# Leaplist: Lessons Learned in Designing TM-Supported Range Queries

Hillel Avni
Tel-Aviv University
hillel.avni@gmail.com

Nir Shavit
MIT and Tel-Aviv University
shanir@csail.mit.edu

Adi Suissa
Ben-Gurion University
adisuis@cs.bgu.ac.il

## ABSTRACT

We introduce *Leaplist*, a concurrent data-structure that is tailored to provide linearizable range queries. A lookup in *Leaplist* takes O(log n) and is comparable to a balanced binary search tree or to a Skiplist. However, in *Leaplist*, each node holds up-to K immutable key-value pairs, so collecting a linearizable range is K times faster than the same operation performed non-linearizably on a Skiplist.

We show how software transactional memory support in a commercial compiler helped us create an efficient lock-based implementation of *Leaplist*. We used this STM to implement short transactions which we call Locking Transactions (LT), to acquire locks, while verifying that the state of the data-structure is legal, and combine them with a transactional Consistency Oblivious Programming (COP) [2] mechanism to enhance data structure traversals.

We compare *Leaplist* to prior implementations of Skiplists, and show that while updates in the *Leaplist* are slower, lookups are somewhat faster, and for range-queries the *Leaplist* outperforms the Skiplist's non-linearizable range query operations by an order of magnitude. We believe that this data structure and its performance would have been impossible to obtain without the STM support.

## Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*

## Keywords

Transactional-Memory, Data-Structures, Range-Queries

## 1. INTRODUCTION AND RELATED WORK

Consider linearizable concurrent implementations of an abstract dictionary data structure that stores key-value pairs and supports, in addition to the usual Update(key, value), Remove(key), and Find(key), a Range-Query(a, b) operation, where a $\leq$ b, which returns all pairs with keys in the closed interval [a, b], where a and b may not be in the data structure. This type of data structure is useful for various database applications, in particular in-memory databases. This paper is interested in the design of high performance dictionaries with linearizable concurrent range queries. As such, the typically logarithmic search for the first item in the range is not the most important performance element. Rather, it is the coordination and synchronization around the sets of neighboring keys being collected in the sequence. This is a tricky new synchronization problem and our goal is to evaluate which transactional support paradigm, if any, can help in attaining improved performance for range queries.


*PODC'13,* July 22–24, 2013, Montréal, Québec, Canada.


### 1.1 Related Work

Perhaps the most straightforward way to implement a linearizable concurrent version of an abstract dictionary-with-range-queries, is to directly employ software transactional memory (STM) in implementing its methods.[1] An STM allows a programmer to specify that certain blocks of code should be executed atomically relative to one another. Recently, several fast concurrent binary search-tree algorithms using STM have been introduced by Afek et al. [2] and Bronson et al. [4]. Although they offer good performance for Updates, Removes and Finds, they achieve this performance, in part, by carefully limiting the amount of data protected by the transactions. However, as we show empirically in this paper, computing a range query means protecting all keys in the range from modification during a transaction, leading to poor performance using the direct STM approach.

Another simple approach is to lock the entire data structure and compute a range query while it is locked. One can refine this technique by using a more fine-grained locking scheme, so that only part of the data structure needs to be locked to perform an update or compute a range query. For instance, in leaf-oriented trees, where all key-value pairs in the set are stored in the leaves of the tree, updates to the tree can be performed by local modifications close to the leaves. Therefore, it is often sufficient to lock only the last couple of nodes on the path to a leaf, rather than the entire path from the root. However, as was the case for STM, a range query can only be computed if every key in the range is protected, so typically every node containing a key in the range must be locked.

Brown and Avni [5] introduced range queries in k-ary trees with immutable keys. The k-ary trees allow efficient range-

[1]STM is now in the mainline GCC compiler. Unfortunately, mature hardware TM is still unavailable.

queries by collecting nodes in a depth-first-search order, followed by a validation stage. The nodes are scanned, and if any node is outdated, the process is retried from the start. Although this is an efficient solution, it is not practical, as the k-ary search tree is not balanced.

Prokopec et al. [10] presented Ctrie which is a non-blocking concurrent hash trie that offers O(1) time snapshot. Keys are hashed, and the bits of these hashes are used to navigate the trie. To facilitate the computation of fast snapshots, a sequence number is associated with each node in the data structure. Each time a snapshot is taken, the root is copied and its sequence number is incremented. An update or search in the trie reads this sequence number *seq* when it starts and, while traversing the trie, it duplicates each node whose sequence number is less than *seq*. The update then performs a variant of a double-compare-single-swap operation to atomically change a pointer while ensuring the root's current sequence number matches *seq*. Because keys are ordered by their hashes in the trie, it is hard to use Ctrie to efficiently implement range queries. To do so, one must iterate over all keys in the snapshot.

The B-Tree data structure can be used for range queries, however, when looking at the concurrent versions of B-Trees such as the lock-free one of Braginsky and Petrank [3], and the blocking, industry standard from [12], both do not support the range-query functionality. Both algorithms do not have leaf-chaining, forcing one to perform a sequence of lookups to collect the desired range. In [12] this would imply holding a lock on the root for a long time, and in [3] it seems difficult to get a linearizable result. In addition, the keys in both are mutable so one would have to copy each entry individually.

## 1.2 The Leaplist in a Nutshell

*Leaplists* are Skiplists [11] with "fat" nodes and an added shortcut access mechanism in the style of the String B-tree of Ferragina and Grossi [7]. They have the same probabilistic guarantee for balancing, and the same layered forward pointers as Skiplists. Each *Leaplist* node holds up to K immutable keys from a specific range, and an immutable bitwise trie is embedded in each node to facilitate fast lookups when K is large.

When considering large range queries, the logarithmic-time lookup for the start of the range accounts for only a small part of the operation's complexity. Especially when the whole structure resides in memory. The design complexity of a full k-ary structure (in which nodes at all levels have K elements), with $\log_k(n)$ lookup time is thus not justified. In our *Leaplist*, unlike full k-ary structures, an update implies at most one split or merge of a node, and only at the leaf level. This allows updates to lock only the specific leaf being changed and only for the duration of changing pointers from the old node to the new one.

