Combining HTM and RCU to Implement Highly Efficient Balanced 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 combine Hardware Transactional Memory (HTM) with Read-Copy-Update (RCU) to implement highly scalable concurrent balanced Binary Search Trees (BSTs). The two key features of our approach are: a) read-only operations require no synchronization or restarts and b) tree modifications are first performed in private copies of subtrees, then HTM is used to validate their consistency, and upon successful validation, the copy is installed back in the shared tree by modifying only a single pointer.

Our approach can be applied to any type of balanced BSTs (e.g., Red-Black, AVL, B-trees) and for the purpose of this paper we test it in an AVL tree. As our experimental evaluation reveals, the proposed AVL tree implementation achieves higher throughput and scalability compared to previous approaches for parallelizing balanced BSTs with HTM. More specifically on a machine comprising 22 physical cores (44 hardware threads) our tree outperforms other alternatives by 70% and 220%. As shown, this gain is attributed both to the much faster read-only operations and to the reduction in abort ratios thanks to the reduction of transactions' write-sets.

Keywords Hardware Transactional Memory, Read-Copy-Update, Concurrent Data Structures, Balanced Binary Search Trees

1. Introduction

Balanced Binary Search Trees (BSTs) are used in a wide range of applications. Their properties allow for highly efficient indexing of information and, contrary to unbalanced BSTs, they manage to bound the time to search. However, when dealing with concurrent implementations, there are two major challenges. First, when rebalancing the tree, threads may traverse and modify nodes in opposite directions, making it extremely difficult to apply fine-grained synchronization mechanisms (e.g., fine-grain locking). Second, when removing a node with two children, its successor needs to be found and removed. The successor may be many links away from that node and removing it requires exclusive access to the whole path.

In order to overcome the difficulty imposed by the rebalancing operations, researchers have either turned their attention to the less challenging unbalanced BST [1–3] or to relaxing the balancing conditions [4–6]. In a relaxed balance BST, the rebalancing rules are less strict at the expense of allowing the height of the tree to become unbalanced.

To overcome the problem of removing a node with two children, some concurrent tree implementations are based on external representation of the tree structure. External trees, as opposed to internal ones, store the actual information on leaf nodes (i.e., nodes with no children) and the internal nodes are used only for routing to the appropriate leaf. This way, the removal of a node entails deleting a leaf node and no successor needs to be found. On the other hand, external trees occupy twice as much memory as internal ones (i.e., to store N keys an external tree requires $(N-1)+N$ memory space) and traversals have to follow longer paths as they always end at a leaf node.

The advent of Hardware Transactional Memory (HTM) on modern processors such as Intel's Haswell and successors and IBM's Power8, gives the opportunity to devise simple, yet scalable concurrent implementations of complex data structures including balanced BSTs. The most straightforward approach to parallelizing a balanced BST with HTM is to enclose each operation in a single transaction making it atomic (we refer to this approach as cg-htm, where cg stands for coarse-grained). While such an implementation is extremely simple, it has been shown that, under certain con-

TRANSACT '17, February 4–8, 2017, Texas, Austin, USA.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

ditions, its performance is poor and highly unstable [7]. The main reason for this is its large transactions, both in terms of memory footprint and duration. Current HTM implementations are not well suited for such long transactions.

In order to overcome the problems of cg-htm, previous work combines HTM with Consistency Oblivious Programming (COP) [8] (we refer to this approach as cop-htm). In cop-htm an operation is split in three phases: a) traversal of the tree from the root to the appropriate leaf, b) validation that the traversal ended up at the correct leaf and c) modifications of tree nodes as dictated by the rebalancing rules. The first phase is performed without any synchronization and thus the traversal may end up at a wrong path of the tree due to concurrent rotations. The second and third phase are performed within a single HTM transaction that guarantees their atomic execution. The third phase is only relevant to insertion and deletion.

Cop-htm manages, in general, to overcome the limitations of cg-htm, however, it also has some drawbacks that limit scalability. The first one is that read-only operations, such as lookups or insertions and deletions that have no side effects (i.e., insertions that find the key in the tree and deletions that do not find the key in it), may have to be restarted. As these read-only operations are the majority in most workloads, it is highly desirable to allow them to proceed without restarting regardless of concurrent modifying operations. The second drawback is that the rebalancing operations which may require several tree modifications are performed in an HTM transaction. This results in a transaction with a relatively large write-set, which is, potentially, vulnerable to aborts. Moreover, by enclosing the whole rebalance operation in a single transaction, concurrent traversals may falsely cause the rebalancing transaction to abort. This may occur when a traversing thread reads a field of a node that has been modified by an ongoing rebalance transaction. The HTM system will report this as a conflict and cause the transaction to abort.

