which the operation has taken effect. We omit the analysis of the linearization point of the delete operation, as it can be defined in a way similar to that of the insert operation.

Lookup: The reasoning about the linearizability of lookup in *RCU-HTM* is identical to other RCU-based implementations [2], [19]. In all RCU-based algorithms, including *RCU-HTM*, updaters commit their copies by modifying one memory location. In *RCU-HTM* this is the appropriate child pointer of the connection point. Since single-word reads and writes are atomic, readers observe either the old or the new version of the data structure. Moreover, because *RCU-HTM* avoids performing any rotations directly on the tree, traversals never follow a wrong path. In the following paragraph, we provide the linearization points of lookup.

A lookup operation that observes an empty tree is linearized at line 56 when `avl->root` is read as `NULL`. When the tree is not empty, the linearization point is either at line 62 or line 63 when the appropriate child of `leaf` is read. There is a time window between the read of `leaf->left` or `leaf->right` and the point at which the lookup operation returns, during which `leaf` may have been removed from the tree by some concurrent insertion or deletion. Even then, however, the lookup can safely be linearized before this operation.

Insert: An insert operation that finds the key in the tree and does not modify the tree structure is linearized similarly to a lookup operation, i.e., at the point when the node containing the searched key is reached. That is, either line 133 or line 134 of `traverse_with_stack()`. An insert operation that does not find the key and modifies the tree is linearized in one of two points; either at line 109 when a hardware transaction commits installing the private copy of the operation in the tree, or at line 78 when `avl->avl_lock` is acquired.

VI. Experimental Evaluation

For our experiments we used an Intel Broadwell-EP server with an Intel Xeon E5-2699 v4 processor with 22 physical cores and 44 hardware threads. The processor runs at 2.2GHz and each physical core has its own L1 and L2 cache of sizes 32KB and 256KB respectively. A 56MB L3 cache shared by all cores is also available and the server is equipped with 64GB of RAM. The system runs Debian 8.3 with kernel version 4.7.0. For the compilation of all our implementations we used GCC 4.9.2 with the -O3 optimization flag enabled.

Our evaluation includes the following concurrent BSTs:

- **lb-avl**[2]: A relaxed, partially external, lock-based AVL tree by Bronson et al. [6].
- **lf-bst**[2]: An unbalanced, external, non-blocking BST by Natarajan et al. [25].
- **citrus-bst**: The Citrus unbalanced, internal BST by Arbel et al. [2] that uses RCU and fine-grain locks to synchronize updaters.
- **rcu-mrsw-avl**: A multi-reader-single-writer RCU-based AVL tree [19] that synchronizes updaters with a global lock.
- **cop-avl**: The consistency-oblivious-programming based AVL tree by Avni et al. [4].
- **rcu-htm-avl**: An *RCU-HTM* based internal AVL tree.
- **rcu-htm-rbt**: An *RCU-HTM* based internal Red-Black tree.

We evaluate the concurrent BSTs in the following way:

- Each run lasts 2 seconds, during which each thread performs randomly chosen operations. During this time threads perform enough operations to get stable results. We also tried longer durations and got similar results.

[2]We use the implementation from ASCYLIB [11].

```
typedef struct avl_node_s {
  int key;
  void *value;
  int height;
  struct avl_node_s *left,
                    *right;
} avl_node_t;

typedef struct {
  avl_node_t *root;
  pthread_spinlock_t avl_lock;
} avl_t;

int height(avl_node_t *n) {
  if (!n) return -1;
  else return n->height;
}

int balance(avl_node_t *n) {
  if (n == NULL)
    return 0;
  else
    return (height(n->left) -
            height(n->right));
}

avl_node_t *rotate_l(avl_node_t *n) {
  avl_node_t *l = n->left;
  n->left = n->left->right;
  l->right = n;
  n->height = MAX(height(n->left),
                  height(n->right)) + 1;
  l->height = MAX(height(l->left),
                  height(l->right)) + 1;
  return l;
}

avl_node_t *copy_node(avl_node_t *src) {
  avl_node_t *dst;
  dst = new_node(src->key, src->value);
  dst->height = src->height;
  dst->left = src->left;
  dst->right = src->right;
  return dst;
}
```

