Brief Announcement: Efficient Concurrent Range Queries in B+-trees using RCU-HTM

Dimitrios Siakavaras*
Computing Systems Laboratory
School of ECE
National Technical University of Athens
jimsiak@cslab.ece.ntua.gr

Panagiotis Billis*
Computing Systems Laboratory
School of ECE
National Technical University of Athens
pbillis@cslab.ece.ntua.gr

Konstantinos Nikas
Computing Systems Laboratory
School of ECE
National Technical University of Athens
knikas@cslab.ece.ntua.gr

Georgios Goumas
Computing Systems Laboratory
School of ECE
National Technical University of Athens
goumas@cslab.ece.ntua.gr

Nectarios Koziris
Computing Systems Laboratory
School of ECE
National Technical University of Athens
nkoziris@cslab.ece.ntua.gr

ABSTRACT

In this work, we exploit *RCU-HTM*, a synchronization mechanism that combines Read-Copy-Update (RCU) and Hardware Transactional Memory (HTM) to support linearizable and highly efficient range queries in a concurrent B+-tree. Range queries in our B+-tree start with an asynchronized traversal and then perform a horizontal scan of leaf nodes, by following sibling pointers, using hardware transactions. Despite its simplicity, our *RCU-HTM* based B+-tree with range query support greatly outperforms state-of-the-art map data structures for range queries in several execution scenarios.

KEYWORDS

Concurrent data structures, search trees, rcu, htm, rcu-htm

ACM Reference Format:
Dimitrios Siakavaras, Panagiotis Billis, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. 2020. Brief Announcement: Efficient Concurrent Range Queries in B+-trees using RCU-HTM. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '20), July 15–17, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3350755.3400237

1 INTRODUCTION

1.1 RQs in B+-trees

A range query (RQ) operation in a map data structure returns the set of key-value pairs with keys in the range $(lowKey, highKey)$. They are typically met and are highly significant for database and key-value store systems. Maps with RQ support can be implemented with a wide variety of underlying data structures, such as hash tables, singly-linked lists, skiplists, binary search trees, B+-trees, etc. Each data structure has different performance characteristics regarding RQs.

B+-trees are balanced trees with fat nodes, characteristics that make them good candidates for implementing a map with RQ support and for this they are used as indexes in several database management systems and in key-value stores. Moreover, B+-trees are external trees, that is, their data is stored in the leaves and the internal nodes contain only keys and are used for routing the traversals to the appropriate leaves. To facilitate RQs, every leaf contains a sibling pointer to reference its right sibling. RQs start with a traversal to locate the leaf that contains the first key in the range. Then, it horizontally scans the leaves, using the sibling pointers, until a key that is out of the requested range is reached.

1.2 Concurrent RQs in B+-trees

While RQs in a serial version of a B+-tree are simple, in a concurrent setup the correct implementation of an RQ is challenging. Concurrent updaters may modify keys that are in the way of the horizontal scan of the RQ and this may lead to inconsistent execution. An example of an erroneous execution of two RQs, concurrently with two updates is given in Figure 1. Threads T1 and T2 perform an RQ for the same range of keys, $(32 - 54)$. Threads T3 and T4 insert keys 42 and 53, respectively. T1 and T2 follow the same path of leaves, however, the order in which they read the sibling pointers of each leaf causes them to observe a different ordering of the two inserts. RQs that use our approach use an HTM transaction to get a consistent snapshot of this path of leaves. This way they avoid such inconsistent executions.

1.3 The *RCU-HTM* Synchronization Technique

RCU-HTM [8] is a synchronization technique that combines two well known synchronization mechanisms, Read-Copy-Update (RCU) and Hardware Transactional Memory (HTM), and provides a generic method for implementing highly efficient concurrent search trees. In *RCU-HTM*, updaters do not directly modify the nodes of the tree, but they create copies of them and, when their private copy is

*Both authors contributed equally to this research.