Our solution: In this paper we propose an alternative way of using HTM to implement concurrent balanced BSTs. Our approach, which we call rcu-htm, combines HTM with the Read-Copy-Update technique (RCU) [9] in order to achieve two goals: a) allow read-only operations to proceed independently of concurrent modifications on the tree, without the danger of following a wrong path, in which case they would have to be restarted and b) use transactions that perform only a single write on shared data, thus being less vulnerable to aborts.

RCU is a well-known technique, where threads that need to update a part of a data structure, first create a private copy of the affected part, then apply the appropriate modifications on the private copy, and finally install the modified part in the shared data structure in an atomic fashion. This allows threads that only read parts of the data structure to proceed without any synchronization, thus resulting in extremely fast traversals of the data structure. RCU's main aim is to permit multiple readers to execute concurrently with a single updater. To enable multiple updaters to execute simultaneously, synchronization among them is necessary, which is typically achieved with fine-grain locking mechanisms. RCU has been applied to BSTs, but these implementations either allow only a single updater [10] or are restricted to unbalanced BSTs and updaters are synchronized with fine-grain locking [11, 12]. As we explain in Section 3, our approach combines the positives of these two implementations and provides a balanced BST while allowing concurrent updaters.

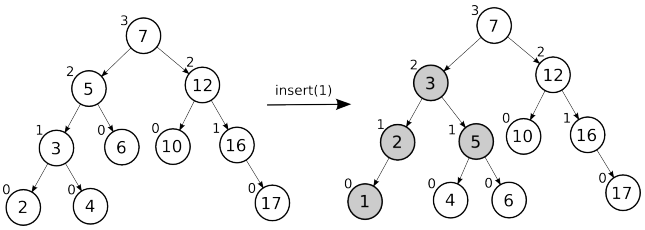


Figure 1. Insertion and rebalance performed in place in an AVL tree. The numbers above the tree nodes represent the height of each node. Nodes in gray are those that are modified. In the specific example rebalancing consisted of 3 height updates (nodes 2,3,5) and a right rotation over node 5.

In our approach, traversal of the tree is performed without any synchronization and modifications are first being applied on private copies, in exactly the same way as in RCU. We then we exploit HTM to validate that the replaced part of the tree has remained intact since it has been read and avoid having simultaneous modifications discard each other. This allows us to permit multiple updating threads to operate on the tree. Our evaluation reveals that with this approach we are able to implement highly scalable concurrent balanced BSTs which outperform the two afforementioned HTM-based implementations, cg-htm and cop-htm, by up to 220% and 70% respectively.

2. Background

In this section we provide a brief overview of the necessary background. In this paper, we focus on the widely used AVL balanced BST, however, our approach is generic and can be applied to other types like Red-Black trees and B-trees.

2.1 AVL Trees

An AVL tree is a self-balancing binary search tree in which the heights of the left and right child branches of a node differ by no more than one. When inserting or deleting a node of the tree, a rebalancing phase may be necessary which potentially updates node height and rotates nodes. An example insertion in an AVL tree is shown in Figure 1.

The process of rebalancing the tree becomes a bottleneck for concurrent tree implementations, because many parts of the tree may need to be modified and many locks need to be acquired. Moreover, the structure of the tree makes it difficult to guarantee that no deadlock occurs. This has led to the idea of relaxing the balance of the tree and several concurrent relaxed trees have been proposed [4–6]. While these implementations perform well in most common scenarios, there are cases where some paths of the tree become very long.

Another factor that complicates the implementation of concurrent BSTs is the deletion of a node with two children. In that case the operation proceeds as follows: a) the immediate successor is found, b) the key of the successor replaces the key of the node with two children and c) the successor node is removed from the tree. A thread performing a deletion in this way needs to have exclusive access to the whole path between the node with the key to be deleted and its successor. To avoid this complexity, external trees can be used, on which the actual information is stored on leaf nodes. Internal nodes in this case contain only keys and are used solely for routing purposes. In external trees there is no need to remove a node with two children, thus simplifying the delete operation; on the other hand they occupy much more memory than their internal counterparts and, typically, they entail longer path traversals.

2.2 Concurrent AVL trees with HTM

Using HTM the complexity related to balanced BSTs is delegated to the underlying TM system. Programmers do not have to cope with acquiring and releasing locks on tree nodes; they only need to mark the regions of code that need to be executed in an atomic fashion as part of a transaction.

2.2.1 Coarse-grained HTM

The most straightforward approach to implement concurrent balanced BSTs with HTM is to enclose each tree operation in an HTM transaction [7]. This way the entire operation (e.g., the traversal of the tree and the rebalancing) is performed in an atomic way and the synchronization among multiple threads is handled by the HTM system. This approach is slightly more complicated than a coarse-grained locking scheme but performs relatively good in several cases. However, it may suffer from severe performance slowdowns. By including the traversal phase in the transaction, all the nodes in the path are added in the transaction's read-set. This, apart from increasing the size of the transaction, may lead to unnecessary aborts when higher levels of the tree are modified.