Figure 5: The structures and helper functions used by our AVL tree.

```
int avl_lookup(avl_t *avl, int key)
{
  avl_node_t *parent, *leaf;

  parent = NULL;
  leaf = avl->root;

  while (leaf) {
    if (leaf->key == key) return 1;

    parent = leaf;
    if (key < leaf->key) leaf = leaf->left;
    else                 leaf = leaf->right;
  }

  return 0;
}
```

Figure 6: Lookup operation.

```
int avl_insert(avl_t *avl, int key,
               void *value)
{
  avl_node_t *node_stack[MAX_HEIGHT];
  avl_node_t *tree_cp_root, *conn_point;
  int stack_top;
  hash_table_t ht;

  int retries = -1;
RETRY:
  if (++retries >= TX_NUM_RETRIES) {
    pthread_spin_lock(&avl->avl_lock);
    traverse_with_stack(...);
    if (stack_top >= 0 &&
        node_stack[stack_top]->key == key) {
      pthread_spin_unlock(&avl->avl_lock);
      return 0;
    }
    insert_node_and_rebalance(...);
    install_copy(...);
    pthread_spin_unlock(&avl->avl_lock);
    return 1;
  }

  traverse_with_stack(avl, key, node_stack,
                      &stack_top);
  if (stack_top >= 0 &&
      node_stack[stack_top]->key == key)
    return 0;

  insert_node_and_rebalance(key, value,
              node_stack, stack_top,
              &tree_cp_root, &conn_point,
              ht);

  int validate_retries = -1;
VALIDATE_AND_INSTALL_COPY:
  if (++validate_retries >= TX_NUM_RETRIES)
    goto RETRY;

  while (avl->avl_lock != LOCK_FREE) ;

  if (TX_BEGIN() == TM_BEGIN_SUCCESS) {
    if (avl->avl_lock != LOCK_FREE)
      TX_ABORT(TX_ABORT_LOCK_TAKEN);

    validate_copy(avl, key, node_stack,
                  stack_top, ht);
    install_copy(avl, key, conn_point,
                 tree_cp_root);
    TX_COMMIT();
  } else {
    if (abort due to validation error)
      goto RETRY;
    else
      goto VALIDATE_AND_INSTALL_COPY;
  }
  return 1;
}
```

Figure 7: Insert operation.

```
118 void traverse_with_stack(avl_t *avl,int key,
          avl_node_t *node_stack[MAX_HEIGHT],
          int *stack_top)
119 {
120   avl_node_t *parent, *leaf;
121
122   parent = NULL;
123   leaf = avl->root;
124   *stack_top = -1;
125
126   while (leaf) {
127     node_stack[++(*stack_top)] = leaf;
128
129     if (leaf->key == key)
130       return;
131
132     parent = leaf;
133     if (key < leaf->key) leaf = leaf->left;
134     else                 leaf = leaf->right;
135   }
136 }
```

Figure 8: Helper function to traverse the tree. Keeps the whole path of traversed nodes in a stack of pointers.

```
137 void validate_copy(avl_t *avl, int key,
          avl_node_t *node_stack[MAX_HEIGHT],
          int stack_top, hash_table_t ht)
138 {
139   avl_node_t *child, **p, *n;
140
141   if (avl->root != node_stack[0])
142     TX_ABORT(TX_VALIDATION_ERROR);
143
144   for (i=0; i < stack_top; i++) {
145     if (key < stack_top[i]->key)
146       child = stack_top[i]->left;
147     else
148       child = stack_top[i]->right;
149     if (child != node_stack[i+1])
150       TX_ABORT(TX_VALIDATION_ERROR);
151   }
152
153   for (every entry e in ht) {
154     p = e.address; v = e.value;
155     if (*p != v)
156       TX_ABORT(TX_VALIDATION_ERROR);
157   }
158 }
159
160 void install_copy(avl_t *avl, int key,
                    avl_node_t *conn_point,
                    avl_node_t *tree_cp_root)
161 {
162   if (conn_point == NULL)
163     avl->root = tree_cp_root;
164   else if (key < conn_point->key)
165     conn_point->left = tree_cp_root;
166   else
167     conn_point->right = tree_cp_root;
168 }
```