For *Leaplist* synchronization, we checked the following options, sorted in an increasing order of required effort:

- **Pure STM:** We tried to put each *Leaplist* operation in a software transactional memory (STM) transaction. This option was especially attractive with the rising support for STM in mainstream compilers. Unfortunately, as we report, we discovered that this aproach introduced unacceptable overheads.

- **Read-write locks:** We explored read-write locks per *Leaplist*. The read-locks were unscalable in NUMA executions, while the write locks serialized many workloads.

- **COP:** We employed consistency oblivious programming (COP) [2] to reduce the overhead of STM. In COP, the read-only prefix of the operation is executed without any synchronization, followed by an STM transaction that checks the correctness of the prefix execution and performs any necessary updates. The COP requires that an un-instrumented traversal of the structure will not crash, which implies strong isolation of transactions in the underlying STM. Otherwise the traversal encounters uncommitted data, and hitting uncommitted data inevitably leads to uninitialized pointers, unallocated buffers, and segmentation faults. The current GCC-TM compiler uses weakly isolated transactions [6], i.e., a non-transactional read operation may see the state of an incomplete transaction. Thus, we had to add transactions also in read-only regions of the code which hurt performance. As argued in [6], weak isolation TM implementations will likely have less overhead than a strong isolation TM implementation, and better performance.

- **Locking Transactions (LT):** With LT, transactions are used only to acquire locks, and not to write tentative data. Thus, a read which sees unlocked data knows it is committed. Another aspect of LT, is that using a short transaction anyone can lock any data and use it.

  We use LT to improve the performance of the previous COP algorithm. In the COP, an updating operation performs its read-only prefix without synchronization, and then executes the verification and updates inside a transaction. In LT, the read-only part is checking for locks, and retries. These checks have negligible overhead compared to a transaction. Then the transaction atomically verifies validity and locks the written addresses. After the transaction commits, a postfix of the operation writes the data to the locked address locations and releases them.

- **Fine grained locks:** To generate the fine grained version of LT *Leaplist* we had to recreate mechanisms that exist in STM, and still, did not manage to create a correct and efficient implementation.

  In case of a merge, where a remove replaces two old nodes by one new node, we need to lock all pointers to and from both nodes. Here, unlike the skiplist case [9], locking can fail at any point and force us to release all locks and retry to avoid deadlocks. This unrolling is "free" using an STM.

  Once a set of nodes is locked, a thread needs to perform validations on the state of the data structure, such as checking live marks etc. With LT, using STM, these validations happen before acquiring the locks, and then when committing, an abort will happen if any check should fail. Thus the locks are taken for a shorter duration. To improve our performance we would need to execute a form of STM revalidation.

  After executing the above sequence, we found that our fine grained implementation still suffered from live-

locks; we did not manage to avoid them. These livelocks were eliminated with the STM based LT approach.

Our conclusion was that we were effectively reproducing the very mechanisms that are already given by an STM, and still did not get the stability of an STM. The LT *Leaplist* implementation has minimal overhead because lookups do not execute transactions and range-queries execute one instrumented access per K values in the range. The LT *Leaplist* is thus the most effective solution.

An added value for using TM based synchronization is that we can compose operations on multiple *Leaplists* into one atomic transaction. In Section 2, we discuss the design of operations that support atomic access of multiple *Leaplists*, and in Section 3, we evaluate their performance.

This paper is organized as follows. Section 2 gives a detailed description of the *Leaplist* design and operations' implementation. In Section 3 we show the LT technique is the best performer for *Leaplist* synchronization, and is scalable even when transactions encompass operations on multiple *Leaplists*. Finally, in Section 4 we summarize our work, and give some directions for future work.

# 2. LEAPLIST DESIGN

We now describe the detailed design of our *L-Leaplists* data structure. Note that the updating functions compose operations on multiple *Leaplists*. Our implementation supports the following operations:

- **Update**($ll$, $k$, $v$, $s$) - Receives arrays of *Leaplists*, keys and values of size $s$, and updates the value of the key $k[i]$ to be $v[i]$ in *Leaplist* $ll[i]$. If the key $k[i]$ is not present in $ll[i]$, it is added to $ll[i]$ with the given value $v[i]$.
- **Remove**($ll$, $k$, $s$) - Receives arrays of *Leaplists* and keys of size $s$, and removes the key-value pair of the given key $k[i]$ from $ll[i]$.
- **Lookup**($l$, $k$) - Receives a single *Leaplist* and a key, and returns the value of the corresponding given key $k$ in $l$. The operation returns an indication in case the key is not present in $l$.
- **Range-Query**($l$, $k_{from}$, $k_{to}$) - Receives a single *Leaplist* and 2 keys, and returns the values of all keys in $l$ which are in the range $[k_{from}, k_{to}]$.

The *Update* and *Remove* operations are applied to $L$ *Leaplists* which allows concurrent operations on multiple database table indexes. We do this to demonstrate that the implementation of *Leaplist* with TM allows composing its operations.

## 2.1 Leaplist Data-Structure

The *Leaplist* node holds a *live* mark, which is used in COP verification stage; *high*, which bounds its keys range; *count*, which is the number of key-value pairs present in the node, and *level* which is the same as a level in Skiplist. It also holds an array of forward pointers *next* each pointing to the next element in the corresponding level. A *trie* is used to quickly find the index of key $k$ in the keys-values array, a