2.2.2 Consistency-Oblivious-Programming HTM

Consistency Oblivious Programming (COP) [8] has been used as a way to avoid executing the traversal phase inside a transaction. Using this approach a tree operation is split in the three following phases: a) traverse the tree using no synchronization, b) validate that the traversal has navigated to the correct node and, if validation is successful, c) perform the insertion/deletion and rebalancing. The validation step is necessary because asynchronized traversals may follow a wrong path of the tree due to concurrent modifications (an example is shown in Figure 2(a)).

To guarantee correctness, steps (b) and (c) are performed within an HTM transaction. In cop-htm transactions are smaller and less vulnerable to spurious and unnecessary aborts in comparison to cg-htm. However, there are two shortcomings. First, traversals, although not acquiring any locks, may follow a wrong path and need to restart from the root. Second, transactions enclose the whole rebalance process which may perform multiple writes.

3. Our Approach

Our approach combines HTM with principles drawn from RCU and can be used to implement concurrent BSTs with the following two characteristics:

a) Read-only operations, such as lookups or insertions and deletions with no side effects (i.e., insertions that find the key in the tree and deletions that do not find the key in it), do not require any synchronization and can proceed independently of concurrent modifications of the tree without any chance of restarting. Practically, trees implemented with our approach require no changes in the source code of the serial version of these operations.

b) Tree modifications use HTM transactions whose majority of memory accesses are reads and contain only a single write. This way we manage to avoid unnecessary and spurious aborts.

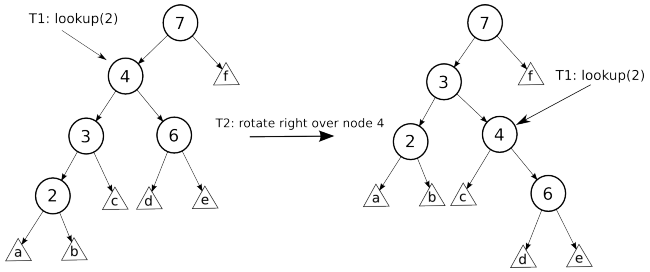
In the next paragraphs we explain in detail how we perform modifications and the three basic tree operations, *lookup()*, *insert()* and *delete()* in our rcu-htm based internal AVL tree.

3.1 Modifications

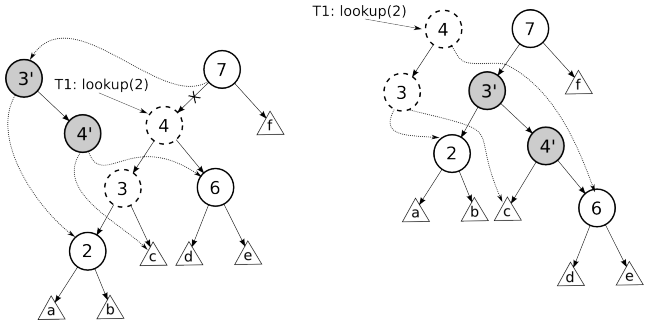
In rcu-htm, we perform modifications of the tree in private copies, rather than in place. An example is given in Figure 2. Figure 2(a) shows a rotation performed with in place modifications of the tree, while Figure 2(b) represents the case where private copies are used. After applying the appropriate modifications on the private copies, the child pointer of a single node needs to be updated (in this case the left child of node 7) so as to point to the modified copy. Given that a write in a single memory location is atomic, the modifications are becoming atomically visible to other threads.

3.2 Lookup

When modifications are performed in private copies, the *lookup* operation can proceed without synchronization and it never follows a wrong path of the tree, in which case it would have to restart from the root. The *lookup* operation in



(a) Lookup operation being led to the wrong subtree by a concurrent rotation. T1 will incorrectly conclude that key 2 is not in the tree.



(b) Lookup is kept on the right track when rotation is being performed in a copy and not in place. Gray nodes are the private copies and nodes with broken outline are the ones being replaced. Now T1 will find key 2 because although nodes 3 and 4 are no longer part of the tree their child pointers have been kept intact.

Figure 2. An illustration of how lookups can benefit when modifications are performed using private copies and not in place.

rcu-htm is performed in exactly the same way as in a serial internal AVL tree.

Figure 2 illustrates how lookups avoid synchronizing when modifications are performed in copies rather than in place. In Figure 2(a) thread T1 searching for key 2 has reached node with key 4, when another thread performs a right rotation over the same node as part of some rebalancing operation. The rotation leads the traversal of T1 to the wrong path and key 2 cannot be found, albeit it is present in the tree. Figure 2(b) presents the case when modifications are performed in private copies. Now, T2 creates a copy of nodes 3 and 4 that it will modify, performs the rotation on the copied nodes, 3' and 4', and then swaps the left child of node 7 to point to node 3'. This way, the original nodes 3 and 4 become inaccessible from the root of the tree, while T1's lookup is still able to navigate to the correct node that contains key 2. The only modification performed on shared data is the swap of node 7's child pointer and, since this is a single memory location, it is performed atomically. This is why reader threads do not need any synchronization; they will either see the whole modification or nothing. For example in Figure 2(b) thread T1 could either see T2's change and walk through node 3' or, as is the case in the example, not observe the change and go through node 4.