Figure 9: Helper functions used to validate and install the private modified copy.

```
169 void insert_node_and_rebalance(
      int key, void *value,
      avl_node_t *node_stack[MAX_HEIGHT],
      int stack_top,avl_node_t **cp_root,
      avl_node_stack **conn_point,
      hash_table_t ht)
170 {
171   avl_node_t *curr_cp;
172
173   *cp_root = new_node(key, value);
174   *conn_point = node_stack[stack_top--];
175
176   while (stack_top >= 1) {
177     if (*conn_point == NULL)
178       break;
179     if ((*cp_root)->height + 1 <=
            (*conn_point)->height)
180       break;
181
182     curr_cp = copy_node(*conn_point);
183     curr_cp->height =
              (*cp_root)->height + 1;
184     if (key < curr_cp->key) {
185       curr_cp->left = *cp_root;
186       ht_insert(ht, &(*conn_point)->right,
                  curr_cp->right);
187     } else {
188       curr_cp->right = *cp_root;
189       ht_insert(ht, &(*conn_point)->left,
                  curr_cp->left);
190     }
191     *cp_root = curr_cp;
192     *conn_point = (stack_top >= 0) ?
                node_stack[stack_top--] : NULL;
193
194     if (balance(curr_cp) == 2) {
195       if (balance((*cp_root)->left) == 1) {
196         *cp_root = rotate_r(*cp_root);
197       } else {
198         (*cp_root)->left =
                  rotate_l((*cp_root)->left);
199         *cp_root = rotate_r(*cp_root);
200       }
201       break;
202     } else if (balance(curr_cp) == -2) {
203       if (balance((*cp_root)->right) == -1){
204         *cp_root = rotate_l(*cp_root);
205       } else {
206         (*cp_root)->right =
                  rotate_r((*cp_root)->right);
207         *cp_root = rotate_l(*cp_root);
208       }
209       break;
210     }
211   }
212 }
```

Figure 10: The function used by *RCU-HTM* that performs the insert and rebalance phase of an AVL tree by creating copies of the affected nodes.

- Each software thread is pinned to a hardware thread. Hyperthreading is enabled only on 44 thread executions.
- For the transactions of *RCU-HTM* based trees and *cop-avl* we set the number of transactional retries before resorting to the global lock non-transactional fallback (i.e., the `TX_NUM_RETRIES` parameter) to 10.
- To minimize the overheads of memory allocation we use per thread pre-allocated memory for the node copies in *RCU-HTM*. Similarly to previous works [2], [25], we perform no memory reclamation during our experiments.
- We test our implementations using read-only, read-dominated and write-only workloads consisting of 100%, 80% and 0% lookups, respectively, while the rest of the operations are equally divided between insertions and deletions. The read-only and write-only cases are the best and worst cases, respectively, for *RCU-HTM*, while the read-dominated workload represents a typical application workload.
- As the key range effectively determines the size of the tree, we evaluate our implementations for ranges of 200, 2K, 20K, 2M and 20M keys, which represent small to large-sized trees. At the beginning of each run, the tree is initialized to contain half the keys of the selected range.
- All reported results are the average of 10 independent executions with no significant variance.

Figure 11 presents the results of the concurrent BSTs for all workloads and tree sizes.

Read-only workloads. The leftmost column of Figure 11 presents the results for the read-only workloads. As is evident, for all tree sizes *RCU-HTM* trees and *rcu-mrsw-avl* outperform all other implementations. *cop-avl* also scales thanks to the absence of conflicts (which translates to zero abort rate for its transactions). However, its performance is lower for two reasons: first, in the small trees, where traversals are very fast, the cost of starting and ending a transaction is comparable to the cost of the traversal itself; second, in the large trees its larger node size (tree nodes in *cop-avl* have 3 additional fields to enable the validation of the traversal) results in larger memory footprint for its traversals. *lb-avl* does not perform as well as the rest of the implementations, because at each step of a traversal the node's version number is read and validated. As our results reveal this imposes a high overhead. Finally, the significance of balancing the tree is evident from the performance of the two unbalanced implementations, *citrus-bst* and *lf-bst*. Neither of them manages to achieve performance comparable to the balanced implementations.

Read-dominated workloads. The middle column depicts the results for the read-dominated workloads, with 80% lookups. For all tree sizes, except for the smaller one with 200 keys, *RCU-HTM* trees outperform all other BSTs. The performance of the unbalanced non-blocking *lf-bst* shows that, under high contention, rebalancing the tree imposes high overheads. This is the reason why *lf-bst* is the most performant in the 200 keys case. However, as the tree size increases and contention is reduced, its performance drops below that of the balanced trees. This result highlights the importance of balancing the tree. Similar conclusions can be drawn from the behaviour of the other unbalanced BST, *citrus-bst*.