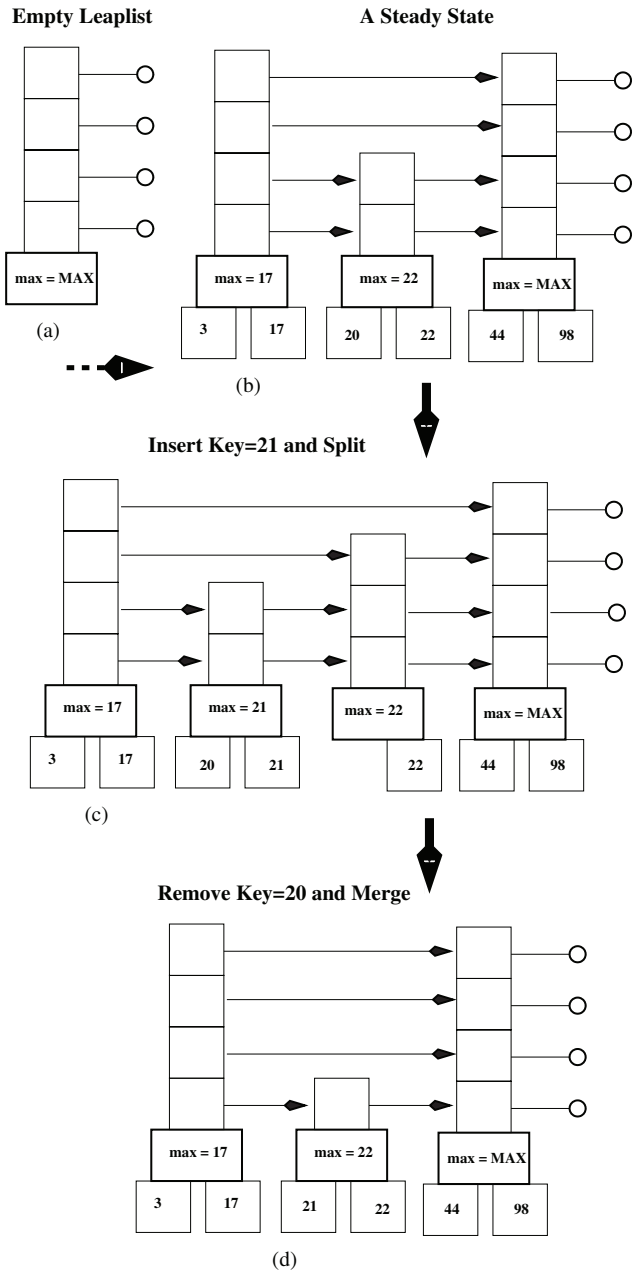



Figure 1: A single *Leaplist* with maximum height of 4 and node size of 2. Each node is composed of a column of up to 4 pointers, below it a square with the high key of that node, and in the bottom, the keys.
Initial state (a) is empty, with all pointers pointing to **null**. (b) is some arbitrary state of the *Leaplist*. In (c) an Insert causes a split and in (d) a Remove yields a merge.
State (b) is long after the initial state (a), but states (b), (c) and (d) are consecutive, i.e., separated by one operation.

*Leaplist* Search Predecessors

```
   input  : Leaplist l, key k
   output: Two node arrays of pointers - pa and na
 1 node *x, *x_next;
 2 int i;
 3 retry:
 4 x := l;
 5 for i = max_level- 1; i ≥ 0; i = i - 1 do
 6     while (true) do
 7         x_next := x→next[i];
 8         if MARKED(x_next) ∨ (¬x_next→live) then
           goto retry;
 9         if x_next→high ≥ k then
10             break;
11         else
12             x := x_next
13         end
14     end
15     pa[i] := x;
16     na[i] := x_next;
17 end
18 return (pa, na);
```

Figure 2: *Leaplist* Search Predecessors

technique introduced in the String B-tree of Ferragina and Grossi [7]. Note that unlike in a Skiplist, where each node represents a single key, in *Leaplist* each node represents a range of keys, i.e. all the keys from a certain range. The *keys-values* array of size *count* holds all the keys and their corresponding values in the node. The trie uses the minimal number of levels to represent all the keys in the node, where the lowest level is comprised of indexes of the keys' values in the *keys-values* array.

In *Leaplist*, a node's keys-values array is immutable, and never changes after an update. We do this to support consistent range-query operations.

When the key or value (and possibly the encompassed range) of a node is updated (due to an update or a remove operation), that node is replaced by a newer node with the modified keys-values array. If the node is full (i.e., the number of keys in the node reaches some predefined number), it is split into two consecutive nodes and the upper bound of the lower node is determined by the highest value in it. An example is Figure 1(c), where key 21, is inserted into a node that is already full in Figure 1(b), and the node is split.

In case the modified node and its subsequent node are sparse (the number of keys in both nodes is less than some predefined number), the nodes are merged into a single node. This scenario is demonstrated in Figure 1(d), where key 20 is deleted, and two consecutive nodes are merge.

In the rest of this section we describe the *Leaplist* functions.

### 2.1.1 Searching for Predecessors

The search predecessors function from Figure 2 receives a key $k$, and traverses the *Leaplist* until the node $N$ (that encompasses the range where key $k$ is included) is reached. The function returns two arrays of nodes, *pa* and *na*, each of size **max_level**. The *pa* array includes all the nodes that "immediately precede" node $N$. That is for each level $i$ up to $N$'s level, $pa[i]\rightarrow next[i]$ points to $N$, and for levels higher than $N$'s level, the nodes that encompass keys that are smaller than $k$ and their next pointer at level $i$ points to a node with higher keys than $k$. The *na* array includes all the nodes that are adjacent to the *pa* nodes, and encompass keys that are greater-than-or-equal-to $k$ (thus $na[i]\rightarrow next[i]$ is $N$ for all levels up to $N$'s level). This function is used in the lookup and range-query operations, as well as in the beginning of the update and remove operations.

The traversal only compares the *high* key of the node in line 9 and decides if it should continue or stop at that node. When reading a pointer, the thread verifies that that pointer is not marked and that the node is still live in line 8, so it only traverses committed and valid nodes. (As previously noted, an alternative method would be replacing the mark by executing line 7 in a transaction. However, with the current GCC-TM implementation the overhead of starting a transaction is too high. We estimate that with HTM this would work much better, and will actually make the lookup wait-free, as a single-location read transaction must succeed.)

*Leaplist* Lookup

```
   input  : Leaplist l, key k
   output: Value or ⊥
19 node *na[max_level ];
20 (null, na)←PredecessorsSearch(l,k);
21 return (na[0]→values[get_index(na[0]→trie,k)].value);
```

Figure 3: *Leaplist* Lookup

### 2.1.2 Lookups

The lookup operation is presented in Figure 3, and is using the predecessors search function. Note that the node returned in $na[0]$ is the node that has $k$ in its range. We can prove the lookup is linearizable, as the predecessors search traverses only committed nodes. If a thread searches for the key $k$, it must traverse a node that $k$ is in its range, and if such a live node is reached, then this node was present in the data-structure during the lookup execution.