3.3 Insert

The *insert* operation in rcu-htm is split in the following four phases:

1. **Traversal:** Traverse the tree until either the key to be inserted is found, or the node that will be the parent of the new node has been reached (we refer to this node as the *insertion point*). If the key is found in the tree, no further action is required and the operation returns. Otherwise the next steps need to be performed.
2. **Modifications:** Perform the insertion and rebalance the tree by making copies of the affected nodes and applying the modifications in these private copies. Conceptually, when this step ends, a modified copy of the whole affected path has been created.
3. **Validation:** Validate that the copied path has not been modified by any other thread while the current thread was copying it. This step ensures that when the current thread installs its private copy, modifications performed by other threads are not discarded.
4. **Installation:** Install the private copy in the shared tree data structure. As already mentioned, this is done by swapping the child pointer of a single node (we refer to this node as the *connection point*).

The first two phases are performed without any synchronization, whereas the last two are executed within a single HTM transaction, so as to appear as a single atomic operation.

3.3.1 Traversal

Every insertion starts with a traversal of the tree, which ends either at a node that contains the key to be inserted or at the node under which the new node with the inserted key is going to be attached. When the key is found in the tree, the operation immediately returns and no further actions are required. As is the case for lookups, this traversal is performed with no synchronization and without the danger of restarting. The only difference with the traversal of a lookup is that, in this case, while moving down the tree we maintain a stack of pointers to the traversed nodes which is later used for the reverse traversal (i.e., moving towards the root) of the tree, performed during the rebalancing phase. We refer to this set of pointers as the *access path stack*. We don't use parent pointers for the reverse traversal, because it would complicate our approach.

3.3.2 Modifications

When the traversal reaches the insertion point, the second phase starts, where the tree modifications take place. In the case of AVL trees, the modifications involve node height updates and rotations. An example insertion in an AVL tree with in place modifications is presented in Figure 1. The insertion of a node with key 1 as the left child of node 2