The impact of allowing only a single updater is evident from the performance of *rcu-mrsw-avl*. Even with a relatively low update ratio (i.e., 20% of total operations) its performance collapses. The HTM-based *cop-avl* tree scales well for all tree sizes, except for the smaller one. In this case, it doesn't scale as well as *RCU-HTM* based trees for two reasons: first, traversals may need to restart, and, second, the transactions executed by the updaters contain multiple memory writes and are, thus, prone to aborts.

Write-only workloads. In the rightmost column of Figure 11 we present the results for the write-only workloads, which is generally the worst case scenario for RCU-based and HTM-based implementations. In this execution scenario *rcu-mrsw-avl* is similar to a single global lock implementation and provides no scalability. *cop-avl* does not scale for the 200 and 2K trees, as well as for 44 threads on a 20K tree. *RCU-HTM* based trees manage to combine the best of both worlds by efficiently allowing concurrent writers and by avoiding the majority of aborts. Even in the most contended cases (i.e., 200 and 2K keys) *RCU-HTM* scales and provides throughput fairly competitive to the best performing trees.

Figure 12 summarizes the results of our experiments providing an overall evaluation of the benefits of *RCU-HTM*. The numbers above the bars in Figure 12a represent the average path length that is been traversed in each implementation. These numbers validate our argument that a balanced, internal BST with on-time deletion is, despite its complexity, the best option in terms of performance, as the unbalanced trees (*lf-bst* and *citrus-bst*) have longer traversal paths. The bars present geometric means of speedups over the serial executions. We provide these means averaged over three dimensions:

Tree size. The only case where *RCU-HTM* is not the best implementation is for the smaller tree with 200 keys. Even in this case, however, its performance is competitive to the best performing, *lf-bst*. For all other tree sizes *RCU-HTM* outperforms all other implementations. An important note here, is that the second best competitor is not always the same. This is expected, as different BSTs fit better the different conditions caused by the combination of workloads and tree sizes; however, *RCU-HTM* is consistently the top performing BST, indicating its robustness across different use cases.

Workload. As is evident from Figure 12b for the read-only workload *RCU-HTM* performs equally to *rcu-mrsw-avl*. This was expected, as when no updaters are executing the two implementations execute the same code. However, the performance of *RCU-HTM* for the other two workloads showcases the benefit of allowing multiple updaters by using HTM. Even in the worst case scenario for RCU, the write-only workload, *RCU-HTM* manages to be competitive with the lock-based and non-blocking implementations.

Total. In Figure 12c we present the average speedups over serial for all tree sizes and workloads for the cases of 22 and 44 threads. In both cases *RCU-HTM* is 18% better than the second best implementation.