In line 21, *Lookup* uses the node's trie to extract the index of the value of key $k$ in the array values, and returns the value from that index.

### 2.1.3 Range Queries

The range query operation is presented in Figure 4, and starts with a predecessors search to find the node where the range starts from. Then, within a transaction, it first checks that the node is still live in line 30 and if not aborts, and retries the range-query operation in line 36. If the node is still marked as live, the transaction traverses the lowest level of the *Leaplist*'s pointers from the first node to the node which has a *high* value which is higher than the requested range high bound, and retrieves a snapshot range query. Note that in line 32 the algorithm ensures that even in the case of a partial update to the pointer to the next node (due to update or remove operations), it can still traverse through it.

### 2.1.4 Updates

Figure 5 describes the update function. As previously described, the function receives arrays of *Leaplists*, keys and values, and their size. The update operation either inserts a

```
Leaplist Range Query
   input  : Leaplist l, key low, key high
   output: Set S of nodes
22 node *na[max_level ], *n;
23 boolean committed ← false;
24 retry:
25 S← ∅;
26 (null, na)←Search(l,low);
27 n := na[0];
28 tx_start;
29 while n→high<high do
30     if ¬n→live then tx_abort;
31     add(S,na[0]);
32     n := unmark((n→next[0]));
33 end
34 committed := true;
35 tx_end;
36 if ¬committed then goto retry;
37 return S;
```

Figure 4: *Leaplist* Range Query

new key-value pair to each *Leaplist* if the key is not already present, or otherwise updates the key's value.

The function is divided into the following 3 parts: (1) setup (Figure 6), (2) LT (Figure 7), and (3) release and update (Figure 8). During the setup part, a thread iterates over each *Leaplist*, performs a predecessors search, and creates a new node with its key-value pairs (including the updated key-value pair). Note that in case the number of keys in the node is above some threshold, it *splits* that node. During a split it creates 2 nodes: one with a new random height that holds the first half of the key-value pairs, and another with the same height as the old node that holds the second half of the key-value pairs. The *max_level* is set to the maximum between the heights of the two nodes. The *CreateNewNodes* function updates the new node (nodes) with its (their) key-value pairs.

The LT part is executed in a single transaction. The algorithm again iterates over each *Leaplist* and first verifies that the updated node is still live (line 71), that all the predecessors' next pointers point to that node, and that the next pointers from that node are still valid (lines 72-80). (In case of a split, the algorithm also verifies this up to the *max_level* height.) In lines 81-87 it continues to verify and mark the pointers to the node and from the node, and in case of a split the nodes to and from the nodes up to *max_height*. Note that if one of the conditions does not hold, the transaction is aborted, and the whole operation restarts. It finishes the transaction by setting the old node's live bit to false (line 89), and attempting to commit the transaction. We note that in this part, the transaction does not observe partial modifications made by other transactions, and so a successful commit ensures a consistent view of the nodes that are affected by the operation.

Following a successful transaction commit, the third part releases and updates the pointers of the predecessor nodes to point to the new node (nodes). In lines 92-113 the algorithm sets the next pointer of the new node (nodes) to the previous nodes that were in the *Leaplist*. It continues by setting the next pointers of the predecessor nodes to the new node (nodes) in lines 115-121, and finishes by setting the live flags of the new node (nodes) to true.

```
Leaplists Update
   input  : Leaplists ll, keys k, values v, and size s
38 node *pa[max_lists ][max_level ], *na[max_lists
   ][max_level ], *n[max_lists ];
39 node *new_node[max_lists ][2];
40 int max_height[max_lists ];
41 boolean committed := false, split[max_lists ];
42 foreach j<s do
43     new_node[j][0] := new node;
44     new_node[j][1] := new node;
45 end
46 retry:
47 Update_Setup(ll, k, v, s, pa, na, n, new_node,
   max_height, split);
48 tx_start ;
49 Update_LT(s, pa, na, n, new_node, max_height);
50 committed := true;
51 tx_end;
52 if ¬committed then goto retry;
53 Update_Release_and_Update(s, pa, na, n, new_node,
   split);
54 Deallocate unneeded nodes.
```

Figure 5: *Leaplist* Update

### 2.1.5 Remove

The remove function is presented in Figure 9. The function receives arrays of Leaplists, keys and their size, and linearizably removes the key-value pair of each given key from its corresponding Leaplist. In case a key is not found in a Leaplist, that Leaplist is not modified.

Similarly to the update function, the remove function is also divided to the setup (Figure 10), LT (Figure 11) and release and update (Figure 12) parts. During the setup part, the thread again iterates over each Leaplist, performs a predecessors search, and searches for the key to be removed. If a Leaplist does not contain the corresponding key, it moves on to the next Leaplist. In case the key exists it keeps the node that holds the key and its successor node in the *old_node* variables (line 145-152). The node and its adjacent node are merged if the sum of the key-value pairs in both nodes is below some threshold. It then verifies that the node and the adjacent node (upon merge) are live, and if not, the retry of the last key removal from the current Leaplist is performed. The thread concludes this part by calling *RemoveAndMerge* which updates a new node with the key-value pairs from the node (and the adjacent node), without the removed key-value pair.

The second part, the LT, is performed in a single transaction. In this part the thread first verifies the nodes that were found in the setup part are still valid (i.e., they are still live), their successive nodes are still live, and the pointers from their predecessors point to them. If one of the conditions does not hold, the transaction is aborted, and the whole remove operation is restarted. It then continues to mark the next pointers of the nodes that are about to be removed, and the next pointers of their predecessors. The transaction concludes by setting the live bit of the nodes to false, and attempts to commit. In case the commit fails, the remove operation is retried from the beginning of the setup part.