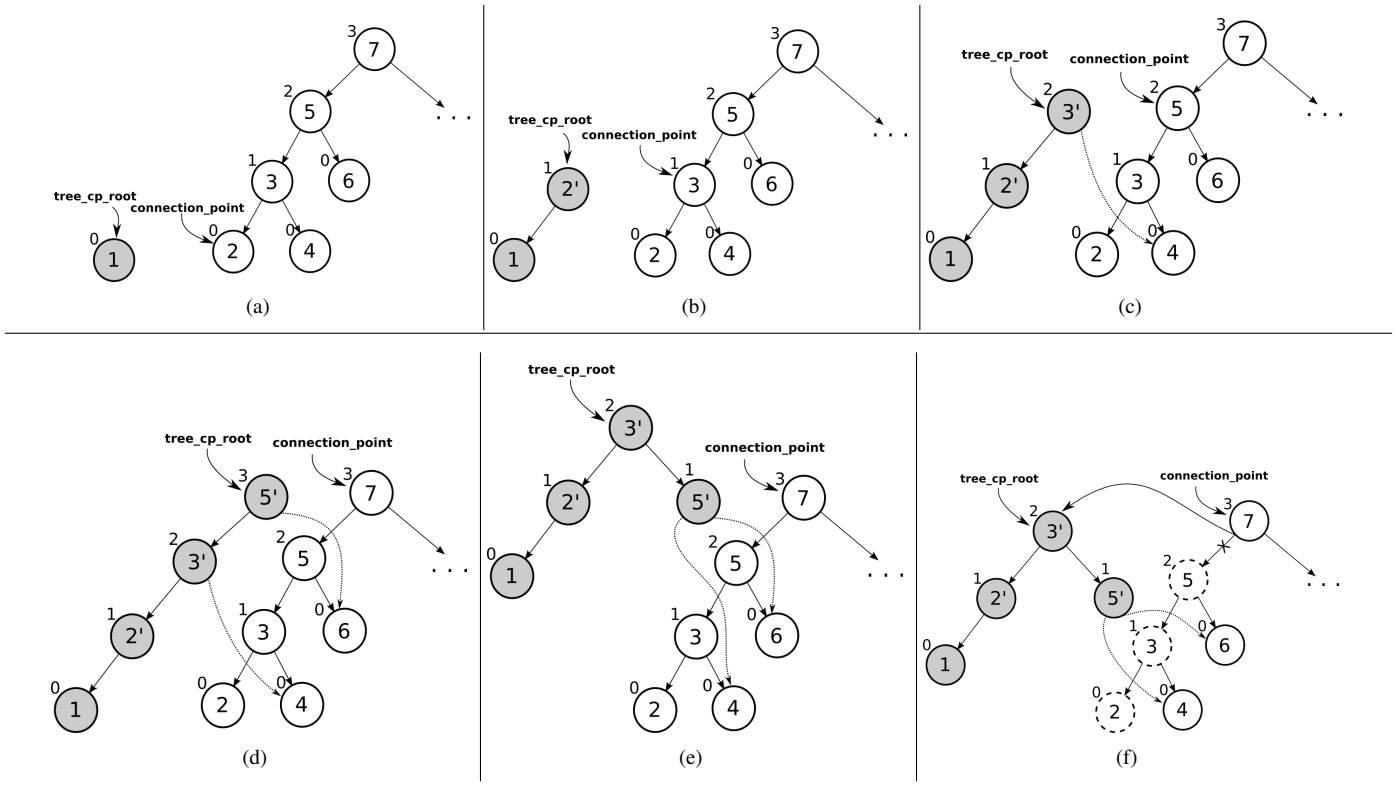


Figure 3. Insertion and rebalance using private copies in an AVL tree. In gray color are the private copies and nodes with broken outline are nodes that are no longer accessible from the root of the tree.

causes three node heights to be updated (nodes 2,3,5) and one right rotation over node 5.

In rcu-htm the modifications are performed in the following way: during the reverse traversal, when a node is to be modified, we create a copy on which we perform the appropriate modifications. This way, we eventually create a modified copy of the affected path. Figure 3 shows the individual steps performed as part of an insertion in an AVL tree using rcu-htm. At each step we maintain two pointers, *tree_cp_root* and *connection_point*. The first points to the root of the modified copy, while the second points to the node of the shared tree where the private copy should be attached.

At the first step (Figure 3(a)), the private copy contains only the newly created node with key 1. As we move upwards towards the root of the tree performing only node height updates (Figures 3(b)- 3(d)), copies of nodes 2,3 and 5 are added in the copy. The right rotation around node 5'(Figure 3(e)) concludes the necessary modifications of the rebalance phase. Now, the private copy is ready to be installed in the shared tree data structure by swapping the left child of node 7. However, the private copy needs to be validated first.

3.3.3 Validation and Installation

When the appropriate modifications have been performed in the private copy, the next two steps are its validation and installation in the shared tree. These two steps need to be performed in an atomic way, so we execute both inside a single HTM transaction. The aim of the validation phase is to ensure that the nodes that will be replaced have not been modified since the current thread read them. Otherwise, if we replace a node that has been altered, that modification will be lost. Such an example is given in Figure 4.

To validate that the private copy can safely be installed in the shared tree, we need to ensure that the following two conditions are true:

a) The *connection_point* is reachable from the root of the tree. This way, we make sure that when *tree_cp_root* is attached in the tree, it is also reachable from the root. We do so by traversing the tree from the root, searching for the key of the *connection_point*. If the traversal does not find the *connection_point* the validation fails.

a) All the nodes that will be replaced by the private copy have not been modified since they were copied. This way, we ensure that no modifications made by other threads are discarded.

If those two conditions are not true, the validation fails and we explicitly abort the transaction and restart the whole insert operation. If validation is successful, we can safely install the modified copy by swapping the appropriate child pointer of *connection_point* to point to *tree_cp_root*.

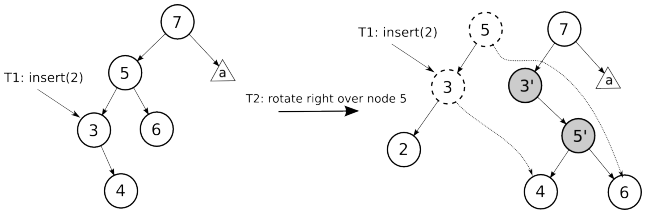


Figure 4. An example of a rotation performed by a thread T2 discarding the modifications of another thread T1. In the specific example T1's insertion of key 2 is discarded and the newly added node with key 2 is not reachable from the root of the tree.

3.4 Delete

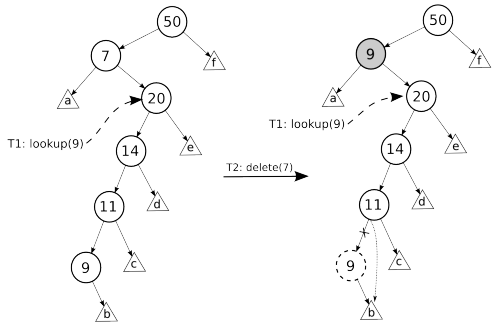
The *delete* operation in rcu-htm is performed in a way similar to insertion when the node to be removed has less than two children. However, when it has two children, its removal is more complicated. In that case we need to find the node containing the smallest key that is larger than the key to be deleted (that node is called the successor) and remove it after replacing its key with the key to be deleted. An example of deleting a node with two children is depicted in Figure 5(a). 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 in its old position. Such an erroneous execution is shown in Figure 5(a), where thread T1 is searching for key 9. T1 is not notified of the repositioning of key 9 and can never find it in the tree.