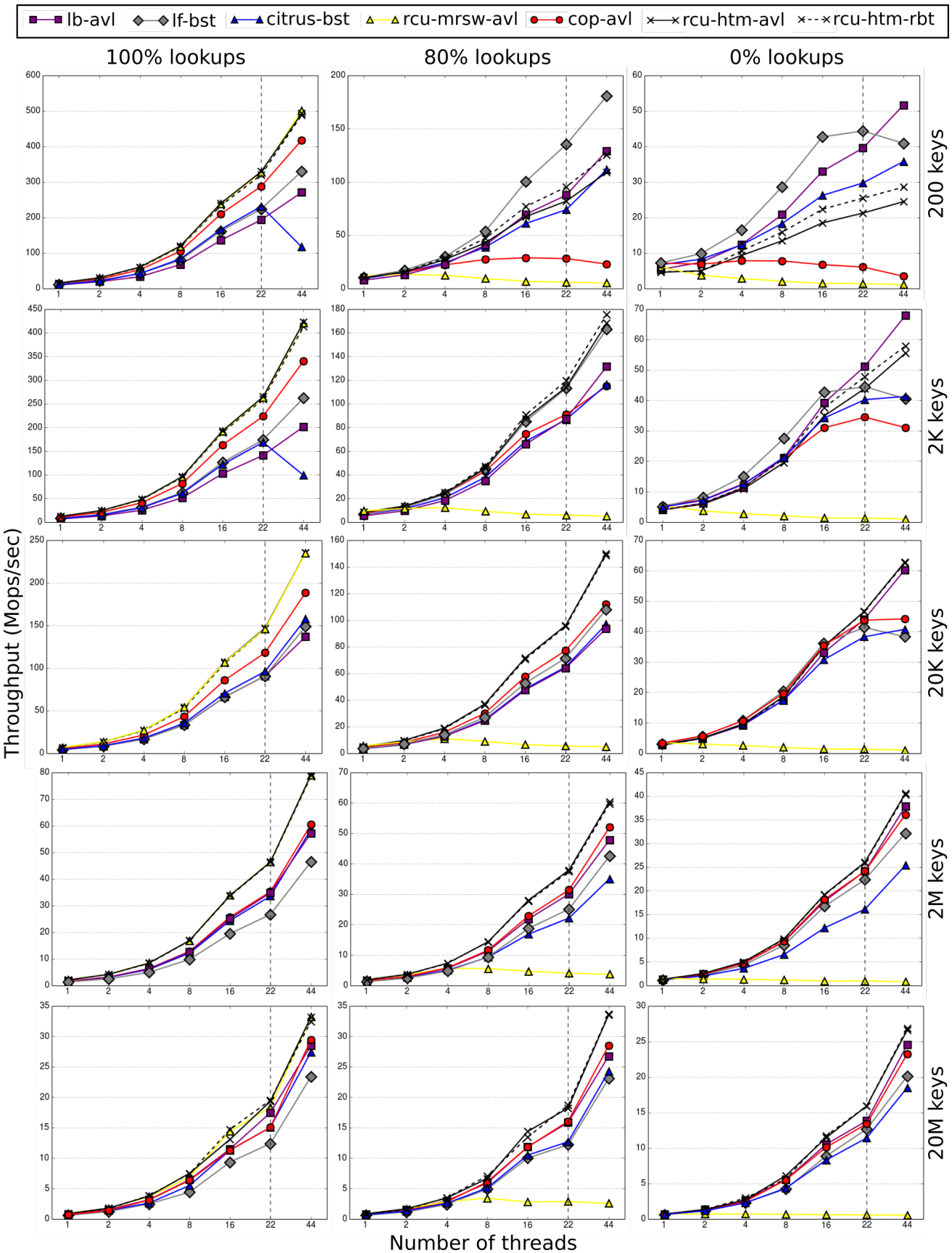


Figure 11: Throughput of concurrent tree implementations. The rows represent different tree sizes and the columns different workloads. The vertical line in each plot shows the point after which we utilize hyperthreads. Notice the differences in the y-axis range between the figures.