However, if the transaction successfully commits, the third

```
Leaplist Update - Setup
   input  : Leaplists ll, keys k, values v, size s, nodes pa,
            nodes na, nodes n, nodes new_node, integers
            max_height, booleans split
55 foreach j<s do
56     (pa[j],na[j])←PredecessorSearch(ll[j],k[j]);
57     n[j] := na[j][0];
58     if n[j]→count = node_size then
59         split[j] := true;
60         new_node[j][1]→level := n[j]→level;
61         new_node[j][0]→level := get_level();
62         max_height[j] := max(new_node[j][0]→level,
           new_node[j][1]→level);
63     else
64         split[j] := false;
65         new_node[j][0]→level := n[j]→level;
66         max_height[j] := new_node[j][0]→level;
67     end
68     CreateNewNodes(new_node[j], n[j], k[j], v[j], split[j]);
69 end
```

Figure 6: *Leaplist* Update - Setup

part releases and updates each Leaplist to include its new node. It first sets the next pointers of the new node to point to the unmarked removed nodes next pointers in lines 208-218. Following this we set the next pointers of the old nodes pointers to the new node (lines 220-221). It concludes, in line 223, by setting the new nodes live bit.

# 3. EVALUATION

In this section we present the evaluation of our *Leaplist* implementation using COP and the LT technique and compare it to an STM-based *Leaplist*, an STM based *Leaplist* implementation that uses only COP, and a RW-Lock *Leaplist* implementation that uses a reader-writer lock. In Section 3.1 we compare to *Skiplist* implementations.

**Experimental setup:** We collected results on a machine powered by four Intel E7-4870. An Intel E7-4870 is a chip multithreading (CMT) processor, with 10 2.4 GHz cores each multiplexing 2 hardware threads, for a total of 20 hardware strands per chip. All implementations were compiled using GCC version 4.7 [1] which has built-in support for transactional memory. We used the linearizable memory allocation manager which was proposed in [8]. We compared the throughput (operations per second) of the following four algorithms:

1. **Leap-LT** - our proposed algorithm that uses COP and the LT technique as described in Section 2.

2. **Leap-tm** - a *Leaplist* implementation which wraps each operation within a transaction.

3. **Leap-COP** - an STM-based Leaplist implementation that uses COP (separating the search and update/remove operation).

4. **Leap-rwlock** - A Read-Write lock *Leaplist* implementation, in which the lookup and range-query operations acquire the read-lock, and the update and remove operations acquire the write-lock.

```
Leaplist Update - LT
   input  : size s, nodes pa, nodes na, nodes n, nodes
            new_node, integers max_height
70 foreach j<s do
71     if ¬n[j]→live then tx_abort;
72     foreach i<n[j]→level do
73         if pa[j][i]→next[i]≠n[j] then tx_abort;
74         if ¬n[j]→next[i]→live then tx_abort;
75     end
76     foreach i<max_height[j] do
77         if pa[j][i]→next[j][i]≠na[j][i] then tx_abort;
78         if ¬pa[j][i]→live then tx_abort;
79         if ¬na[j][i]→live then tx_abort;
80     end
81     foreach i<n[j]→level do
82         if MARKED(n[j]→next[i]) then tx_abort;
83         n[j]→next[i] := MARK(n[j]→next[i]);
84     end
85     foreach i<max_height[j] do
86         if MARKED(pa[j][i]→next[i]) then tx_abort;
87         pa[j][i]→next[i] := MARK(pa[j][i]→next[i]);
88     end
89     n[j]→live := false ;
90 end
```

Figure 7: *Leaplist* Update - LT

**Settings:** We compared different mixtures of update, remove, lookup and range-query operations using the above algorithms on 4 *Leaplists* (i.e., the size of the arrays on update and remove operations is 4). Each *Leaplist* is configured with a node of size 300, and with a maximal level of 10. We experimentally found these values achieve good performance. Each experiment execution is set to 10 seconds, and is repeated three times. We show the average of the three results. We now present the throughput of the above algorithms using various workload configurations. The keys range between 0 to 100000, and a range-query operation range spans a random range between 1000 to 2000.

Figure 13 exhibits the throughput of the different algorithms when varying the number of threads from 1 to 80. In this scenario each *Leaplist* is initialized with 100,000 successive elements. The write-only case, 100% modifications (only updates and removes), is presented in Figure 13-(a). We observe that the throughput of the *Leap-LT* is better than all other algorithms, and scales well up to 32 threads. It achieves up to 220%, 355%, and 930% better throughput compared with the *Leap-COP*, *Leap-tm*, and *Leap-rwlock* algorithms respectively. This shows that even under an extreme write-dominated workload, our algorithm still performs well.

In Figure 13-(b) we present a read-dominated case with a mixture of 40% lookups, 40% range-queries and 20% modifications. *Leap-LT* scales up to 40 threads because there are less modifications. Compared with the *Leap-COP*, *Leap-tm*, and *Leap-rwlock* algorithms it achieves up to 200%, 330%, and 980% better throughput respectively. When comparing the absolute throughput values, one can see that the read-dominated workload has a higher throughput than the write-only workload. This is because a higher modifications rate incurs a high overhead of update and remove operations

Leaplist Update - Release and Update

```
   input : size s, nodes pa, nodes na, nodes n, nodes
           new_node, booleans split
91 foreach j<s do
92     if split[j] then
93         if new_node[j][1]→level > new_node[j][0]→level
           then
94             foreach i<new_node[j][0]→level do
95                 new_node[j][0]→next[i] := new_node[j][1];
96                 new_node[j][1]→next[i] :=
                   UNMARK(n[j]→next[i]);
97             end
98             foreach
               new_node[j][0]→level≤i<old_node[j][1]→level
               do
99                 new_node[j][1]→next[i] :=
                   UNMARK(n[j]→next[i]);
100            end
101        else
102            foreach i<new_node[j][1]→level do
103                new_node[j][0]→next[i] := new_node[j][1];
104                new_node[j][1]→next[i] :=
                   UNMARK(n[j]→next[i]);
105            end
106            foreach
               new_node[j][1]→level≤i<old_node[j][0]→level
               do
107                new_node[j][0]→next[i] :=
                   UNMARK(na[j][i]);
108            end
109        end
110    else
111        foreach i<new_node[j][0]→level do
112            new_node[j][0]→next[i] :=
               UNMARK(n[j]→next[i]);
113        end
114    end
115    foreach i<new_node[j][0]→level do
116        pa[j][i]→next[i] := new_node[j][0];
117    end
118    if split[j] ∧ (new_node[j][1]→level >
       new_node[j][0]→level) then
119        foreach
           new_node[j][0]→level≤i<old_node[j][1]→level do
120            pa[j][i]→next[i] := new_node[j][1];
121        end
122    end
123    new_node[j][0]→live := true;
124    if split[j] then new_node[j][1]→live := true;
125 end
```

Figure 8: *Leaplist* Update - Release and Update

(compared to the lookup operations), and increased number of conflicts and retries.