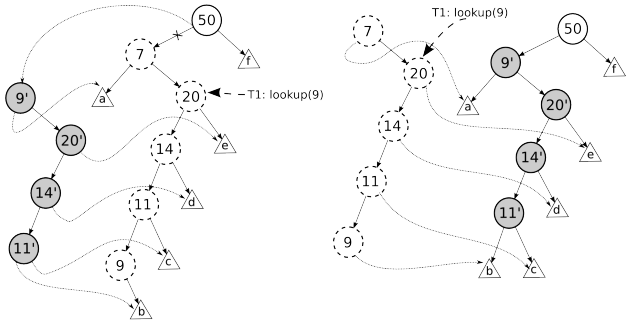
To avoid such problematic cases in rcu-htm, when a node with two children is to be removed we copy the whole path from the node that contains the key to be deleted to the successor node. If the copies performed during the rebalance phase already include all the nodes in this path, no further action needs to be performed. Figure 5(b) presents an example of deleting a node with two children using rcu-htm. In this case, T1 will be navigated to the node with key 9 regardless of the fact that in the new version of the tree key 9 is located higher in the tree.

3.5 Memory management

As is also the case for every concurrent data structure implementation that allows threads to access nodes even when they have been removed from the data structure, in rcu-htm it is not straightforward to release the memory of the removed nodes. When a thread removes a set of nodes from the tree by replacing them with modified copies, it is not safe to immediately free them as other threads may still keep references on them. One possible solution could be the use of an epoch-based memory allocator, such as ssmem used in [13]. In this work we have not dealt with this problem and none of the implementations of our experimental evaluation releases the allocated memory. It remains, though, an open future direction to explore how a memory reclamation scheme affects our concurrent trees.



(a) Deletion of key 7 from the tree. Key 7 is replaced by key 9 and the successor node (the one which previously contained key 9) is removed from the tree. Thread T1 will incorrectly conclude that key 9 is not in the tree.



(b) Deletion of key 7 using rcu-htm. By replacing the whole path connecting nodes 7 and 9 we allow thread T1 to successfully find key 9 in its previous position.

Figure 5. Removal of a node with two children from an internal AVL tree.

4. Experimental Evaluation

For our experiments we used a dual socket Intel Broadwell-EP server. The main characteristics of the server are summarized in Table 1. However, to avoid NUMA-related performance issues which are beyond the scope of this paper, in our experiments we only employ one socket.

Our evaluation includes the following concurrent AVL implementations:

- *cg-htm*: An internal AVL tree with each operation enclosed in an HTM transaction.
- *cop-htm*: An internal AVL tree that uses the COP [8] approach in order to synchronize the tree operations.
- *rcu-htm*: An internal AVL tree on which we have applied our proposed approach.

To evaluate the concurrent AVL implementations, we perform random operations varying the number of threads, the

Table 1. The characteristics of the server used for our experiments. Note that although this is a dual socket server only one socket is utilized in our experiments.

Name	**Broadwell-EP**
Processors	2 x Intel Xeon E5-2699 v4
# Cores	2 x 22
# Threads	88
Core clock	2.2 GHz
L1 (Data)	8-way, 32 KB, 64B block size
L2	8-way, 256 KB, 64B block size
L3	20-way, 56 MB, 64B block size (shared per die)
Memory	64 GB
OS	Debian 8.3
Linux Kernel	4.7.0
GCC	4.9.2 with -O3 optimization

mixture of lookup, insert and delete operations as well as the key range, in the following way:

- Each run lasts 2 seconds, during which each thread performs randomly chosen operations.
- Each software thread is pinned to a hardware thread. Moreover, we only enable hyperthreads when all physical cores have been fully occupied. So executions with less than 44 threads do not have hyperthreading enabled.
- For the transactions of cop-htm and rcu-htm we set the number of retries to 10 and for the transactions of cg-htm to 50. We do so because previous research has shown that coarse-grained HTM tree implementations benefit from many retries [7].
- In order to minimize the overheads of memory allocation we use jemalloc [1] allocator, which is designed to perform better in multi-threaded environments.
- To test our implementations under various contention levels we use three workloads, namely 100-0-0, 80-10-10 and 20-40-40, with 100%, 80% and 20% of operations respectively being lookups in the tree, i.e., read-only traversals, while the rest are equally divided between insertions and deletions. These workloads represent a read-only, read-dominated and write-dominated access pattern on the tree respectively.
- As the key range effectively determines the size of the tree, we evaluate our implementations for ranges of 2K, 20K and 2M keys, which represent medium to large-sized trees. At the start of each run the tree is initialized to contain half the keys of the selected range.
- All reported results are the average of 20 independent executions. We observed no variance in the results of different executions.