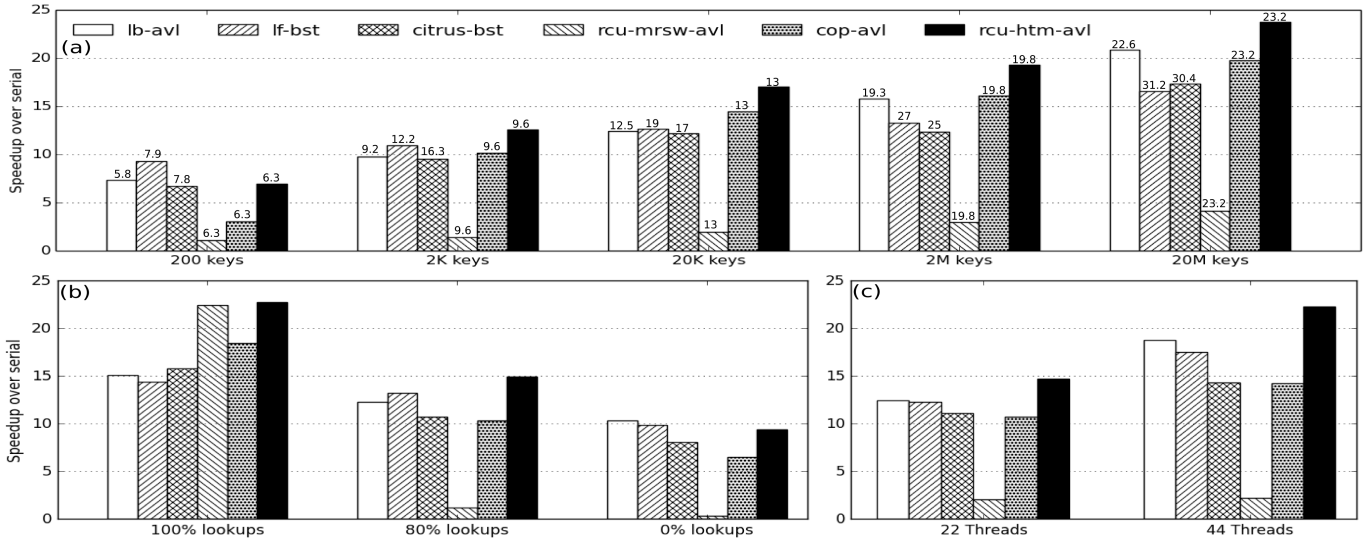


Figure 12: Geometric means of speedups over serial presented (a) by tree size (22 threads), (b) by workload (22 threads) and (c) in total (22 and 44 threads). The speedups have been calculated using the execution time of a serial internal AVL tree as the baseline. The numbers above the bars in (a) present the average number of nodes traversed in each implementation.

VII. Discussion

In this work we mainly focused on the performance of BSTs. However, when designing and implementing a concurrent data structure, three more factors are typically taken into consideration: robustness, programmability and memory requirements. In this section we provide a brief discussion for each of these factors regarding *RCU-HTM* based BSTs.

Robustness, i.e., the ability of a data structure to handle cases such as thread delays or failures; non-blocking implementations are the most robust, because threads make progress independently of other thread failures. On the contrary, lock-based BSTs are the least robust, because when a thread holding a lock is delayed, other threads wait for the lock to be released and make no progress. The granularity and the frequency by which locks are acquired, directly affects the robustness of a lock-based BST. BSTs implemented with *RCU-HTM* are robust for two reasons: first, read-only operations are not affected by other operations and can never be delayed due to other thread actions; second, updaters are only delayed by other updaters in the extremely rare case when an update operation has restarted for more than `TX_NUM_RETRIES` times. In this case the updater will acquire the global lock. Our experiments validated that this is very uncommon. More specifically, in the 200 keys tree with 0% lookups, where we expect the highest number of aborts and therefore retries, the global lock was acquired by only 1.6% of the update operations. On every other case this fraction was zero.

Programmability, i.e., the effort required to implement and maintain a data structure; although this is the hardest to quantify factor we intend to focus more on it in our future work. We plan to use metrics such as the percentage of code or the Halstead complexity metric [15] to compare *RCU-HTM* against state-of-the-art BSTs.

Memory requirements, i.e., the size of the memory footprint required by a data structure; unlike other approaches (e.g., lazy deletion and external trees), *RCU-HTM* trees do not leave unused nodes in the tree. They stress the memory management system by creating node copies, however, the replaced nodes can be reclaimed as soon as no other threads are accessing them. Our future work aims at exploring ways to safely reclaim and free these nodes.

VIII. Conclusion & Future Work

In this work we introduced *RCU-HTM*, a technique that combines HTM with RCU in an innovative way to implement highly scalable concurrent BSTs. As our experimental evaluation revealed trees implemented using *RCU-HTM* outperform previous RCU- and HTM-based approaches as well as state-of-the-art non-blocking and lock-based ones for a wide range of workloads and tree sizes.

As future work we plan to explore how memory reclamation techniques can be applied to *RCU-HTM* and investigate their impact on performance. Moreover, we intend to find other data structures on which *RCU-HTM* can be applied with similar performance benefits.

Acknowledgment

This research has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement no. 732366 (ACTiCLOUD).

REFERENCES

[1] M. AdelsonVelskii and E. M. Landis, "An algorithm for the organization of information," DTIC Document, Tech. Rep., 1963.

[2] M. Arbel and H. Attiya, "Concurrent Updates with RCU: Search Tree As an Example," in *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, ser. PODC '14. New York, NY, USA: ACM, 2014, pp. 196–205. [Online]. Available: http://doi.acm.org/10.1145/2611462.2611471

[3] M. Arbel and A. Morrison, "Predicate rcu: An rcu for scalable concurrent updates," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/2688500.2688518

[4] H. Avni and B. C. Kuszmaul, "Improving htm scaling with consistency-oblivious programming," *TRANSACT*, vol. 6, p. 5, 2014.