Figure 14 shows the performance of the algorithms while varying the number of elements each *Leaplist* is initialized with, and setting the number of threads to 80. (The x-axis is log-scaled). We observe that when there are only update and remove operations (Figure 14-(a)), the highest throughput is achieved when a *Leaplist* is initialized with 1,000,000 elements. This is because there are less conflicts due to

Leaplist Remove

```
   input : Leaplists ll, keys k, size s
126 node *pa[max_lists ][max_level ], *na[max_lists
    ][max_level ], *n[max_lists ];
127 node *old_node[max_lists ][2];
128 boolean committed := false, merge[max_lists ],
    changed[max_lists ];
129 foreach j<s do
130     n[j] := new node;
131 end
132 retry_all:
133 Remove_Setup(ll, k, v, s, pa, na, n, old_node, merge,
    changed);
134 tx_start;
135 Remove_LT(s, pa, na, n, old_node, merge, changed);
136 committed := true;
137 tx_end ;
138 if ¬committed then goto retry_all;
139 Remove_Release_and_Update(s, pa, na, n, old_node,
    merge, changed);
140 Deallocate unneeded nodes.
```

Figure 9: *Leaplist* Remove

Leaplist Remove - Setup

```
   input : Leaplists ll, keys k, values v, size s, nodes pa,
           nodes na, nodes n, nodes old_node, booleans
           merge, booleans changed
141 foreach j<s do
142     int total;
143     retry_last: merge[j] := false;
144     (pa,na)←PredecessorSearch(ll[j],k[j]);
145     old_node[j][0] := na[j][0];
146     if get_index(old_node[j][0]→trie,k[j]) =
        NOT_FOUND then
147         changed[j] := false;
148         continue;
149     end
150     repeat
151         old_node[j][1] := old_node[j][0]→next[0];
152         if ¬ then goto retry_last;
153     until ¬is_marked(old_node[j][1]) ;
154     total := old_node[j][0]→count;
155     if old_node[j][1] then
156         total += old_node[j][1]→count;
157         if total≤node_size then merge[j] := true;
158     end
159     Set n[j] level, count, high and low;
160     if ¬old_node[j][0]→live then goto retry_last;
161     if merge[j] ∧ ¬old_node[j][1]→live then goto
        retry_last;
162     changed[j] := RemoveAndMerge(old_node[j], n[j],
        k[j], merge[j]);
163 end
```

Figure 10: *Leaplist* Remove - Setup

the high number of nodes. Note that when the number of elements is higher, the overhead stems from the long predecessors search operation. In Figure 14-(b) we see that when there are only lookup operations, the highest throughput is achieved when the number of elements is 10,000. This is

```
Leaplist Remove - LT
   input : size s, nodes pa, nodes na, nodes n, nodes
           old_node, booleans merge, booleans changed
164 foreach j<s do
165     if changed[j] then
166         if ¬old_node[j][0]→live then tx_abort;
167         if merge[j] ∧ ¬old_node[j][1]→live then
            tx_abort;
168         foreach i<old_node[j][0]→level do
169             if pa[j][i]→next[i]≠old_node[j][0] then
                tx_abort;
170             if ¬pa[j][i]→live then tx_abort;
171             if ¬old_node[j][0]→next[i]→live then
                tx_abort;
172         end
173         if merge[j] then
174             if old_node[j][0]→next[0]≠old_node[j][1] then
                tx_abort;
175             if old_node[j][1]→level > old_node[j][0]→level
                then
176                 foreach i<old_node[j][0]→level do
177                     if ¬old_node[j][1]→next[i]→live then
                        tx_abort;
178                 end
179                 foreach old_node[j][0]→level ≤ i <
                    old_node[j][1]→level do
180                     if pa[j][i]→next[i]≠old_node[j][1] then
                        tx_abort;
181                     if ¬pa[j][i]→live then tx_abort;
182                     if ¬old_node[j][1]→next[i]→live then
                        tx_abort;
183                 end
184             else
185                 foreach i<old_node[j][1]→level do
186                     if ¬old_node[j][1]→next[i]→live then
                        tx_abort;
187                 end
188             end
189             foreach i<old_node[j][1]→level do
190                 if MARKED(old_node[j][1]→next[i])
                    then tx_abort;
191                 old_node[j][1]→next[i] :=
                    MARK(old_node[j][1]→next[i]);
192             end
193         end
194         foreach i<old_node[j][0]→level do
195             if MARKED(old_node[j][0]→next[i]) then
                tx_abort;
196             old_node[j][0]→next[i] :=
                MARK(old_node[j][0]→next[i]);
197         end
198         foreach i<n[j]→level do
199             if MARKED(pa[j][i]→next[i]) then tx_abort;
200             pa[j][i]→next[i] := MARK(pa[j][i]→next[i]);
201         end
202         old_node[j][0]→live := false;
203         if merge[j] then old_node[j][1]→live := false;
204     end
205 end
```

Figure 11: *Leaplist* Remove - LT

```
Leaplist Remove - Release and Update
   input : size s, nodes pa, nodes na, nodes n, nodes
           old_node, booleans merge, booleans changed
206 foreach j<s do
207     if changed[j] then
208         if merge[j] then
209             foreach i<old_node[j][1]→level do
210                 n[j]→next[i] :=
                    UNMARK(old_node[j][1]→next[i]);
211             end
212             foreach
                old_node[j][1]→level≤i<old_node[j][0]→level
                do
213                 n[j]→next[i] :=
                    UNMARK(old_node[j][0]→next[i]);
214             end
215         else
216             foreach i<old_node[j][0]→level do
217                 n[j]→next[i] :=
                    UNMARK(old_node[j][0]→next[i]);
218             end
219         end
220         foreach i<n[j]→level do
221             pa[j][i]→next[i] := n[j];
222         end
223         n[j]→live := true ;
224     end
225 end
```