SPAA '20, July 15–17, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-6935-0/20/07.
https://doi.org/10.1145/3350755.3400237

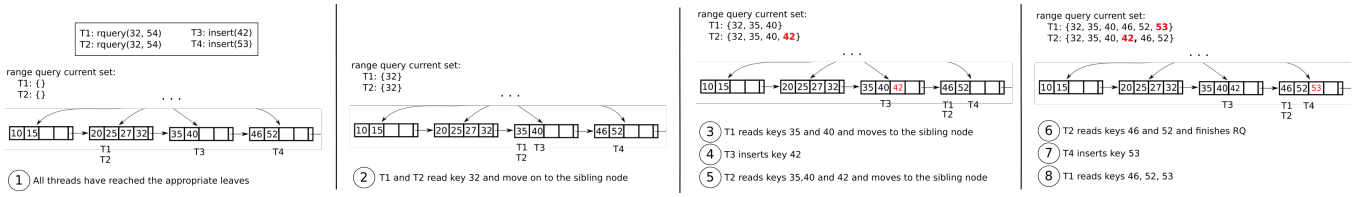

Figure 1: A non-linearizable execution of two RQs that run concurrently with two updates. The two RQ threads observe the two updates in different order. We only show the leaves that are involved in the four operations.

ready, install it in the shared tree by changing the appropriate child pointer of a single node. In an *RCU-HTM* B+-tree the keys (and their associated values) of a node can never be modified and remain stable throughout the whole lifetime of the node. We exploit this characteristic to decrease the size of the HTM transactions used by our RQs.

Algorithm 1: Range Query operation in *RCU-HTM* B+-tree.

```
// Per-thread heap-allocated data
1  __thread int *rquery_keys;
2  __thread void *rquery_values;
3  __thread bptree_node_t *rquery_leaves;
4  int bptree_rcuhtm_rquery (bptree *bpt, int key1, int key2)
5      int nleaves, nkeys;
6      int tx_retries = TX_MAX_RETRIES;
7      bpt_node_t *leaf = bptree_traverse(bpt, key1);
       // If only one leaf is involved we can
       // avoid transactions
8      if key2 < leaf->keys[leaf->nkeys-1] then
9          rquery_leaves[0] = leaf;
10         nkeys = read_keys_from_leaves(key1, key2, 1);
11         return nkeys;
       // First try with HTM transactions
12     while bptree->lock is locked do ; // wait for the lock to be released
13     while tx_retries−− > 0 do
14         if TX_BEGIN() == TM_BEGIN_SUCCESS then
15             if bptree->lock is locked then TX_ABORT();
16             nleaves = get_leaves(leaf, key2);
17             TX_END();
18             break;
       // If necessary, resort to the global lock
19     if tx_retries <= 0 then
20         lock_acquire(bptree->lock);
21         nleaves = get_leaves(leaf, key2);
22         lock_release(bptree->lock);
       // Now we can read the keys from the leaves.
23     nkeys = read_keys_from_leaves(key1, key2, nleaves);
24     return nkeys;
```

2 RQS IN AN *RCU-HTM* B+-TREE

We build on top of an *RCU-HTM* based B+-tree and extend it to support very simple, linearizable and efficient RQs. We exploit the fact that in an *RCU-HTM* B+-tree the keys of a leaf, and their associated values, never change[1]; when a key needs to be added/removed from a leaf, a copy of that node is created and replaces the old one. Based on that characteristic, an RQ can quickly take a snapshot of all the leaves involved in the RQ and then, without the need for synchronization, read all their key-values pairs.

Our RQs proceed in the following way. First, we traverse the tree until we reach the leaf with the lowest key in the range. Then, we start an HTM transaction, which uses the leaves' sibling pointers to locate all the leaves with keys inside the range. During this transaction we only store pointers to these nodes and we do not have to copy them. Inside the transaction we walk the list of sibling pointers and at each leaf we compare the highest key in the requested range with the highest key of the leaf. By doing this, we avoid reading the whole array of keys which, for large node sizes, would result in adding multiple cache lines in the transactional read-set. By reading only the highest key, we add one cache line per leaf, thus we greatly decrease the memory footprint of the transaction.