[1] http://jemalloc.net/

Figure 6 presents the throughput achieved by the three concurrent AVL implementations for all tree sizes and operation mixes. In all cases rcu-htm outperforms the other two implementations. At best, in the case of 2K keys and 20-40-40 workload, it is 70% better than cop-htm and 220% than cg-htm.

In the read-only workloads, all implementations scale well because there are no data conflicts and transactions are small enough to fit in the hardware HTM buffers, therefore, there are also no capacity aborts. Even in this case, though, rcu-htm provides higher throughput than the other two alternatives. This can be attributed to the fact that in rcu-htm read-only operations impose no overhead at all. On the contrary, both cg-htm and cop-htm require a transaction to be executed, so they suffer the overhead of starting and commiting transactions.

In the read-dominated and write-dominated workloads the trend is similar. In both cases rcu-htm provides better scalability. Despite the fact that insertions and deletions are much slower due to the copying of nodes they include, rcu-htm manages to compensate for that overhead. There are two facts that justify rcu-htm's dominance. First, traversals on the tree are performed without any synchronization and never need to restart, making them extremely fast. Second, the transactions used for insertions and deletions perform only a single write, thus are far less vulnerable to aborts than the transactions of the other two approaches.

Figure 7 presents the number of committed and aborted transactions of the three implementations for all workloads in the case of 2K keys. In all cases rcu-htm executes significantly fewer transactions and, consequently, has less aborts. This fact justifies its performance gain observed in Figure 6. For example, in the read-only workload, even though in both cg-htm and cop-htm all transactions manage to commit, these two implementations have to pay the overhead of starting and committing transactions. Rcu-htm on the other hand avoids this overhead for the read-only operations.

5. Summary and Future Work

In this work we have combined HTM with RCU and implemented highly scalable concurrent balanced BSTs. Our approach manages to outperform previous htm-based concurrent balanced BSTs by as much as 70% and 220%.

For future work, we mainly identify two directions. First, we plan to extend our experimental evaluation with rcu-htm Red-Black and B-trees and compare them with state-of-the-art concurrent BSTs [1–6]. Second, we intent to explore how memory reclamation techniques can be applied to our concurrent trees and the impact they have on their performance.

Acknowledgments

We would like to thank Intel Corporation for kindly providing the server for our experimental evaluation.