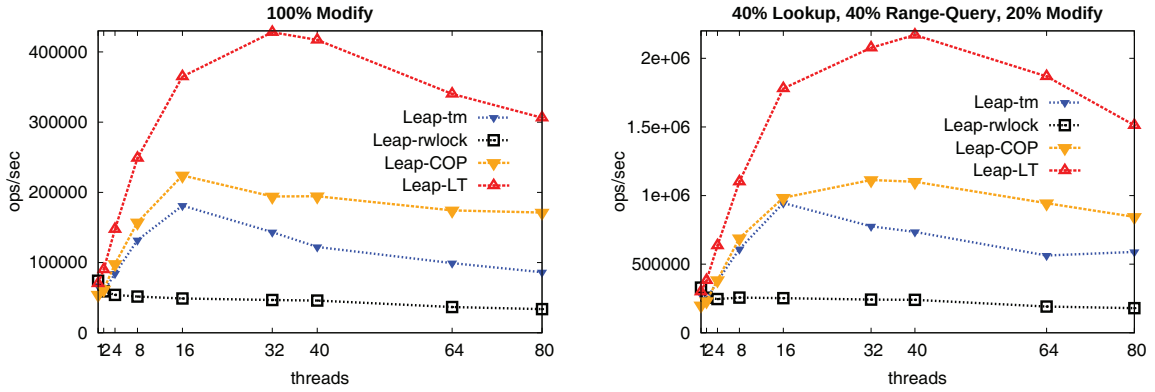
Figure 12: *Leaplist* Remove - Release and Update

again due to the long predecessors search operations when the number of nodes is larger.

Figure 15-(a) and Figure 15-(b) depict the throughput when using 80 threads, a *Leaplist* with 100,000 elements and varying the rate of lookup and range-query operations respectively between 0% to 90%. Both figures show that as the modifications rate is decreased, the throughput of all algorithms increases. In the case where no range-query operations occur (Figure 15-(a)) *Leap-LT* shows between 190% (0% lookup rate) to 260% (90% lookup rate) higher throughput compared with *Leap-COP*. The case where no lookup operations occur (Figure 15-(b)) exhibits similar results where *Leap-LT* shows between 240% (0% range-queries rate) to 200% (90% range-queries rate) higher throughput compared with *Leap-COP*. Note that in the case of 100% lookup and range-query operations rate (not shown here) the *Leap-LT* results are even better. *Leap-LT* is better by 650% and 320% compared to the second best *Leap-COP* in the 100% lookup and 100% range-query cases respectively.

## 3.1 Comparison to Skiplists

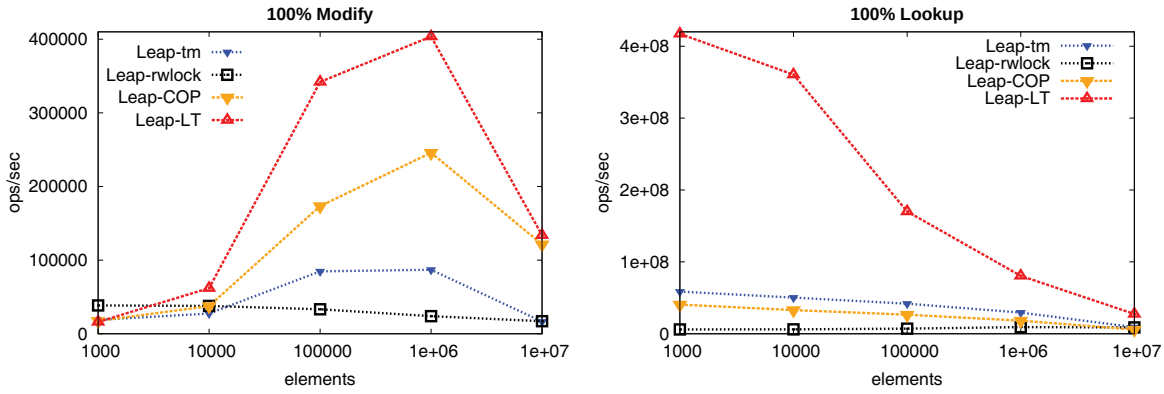
It is natural to compare our *Leap-LT* to the known *Skiplist* data-structure. We compare the throughput of various settings of a single *Leaplist* to: (1) *Skip-tm* - a Skiplist implementation that uses the GCC-TM to synchronize operations; (2) *Skip-cas* - a Skiplist implementation as described in [8]. These implementations store a single key-value pair in each node, and use mutable objects, thus having a lower modify operations overhead compared to our *Leap-LT*. Note that for this comparison we used a single Leaplist data-structure ($L = 1$), and that the range-query operation of the *Skip-*




(a) various threads - 100% modify operations

(b) various threads - 40% lookup, 40% range-query, 20% modify operations

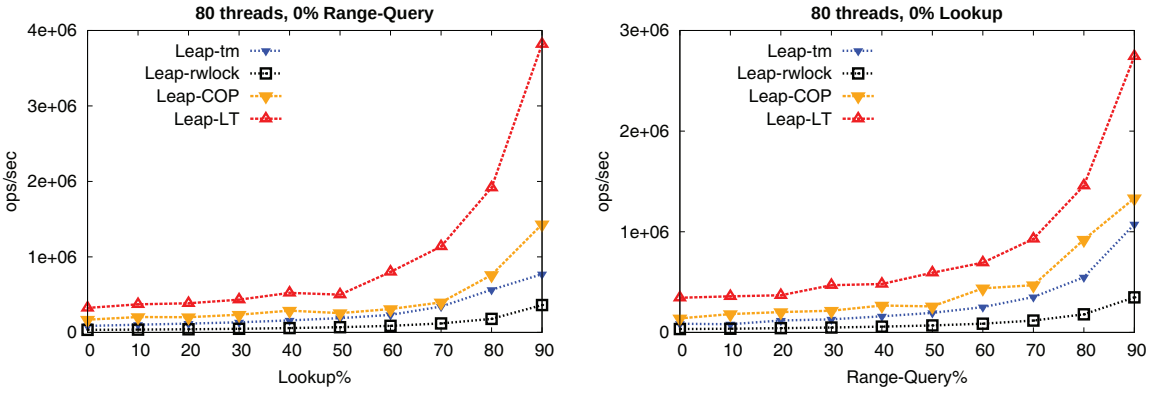
Figure 13: Leaplist size 100K. Workload: different amount of modifications (updates and removes), lookups and range queries. (a) 100% modify operations, (b) 40% lookup, 40% range-query and 20% modify operations.