[5] R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica*, vol. 1, no. 3, pp. 173–189, 1972. [Online]. Available: http://dx.doi.org/10.1007/BF00288683

[6] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun, "A practical concurrent binary search tree," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '10. New York, NY, USA: ACM, 2010, pp. 257–268. [Online]. Available: http://doi.acm.org/10.1145/1693453.1693488

[7] T. Brown, F. Ellen, and E. Ruppert, "A General Technique for Non-blocking Trees," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: ACM, 2014, pp. 329–342. [Online]. Available: http://doi.acm.org/10.1145/2555243.2555267

[8] B. Chatterjee, N. N. Dang, and P. Tsigas, "Efficient lock-free binary search trees," *CoRR*, vol. abs/1404.3272, 2014. [Online]. Available: http://arxiv.org/abs/1404.3272

[9] T. Crain, V. Gramoli, and M. Raynal, "A speculation-friendly binary search tree," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 161–170. [Online]. Available: http://doi.acm.org/10.1145/2145816.2145837

[10] T. Crain, V. Gramoli, and M. Raynal, "A contention-friendly binary search tree," in *Proceedings of the 19th International Conference on Parallel Processing*, ser. Euro-Par'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 229–240. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40047-6_25

[11] T. David, R. Guerraoui, and V. Trigonakis, "Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 631–644. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694359

[12] D. Drachsler, M. Vechev, and E. Yahav, "Practical Concurrent Binary Search Trees via Logical Ordering," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: ACM, 2014, pp. 343–356. [Online]. Available: http://doi.acm.org/10.1145/2555243.2555269

[13] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, "Non-blocking Binary Search Trees," in *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, ser. PODC '10. New York, NY, USA: ACM, 2010, pp. 131–140. [Online]. Available: http://doi.acm.org/10.1145/1835698.1835736

[14] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *Foundations of Computer Science, 1978., 19th Annual Symposium on*, Oct 1978, pp. 8–21.

[15] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977.

[16] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ser. ISCA '93. New York, NY, USA: ACM, 1993, pp. 289–300. [Online]. Available: http://doi.acm.org/10.1145/165123.165164

[17] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[18] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990. [Online]. Available: http://doi.acm.org/10.1145/78969.78972

[19] P. W. Howard and J. Walpole, "Relativistic red-black trees," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 16, pp. 2684–2712, 2014. [Online]. Available: http://dx.doi.org/10.1002/cpe.3157

[20] S. V. Howley and J. Jones, "A Non-blocking Internal Binary Search Tree," in *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '12. New York, NY, USA: ACM, 2012, pp. 161–171. [Online]. Available: http://doi.acm.org/10.1145/2312005.2312036

[21] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, Jun. 1981. [Online]. Available: http://doi.acm.org/10.1145/319566.319567

[22] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: Artful indexing for main-memory databases," in *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ser. ICDE '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 38–49. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2013.6544812

[23] A. Matveev, N. Shavit, P. Felber, and P. Marlier, "Read-log-update: A lightweight synchronization mechanism for concurrent programming," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 168–183. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815406

[24] P. E. Mckenney and J. D. Slingwine, "Read-Copy Update: Using Execution History to Solve Concurrency Problems," in *Parallel and Distributed Computing and Systems*, Las Vegas, NV, Oct. 1998, pp. 509–518.

[25] A. Natarajan and N. Mittal, "Fast Concurrent Lock-free Binary Search Trees," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: ACM, 2014, pp. 317–328. [Online]. Available: http://doi.acm.org/10.1145/2555243.2555256

[26] R. Sedgewick, "Left-Leaning red black trees." [Online]. Available: http://www.cs.princeton.edu/\~{}rs/talks/LLRB/LLRB.pdf

[27] D. Siakavaras, K. Nikas, G. Goumas, and N. Koziris, "Massively Concurrent Red-Black Trees with Hardware Transactional Memory," in *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, Feb 2016, pp. 127–134.

[28] D. Siakavaras, K. Nikas, G. Goumas, and N. Koziris, "Performance analysis of concurrent red-black trees on htm platforms," *TRANSACT*, 2015.