The C code for the RQ operation in our *RCU-HTM* B+-tree is given in Algorithm 1. We first perform an asynchronized traversal of the tree in line 7 to find the leaf that contains the first key in the requested range (or the leaf that would contain this key, if the key is not present in the map). `bptree_traverse()` performs a typical traversal following child pointers until the appropriate leaf is reached. If the reached leaf contains all the keys in the requested range, we can safely read and return the keys and their associated values without using transactions (lines 8–11). `read_keys_from_leaves()` reads all the leaves in the `rquery_leaves` array and fills `rquery_keys` and `rquery_values` with the key-value pairs that are included in the requested range. If additional leaves need to be scanned, we need to guarantee that the scan is performed atomically with respect to concurrent update operations. We achieve this either with an HTM transaction (lines 12–18) or with a global lock that prevents the execution of concurrent updaters (lines 19–22). In lines 23–24, the array `rquery_leaves` stores pointers to all these leaves that contain keys in the requested key range. Since the keys (and the associated values) inside these leaves can never be modified, we can safely use `read_keys_from_leaves()` to scan these leaves without using any synchronization.

3 EXPERIMENTAL EVALUATION

We conduct our experiments on a dual socket Intel Broadwell-EP server with two Intel Xeon E5-2699 v4 processors each with 22 physical cores and 44 hardware threads, for a total of 44 and 88

[1]This is true for internal nodes as well, but it is irrelevant to our work.

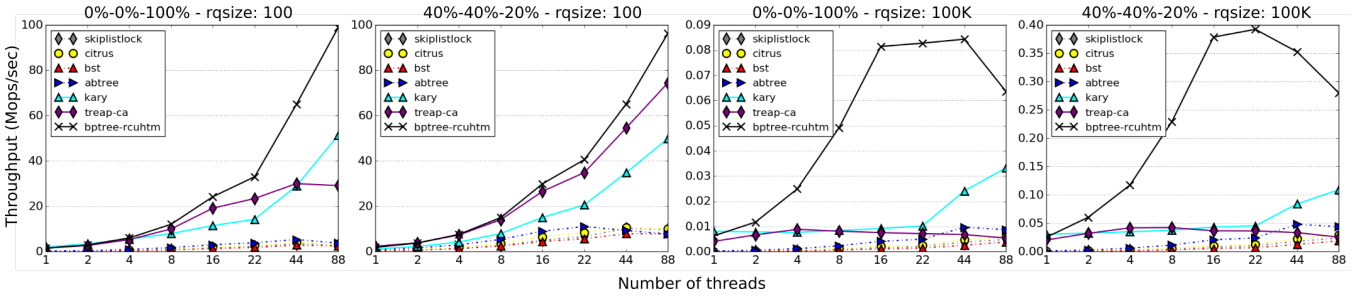

Figure 2: Performance of concurrent maps with RQ support.

physical cores and hardware threads respectively. The processors run at a fixed frequency of 2.2GHz with TurboBoost disabled. Each core has private 32KB L1 and 256KB L2 caches, while a 56MB L3 cache is shared by all cores. The server has 256GB of RAM running at 2134MHz. The OS is Debian 8.3 with kernel version 4.7.0.

We use the C++ version of the benchmark code used in [2] that has been made publicly available in https://bitbucket.org/trbot86/ implementations. Apart from the already provided concurrent maps (i.e., fg-locking skiplist [6], citrus BST [1], lock-free external BST [5] and lock-free external (a,b)-tree [2]), we implemented our *RCU-HTM* based B+-tree as well as the non-blocking k-ary tree [4] and the contention adaptive treap [7]. All implementations were compiled using GCC 4.9.2 with -O3 optimizations enabled.