(a) various total elements - 100% modify operations

(b) various total elements - 100% lookup operations

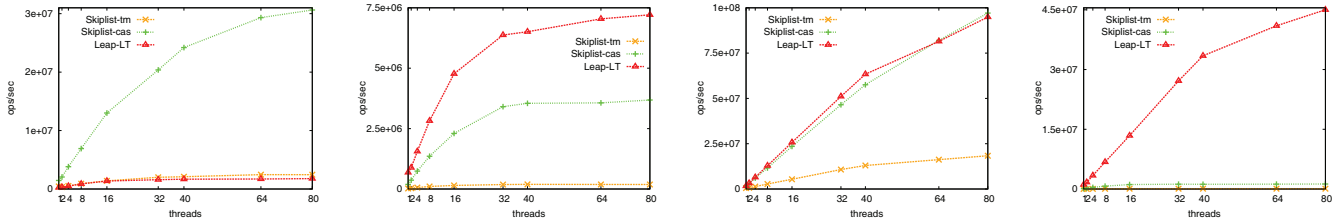
Figure 14: Various total elements number. Workload: different amount of modifications (updates and removes) and lookups. (a) 100% modifications, (b) 100% lookups.




(a) No range-query

(b) No lookup

Figure 15: Leaplist size 100K, 80 threads. Workload: different rates of modifications. (a) 0%-90% lookup and modify operations (no range-query), (b) 0%-90% range-query and modify operations (no lookup).




(a) 100% modify operations (b) 40% lookup, 40% range-query, 20% modify operations (c) 100% lookup operations (d) 100% range-query operations

Figure 16: Leaplist comparison to Skiplists with 1M elements. Workload: different amount of modifications (updates and removes), lookups and range queries. (a) 100% modify operations, (b) 40% lookup, 40% range-query and 20% modify operations, (c) 100% lookup operations, (d) 100% range-query operations.

*cas* implementation does not return a consistent range-query (i.e., this operation is non-atomic and may return an inconsistent result).

Figures 16-(a), 16-(b), and 16-(c) show the throughput when using a data-structure with 1,000,000 elements, and varying the number of threads between 1 to 80. When there are only modify operations (Figure 16-(a)), we observe that both *Skip-cas* and *Skip-tm* are better than *Leap-LT*, and that *Skip-cas* is much better. This is due to the higher overhead of the update and remove operations in *Leap-LT*.

However, we see different results when there are more lookup and range-query operations, as can be seen in Figure 16-(b) where there are 40% lookups, 40% range-queries and 20% modifications. Here we see that *Leap-LT* is up to 2x and 38x better than *Skip-cas* and *Skip-tm* respectively. This is due to the overhead of the range-query operation that needs to iterate many nodes and to the large number of elements which reduces conflicts between concurrent modifying operations.

A workload which exhibits only lookup operations (Figure 16-(c)), shows that *Leap-LT* and *Skip-cas* are comparable and are much better than *Skip-tm*. This is because no contention occurs, and the reduced overhead of the former algorithms produces better throughput.

Figure 16-(d) shows the main strength of our *Leap-LT* implementation on a workload of only range-query operations. It achieves better scalability and up to 35x better throughput on this workload compared to the *Skip-cas* implementation. Moreover, we note that this is achieved while ensuring a consistent operation result (which is not ensured in *Skip-cas*).

# 4. SUMMARY

In this paper we presented a novel concurrent data-structure, *Leaplist*, that provides linearizable range queries. We implemented it using a technique called *Locking Transactions*, which reduces the executed transactions' lengths. We compared different *Leaplist* implementations, and also compared our technique to a *Skiplist* implementation.

We believe that the availability of hardware transactions will greatly enhance *Leaplist* performance because its implementation is based on short transactions. In the future we plan to test the Leaplist in an In-Memory Data-Base implementation, to replace the B-trees for indexes. We believe this can significantly improve the throughput of many Data-Base workloads.

# 5. ACKNOWLEDGEMENTS

The work of the first and second authors was supported in part by NSF grant CCF-1217921, ISF grant 1386/11, DoE ASCR grant ER26116/DE-SC0008923, and by grants from the Oracle and Intel corporations. The third author was partially supported by the Lynne and William Frankel Center for Computer Science, and by ISF grant 1227/10.

# 6. REFERENCES

[1] Gcc version 4.7.0, (http://gcc.gnu.org/gcc-4.7/), Apr. 2012.

[2] Y. Afek, H. Avni, and N. Shavit. Towards consistency oblivious programming. In *OPODIS*, pages 65–79, 2011.

[3] A. Braginsky and E. Petrank. A lock-free b+tree. In *SPAA*, pages 58–67, 2012.

[4] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. Transactional predication: high-performance concurrent sets and maps for stm. In *PODC*, pages 6–15, 2010.

[5] T. Brown and H. Avni. Range queries in non-blocking k-ary search trees. In *OPODIS*, 2012.

[6] L. Dalessandro and M. L. Scott. Strong isolation is a weak idea. In *TRANSACT '09: 4th Workshop on Transactional Computing*, feb 2009.

[7] P. Ferragina and R. Grossi. The string b-tree: a new data structure for string search in external memory and its applications. *J. ACM*, pages 236–280, 1999.

[8] K. Fraser. *Practical lock freedom.* PhD thesis, Cambridge University Computer Laboratory, 2003.

[9] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A simple optimistic skiplist algorithm. In *Proceedings of the 14th international conference on Structural information and communication complexity*, pages 124–138, 2007.

[10] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent tries with efficient non-blocking snapshots. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 151–160, 2012.

[11] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. In *WADS*, pages 437–449, 1989.

[12] O. Rodeh. B-trees, shadowing, and clones. *Trans. Storage*, 3(4):2:1–2:27, Feb. 2008.