[29] Z. Wang, H. Qian, J. Li, and H. Chen, "Using restricted transactional memory to build a scalable in-memory database," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: ACM, 2014, pp. 26:1–26:15. [Online]. Available: http://doi.acm.org/10.1145/2592798.2592815

APPENDIX

A. Abstract

The artifact contains the source code of the concurrent tree implementations used in our experimental evaluation, as well as a serial AVL tree which serves as baseline for calculating speedups. We also include scripts for executing the experiments and producing plots similar to Figures 11 and 12.

B. Description

1) Check-list (artifact meta information):

- **Algorithm:** Concurrent BSTs, AVL tree, Red-Black tree
- **Program:** C code with inline HTM assembly instructions
- **Compilation:** gcc 4.9.2 with -O3 flag
- **Binary:** x86_64 dynamically linked executables
- **Run-time environment:** Debian jessie 8.3, 4.7.0 kernel
- **Hardware:** Any Intel processor that supports TSX (Haswell and all successors)
- **Experiment workflow:** Clone repo; make; run; create plots
- **Publicly available?:** Yes

2) How delivered:

The source code of *RCU-HTM* is provided on github along with instructions on how to compile and execute it.

3) Hardware dependencies:

To execute *RCU-HTM* an Intel processor with TSX support is required. Haswell processors and all their successors provide this functionality. To check the availability of TSX one needs to check if flags *tm, tm2* and *rtm* are reported in `/proc/cpuinfo`.

4) Software dependencies:

To execute the plot scripts, the following software is necessary:

- python: tested with version 2.7.6.
- python-numpy: tested with version 1.8.2.
- python-matplotlib: tested with version 1.3.1.

In typical Debian/Ubuntu linux distributions the above packages can be installed using the following commands:

```
$ apt-get update
$ apt-get install python python-numpy python-
    matplotlib
```

C. Installation

Clone our github repository (https://github.com/rcu-htm/pact-ae) and execute make command. Upon successfull compilation, the executables are created with their names starting with *x.* followed by the implementation name (e.g., *x.avl.int.rcu_htm*).

D. Experiment workflow

The first step consists of cloning the git repository and compiling the source code:

```
$ git clone https://github.com/rcu-htm/pact-ae
$ cd pact-ae
$ make
```

Upon successfull compilation, the executables should have been created:

```
$ ls x.*
x.avl.bronson x.avl.int.cop
x.avl.int.rcu_htm x.avl.int.rcu_sgl
x.avl.int.seq x.bst.aravind
x.bst.citrus x.rbt.int.rcu_htm
```

After that point, our provided script can be used to execute the benchmarks:

```
$ source ./scripts/source_me.sh
$ ./scripts/run.sh
```

E. Evaluation and expected result

After the execution of `run.sh` script, a directory has been created inside the *results* directory with a filename that consists of the date and hostname of the machine on which the script has been executed (e.g., 2017_06_12-15_06-node1).

Inside the directory are the output files for each combination of executable, init tree size and workload. There is also a *SERIAL* directory which includes the results of the serial execution. Finally, there is also a file called *INFORMATION* which contains the parameters of the specific execution.

We provide two scripts that can be used to produce figures, similar to Figure 11 and 12 in the paper:

```
$ cd scripts
$ source source_me.sh
$ ./create-figure-11-plots.sh ../results/2017
    _06_12-15_06-node1
$ ./create-figure-12-plots.sh ../results/2017
    _06_12-15_06-node1 22
```

The second argument of `create-figure-12-plots.sh` is the number of threads for which to create the plots. The plots can be found in the *plots* directory.

F. Experiment customization

By modifying appropriately the `scripts/source_me.sh` file the following parameters can be set:

- *RCU_HTM_NR_EXECUTIONS:* Number of times to repeat each execution.
- *RCU_HTM_RUNTIME:* Duration of each benchmark in seconds.
- *RCU_HTM_WORKLOADS:* Different workloads to be executed.
- *RCU_HTM_INIT_SIZES:* Different init tree sizes.
- *RCU_HTM_THREADS_CONF:* Different number of threads configurations.
- *RCU_HTM_EXECUTABLES:* The executables to be taken into consideration.
- *RCU_HTM_PLOT_LABELS:* The labels to be used for each executable in the produced plots.
- *RCU_HTM_SERIAL_EXE:* The serial tree implementation to be considered as the baseline for speedup calculations.

G. Notes

The methodology followed for the submission and review of this artifact can be found in the following link:
http://ctuning.org/ae/submission-20170414.html