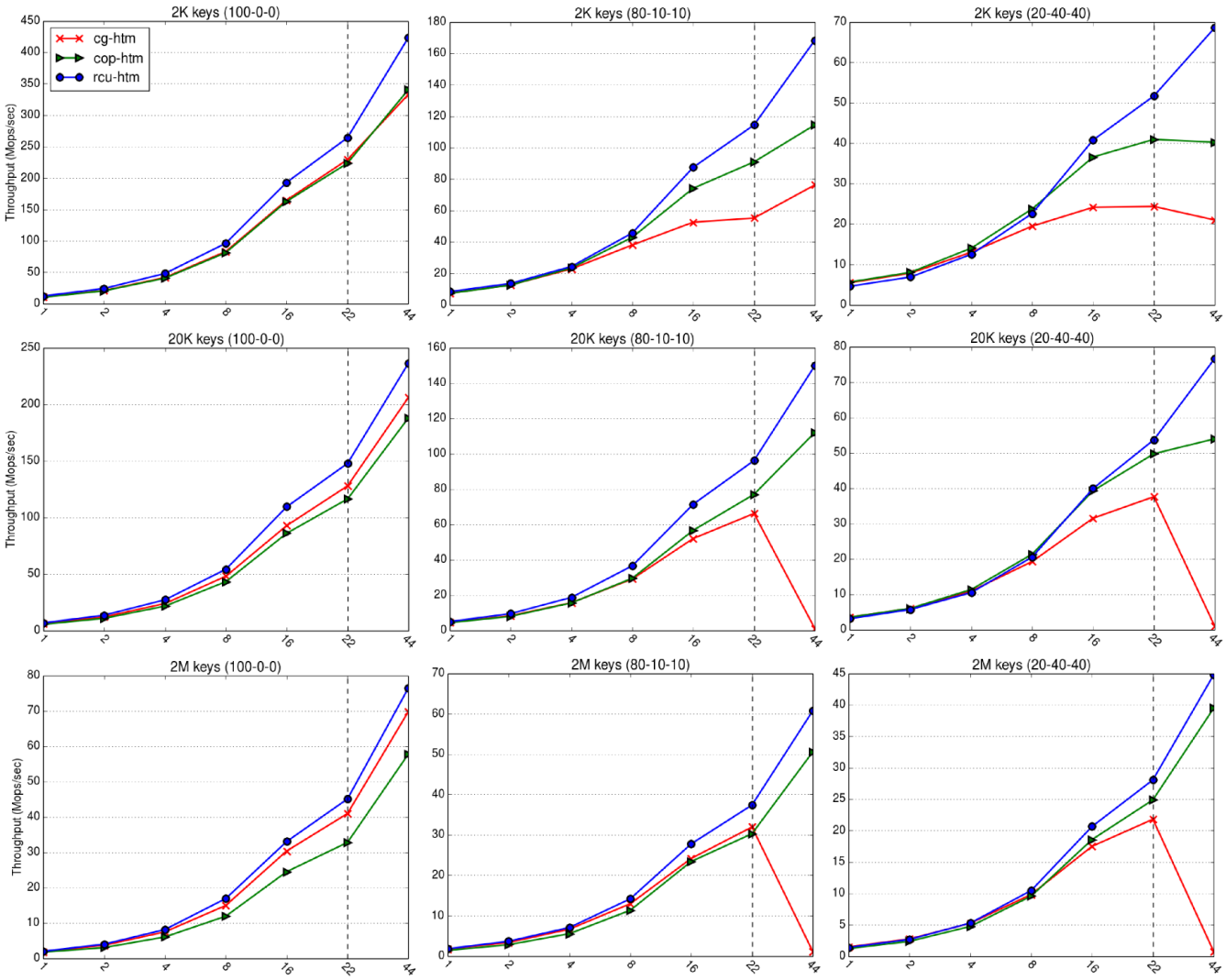


Figure 6. Performance of concurrent AVL implementations. In the x-axis is the number of threads and in the y-axis the throughput in million operations per second. The vertical line in each plot represents the point after which hyperthreading is enabled.

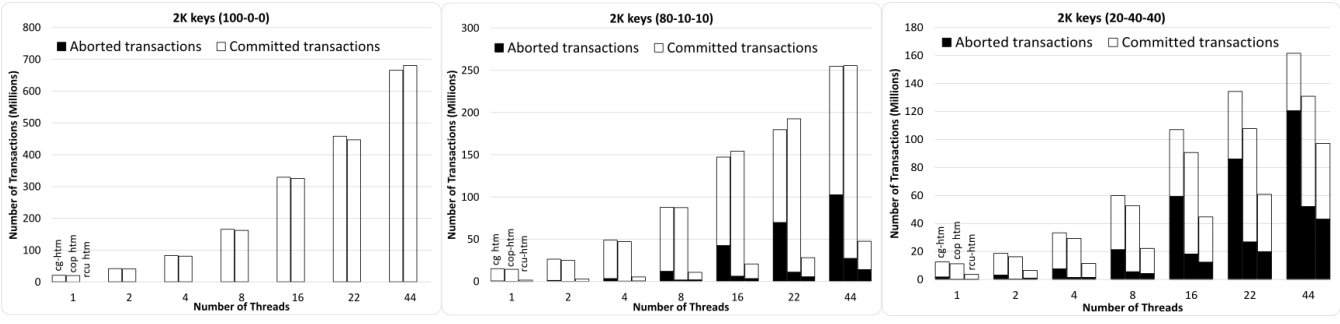


Figure 7. Number of committed and aborted transactions of the three implementations for the 2K tree and all workloads.

References

[1] 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*, PPoPP '14, (New York, NY, USA), pp. 317–328, ACM, 2014.

[2] 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*, SPAA '12, (New York, NY, USA), pp. 161–171, ACM, 2012.

[3] 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*, PODC '10, (New York, NY, USA), pp. 131–140, ACM, 2010.

[4] 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*, PPoPP '10, (New York, NY, USA), pp. 257–268, ACM, 2010.

[5] 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*, PPoPP '14, (New York, NY, USA), pp. 343–356, ACM, 2014.

[6] 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*, PPoPP '14, (New York, NY, USA), pp. 329–342, ACM, 2014.

[7] 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)*, pp. 127–134, Feb 2016.

[8] H. Avni and B. Kuszmaul, "Improving HTM Scaling with Consistency-Oblivious Programming," *TRANSACT*, 2014.

[9] 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), pp. 509–518, Oct. 1998.

[10] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "Scalable Address Spaces Using RCU Balanced Trees," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (New York, NY, USA), pp. 199–210, ACM, 2012.

[11] 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*, PODC '14, (New York, NY, USA), pp. 196–205, ACM, 2014.

[12] 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*, SOSP '15, (New York, NY, USA), pp. 168–183, ACM, 2015.

[13] 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*, ASPLOS '15, (New York, NY, USA), pp. 631–644, ACM, 2015.