Our benchmarking methodology consists of two phases. In the warmup phase a single thread inserts random keys until the tree is filled with half of the keys of the key range. In the execution phase we spawn a number of worker threads, which repeatedly perform lookup, update (insert or delete) and RQ operations with randomly selected keys. The execution phase lasts for a predefined time duration, which we currently set to 5 seconds. We have validated that longer time durations produce similar results. We pin each worker thread on a single hardware thread. The first 22 threads occupy the 22 physical cores of a single socket, 44 threads span two sockets and 88 threads use hyperthreads. All reported results are the average of 10 independent executions with no significant variance.

We perform experiments with varying operation mixes and RQ sizes for maps that contain 1M keys. We present the results for four different workloads in Figure 2. The format of the labels are *L%-U%-R% - rqsize: S* where L, U and R are the proportions of lookups, updates and RQs and S is the size of the requested range. Updates are equally divided between inserts and deletes, thus the size of the data structure does not vary significantly throughout the execution.

For the already provided implementations of skiplist, citrus, bst and abtree we present the results of the lock-free RQ provider since this provided the best results. For the (a-b)-trees we set $a = 6$ and $b = 16$, as indicated by the authors in [3] and [2]. This means that a node may contain 6 to 16 keys. For the contention-adaptive treap we set the maximum number of keys in a leaf to 64 as indicated in [7]. For the k-ary tree we have set the k parameter to 32 as indicated in [4]. For our B+-tree, we have set the order of the tree to 8, which means that a node can contain from 8 up to 16 keys.

As depicted in Figure 2, for both the small and the large RQ size, our *RCU-HTM* B+-tree performs better than all the other competitors. In the small RQ size, its high performance is attributed to the low abort ratio for the HTM transactions that RQs execute. In this case, it reaches up to 2x higher throughput than the second best implementation. Even when we execute 40% update operations our B+-tree provides very high performance. For large RQs, none of the competitors manages to scale and provide efficient concurrent RQs. Our B+-tree still provides the best performance despite the large size of its transactions. We observe a performance drop for 44 and 88 threads, which is attributed to NUMA effects due to *RCU-HTM*'s node allocating and copying mechanism which stresses the memory subsystem more than the other implementations.

4 CONCLUSIONS

In this work, we added range query support in a concurrent *RCU-HTM* based B+-tree. Our evaluation revealed that, besides its simplicity, our proposed approach provides high performance under a variety of execution scenarios and outperforms the state-of-the-art concurrent map implementations with RQ support.

REFERENCES

[1] Maya Arbel and Hagit Attiya. 2014. Concurrent Updates with RCU: Search Tree As an Example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC '14)*. ACM, New York, NY, USA, 196–205. https://doi.org/10.1145/2611462.2611471

[2] Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. 2018. Getting to the Root of Concurrent Binary Search Tree Performance. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 295–306. https://www.usenix.org/conference/atc18/presentation/arbel-raviv

[3] Trevor Brown. 2017. A Template for Implementing Fast Lock-free Trees Using HTM. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC '17)*. ACM, New York, NY, USA, 293–302. https://doi.org/10.1145/3087801.3087834

[4] Trevor Brown and Hillel Avni. 2012. Range Queries in Non-blocking *k*-ary Search Trees. In *Principles of Distributed Systems, 16th International Conference, OPODIS 2012, Rome, Italy, December 18-20, 2012. Proceedings*. 31–45. https://doi.org/10.1007/978-3-642-35476-2_3

[5] Trevor Brown, Faith Ellen, and Eric Ruppert. 2014. A General Technique for Non-blocking Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, New York, NY, USA, 329–342. https://doi.org/10.1145/2555243.2555267

[6] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[7] Konstantinos Sagonas and Kjell Winblad. 2016. Efficient Support for Range Queries and Range Updates Using Contention Adapting Search Trees. In *Languages and Compilers for Parallel Computing*, Xipeng Shen, Frank Mueller, and James Tuck (Eds.). Springer International Publishing, Cham, 37–53.

[8] D. Siakavaras, K. Nikas, G. Goumas, and N. Koziris. 2017. RCU-HTM: Combining RCU with HTM to Implement Highly Efficient Concurrent Binary Search Trees. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 1–13. https://doi.org/10.1109/PACT.2017.17